Faculty of Sciences
Computer Science Department
Programming Technology Lab

A bottom-up approach to program variation

A dissertation submitted in partial fulfillment of the requirements for the degree of
"Doctor in de Wetenschappen" by

Peter Ebraert

March 2009

Promoter: Prof. Dr. Theo D'Hondt
Co-promotors: Prof. Dr. Patrick Heymans, Dr. Pascal Costanza

To my father
whose passions for football, computer-
games and economy resulted in
many exciting discussions

To my wife
whose endless patience and
support made this dissertation possible

To my mother
whose belief in her children's capacities
is so high that it makes things
like a PhD just happen

Acknowledgements

The foundations of this dissertation were established in 2002, when Theo D'Hondt proposed me to enroll for the European Master in Object-Oriented Software Engineering (EMOOSE). In this master-after-master formation, I was given the opportunity to meet and learn from expert researchers in the domain of software engineering, and more specifically in the field of the separation of programming concerns. Under the guidance of Eric Tanter, I realised that research on the separation of programming concerns was really interesting in a context of software evolution. Thanks Theo and Eric!

With the help of Tom Tourwé and other colleagues from the programming technology lab, I assembled an application for obtaining a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). I was one of the lucky students that were awarded a four year grant for conducting the research described in the application. I would like to express my gratitude to Tom and the IWT!

Of course, a phd student needs a *good* promotor. According to our university, a *good* promotor provides the necessary infrastructure you need to do your research, helps you to find the necessary funding, guides and supervises your research, making sure that your plans are realistic, evaluates and gives feedback on what you have done, puts your ideas in a broader context, pointing out links with other research, suggests methods and approaches to tackle the questions you raise, brings you in contact with other people or publications that are relevant for your work, stimulates you to present and publish your work and will promote and support your work towards the rest of the academic community. I can formally state that Theo complies with all these characteristics and can consequently be classified as a *good* promotor!

According to our university, however, the promotor cannot guarantee funding, does not always have time to see you or help you, expects you to study and work largely on your own, cannot do the research or write a paper for you, has many other responsibilities and expects you to share the burden by assisting in some of these non-research tasks. While Theo always found funding, he was indeed a busy man who had many other responsibilities wearing ties and suspenders. Luckily, Pascal Costanza and Patrick Heymans managed to bridge those moments and acted as *perfect* co-promotors.

I wish to thank the other members of my reading committee, Prof. Serge

Demeyer and dr. Yves Vandewoude for their numerous suggestions which have considerably improved the quality of this text. Further thanks go to the other members of my jury for taking an interest in my work: Prof. Stéphane Ducasse, Prof. Viviane Jonckers and Prof. Ann Nowé.

I thank Jorge Vallejos for the many discussions we had when sharing the office. It are these discussions that form the corner stones of this dissertation. I thank Andy Kellens, Johan Brichau and Kris Gybels for helping me resolve Smalltalk and SOUL issues. A formal thank you to Andreas Classen for his valuable input with respect to the formalisation of this work. I thank Laurent Schumacher for his efforts in increasing the collaboration between the VUB and the FUNDP. I also would like to thank my other colleagues at the Programming Technology Lab for our co-operation over the years. Also many thanks to our secretaries, Lydie Seghers, Brigitte Beyens and Simonne De Schrijver, for their support in helping me deal with the university administration.

Special thanks also go out to Tom Mens and Kim Mens, whose dedication and effort in the software evolution research field resulted in numerous scientific symposia which form the basis of a research network in software evolution. It was on those events that I got to know Patrick Heymans, Serge Demeyer, Stéphane Ducasse and Yves Vandewoude. Without Kim and Tom, it would have been a lot harder to establish this network.

I also want to thank my closest family and friends (in random order) Ellen, Inge, Dimitri, Nele, Ellen, Herlinde, Kaat, Michael, Bart, Kris, Jan, Dave, Evert, Eric, Olivier, Hilde, Stijn, Stefaan, Lies, Katrijn, Kenny, Peter, my colleague football referees, the orcs from the Grombasha clan and the members of the scuba diving club for constantly enquiring how this dissertation was progressing and for occasionally providing me with a welcome distraction from my work.

Last but certainly not least, I wish to thank my parents for allowing me to obtain a higher education and my wife Sabine for unconditionally supporting me. I dedicate this work to the three of you.

Peter Ebraert
March 2009

Abstract

A growing trend in software construction advocates the encapsulation of software building blocks as features which better match the specification of requirements. As a result, programmers find it easier to design and compose different variations of their systems. Feature-oriented programming (FOP) is the research domain that targets this trend.

The thesis of this research is that the software development environment should provide support for recording modularisation information that results from development actions, so that software can automatically be restructured in recomposable feature modules.

This dissertation proposes to model features as transitions that have to be applied to a software system in order to add the corresponding feature's functionality to the system. This model matches the development operations, which are also transitions of a software system. Concretely, this approach enables bottom-up feature-oriented programming and consists of three phases. First, the change operations are captured into first-class entities. Second, these operations are classified into separate sets that each implement one functionality. Finally, those modules are recomposed in order to form software variations that provide different functionalities. Capturing change in frist-class objects can be done in three ways: logging, differentiating or change-oriented programming (a novel programming style that centralises change as the main development entity). Several classification strategies are conceivable. In this work, three classification strategies are used to group the change objects: manual classification, classification based on common characteristics and automatic classification through development operation tagging. The recomposition of the separated modules is performed by producing a totally ordered set of changes that can subsequently be applied in order to produce the desired software variation.

The viability of the proposed concepts, methods and tools is demonstrated by a proof-of-concept implementation that is tested on a text editor that is developed in a standard object-oriented way and reconfigured afterwards to form different program variations.

Keywords: software variability, feature-oriented programming, software classification, software evolution, reverse engineering, object-oriented software development, development environments.

Samenvatting

In het domein van software constructie wordt er meer en meer gebruik gemaakt van een software modularisatie op basis van de functionaliteit. Zulke modularisatie groepeert de bouwstenen van een software applicatie volgens de functie die ze samen implementeren. Het voordeel van die modularisatie in *functionaliteitsmodules*, is dat ze beter aanleunt bij de specificate van de wensen van de gebruikers, aangezien elke functionaliteitsmodule slechts één functionaliteit implementeert. Het resultaat is dat software ontwikkelaars het gemakkelijker vinden om variaties van de software te ontwerpen en te implementeren. Feature-oriented programming (FOP) is het wetenschapsdomein dat deze trend onderzoekt.

De thesis van dit onderzoek is dat een software ontwikkelingsomgeving de nodige ondersteuning moet bieden om modularisatie-informatie te extraheren vanuit de acties van de software ontwikkelaars, zodat de ontwikkelde software automatisch geherstructureerd kan worden in hersamenstelbare functionaliteitsmodules.

Deze verhandeling stelt voor om functionaliteitsmodules te modeleren als transities die op een software systeem moeten toegepast worden om de functionaliteit toe te voegen aan dat systeem. Dit model stemt overeen met de ontwikkelingsacties, die ook transformaties zijn van software systemen. Meer concreet stellen we een nieuwe benadering van FOP voor die bestaat uit drie fasen. Eerst moeten de ontwikkelingsacties geëncapsuleerd worden in entiteiten. Vervolgens worden deze entiteiten geklasseerd in verschillende verzamelingen die elk één functionaliteit implementeren. Nadien kunnen deze modules dan samengevoegd worden om software variaties te produceren met verschillende combinaties van functionaliteiten. Het encapsuleren van ontwikkelingsacties kan gebeuren op drie manieren: loggen, differentiatie of change-oriented programming (een nieuwe programmeerstijl die ontwikkelingsoperaties centraliseert). Er bestaan verschillende classificatiestrategiëen, waarvan wij er drie bespreken: de manuele classificatie, de halfautomatische strategie die gebaseerd is op gemeenschappelijke karakteristieken en de automatische classificatie op basis van annotaties. De hersamenstelling van de modules gebeurt door de verzameling van ontwikkelingsacties te ordenen en toe te passen en heeft als doel een variatie te produceren van het software systeem die overeenstemt met de noden van de software gebruiker.

De naar voren geschoven concepten, benaderingen en werktuigen worden gevalideerd door middel van een bewijs-van-concept implementatie die getest

wordt op een tekst editor die ontwikkeld wordt in een standaard object-gerichte manier en achteraf geconfigureerd wordt om versies te bekomen met verschillende combinaties van functionaliteit.

Kernwoorden: software variabiliteit, feature-oriented programming, software classificatie, software evolutie, reverse engineering, object-geörienteerde software ontwikkeling, ontwikkelingsomgevingen.

Contents

Abstract v

Samenvatting vii

Table of contents ix

List of figures xiii

List of listings xv

List of algorithms xvii

List of tables xix

1 Introduction 1
- 1.1 Software modularisation 3
  - 1.1.1 Object-oriented programming 3
  - 1.1.2 Component-based software engineering 4
  - 1.1.3 Aspect-oriented software development 4
- 1.2 Modularisation for variation 5
  - 1.2.1 Feature-oriented programming 5
  - 1.2.2 Change-oriented feature-oriented programming 6
- 1.3 Bottom-up approach to feature-oriented programming 7
- 1.4 Scope of the dissertation 8
- 1.5 Structure of the dissertation 9

2 Background 13
- 2.1 Program variability 14
- 2.2 Feature-oriented programming 16
  - 2.2.1 FODA diagrams 16
  - 2.2.2 Generic feature-based composition 18
  - 2.2.3 Mixin-layers 19
  - 2.2.4 FeatureC++ 22
  - 2.2.5 Lifting functions 23
  - 2.2.6 AHEAD 25

2.2.7 Discussion 28
2.3 First-class changes 30
2.3.1 Principles 30
2.3.2 VisualWorks: Change List 31
2.3.3 SpyWare 31
2.3.4 CatchUp! 32
2.3.5 Changeboxes 33
2.3.6 Change-impact analysis 34
2.3.7 Discussion 35
2.4 Aspect-oriented software development 37
2.4.1 AspectJ 38
2.4.2 EAOP 40
2.4.3 Logical meta programming 41
2.4.4 Discussion 42
2.5 Conclusions 44

3 Change-oriented programming 47
3.1 Context 48
3.1.1 Evolution scenario 48
3.2 Change as the central development action 52
3.3 Requirements for ChOP 53
3.3.1 First-class changes 53
3.3.2 Change management 56
3.4 Advantages of ChOP 58
3.4.1 Incremental change management 58
3.4.2 Combination with other paradigms 59
3.5 Tool support 60
3.6 Discussion 66
3.7 Conclusions 68

4 Model of first-class change objects 71
4.1 The FAMIX model 72
4.1.1 Basic data types 73
4.1.2 Object 74
4.1.3 Entity 75
4.1.4 Association 79
4.1.5 Argument 80
4.2 Code statements in FAMIX 80
4.3 A model of changes 81
4.3.1 Atomic changes 83
4.3.2 Composite changes 87
4.4 Dependencies between change objects 90
4.4.1 Structural dependencies 91
4.4.2 Semantical dependencies 94
4.5 Conclusion 94

5 Change-oriented feature-oriented programming 97
5.1 Principles . . . 98
5.1.1 Features as functions . . . 98
5.1.2 Changes as feature building blocks . . . 98
5.1.3 Dependencies . . . 99
5.2 Mathematical properties . . . 100
5.2.1 The dependency relation . . . 100
5.2.2 Dependency graphs . . . 103
5.3 Advantages . . . 105
5.4 Bottom-up approach to FOP . . . 106
5.4.1 Obtaining the changes . . . 106
5.4.2 Classification of Changes . . . 107
5.4.3 Change composition . . . 112
5.5 Conclusion . . . 117

6 Formalism for feature composition 119
6.1 Feature Diagrams . . . 119
6.2 A formal model for ChOP . . . 122
6.2.1 Fundamental concepts . . . 122
6.2.2 Properties . . . 125
6.3 From change specification to feature diagram . . . 129
6.3.1 Translating the formalism . . . 132
6.3.2 Applications . . . 134
6.4 Conclusion . . . 136

7 Expressing crosscutting concerns 139
7.1 Crosscutting functionality in feature-oriented programming . . . 139
7.1.1 Flexible features . . . 140
7.1.2 Composing flexible features . . . 141
7.1.3 Other uses for flexible features . . . 142
7.2 Extensional changes . . . 143
7.3 Intensional changes . . . 144
7.3.1 Language for specifying intensional changes . . . 146
7.3.2 Intensional change evaluation . . . 151
7.3.3 Implementation . . . 153
7.3.4 Formalising intensional changes . . . 155
7.3.5 Advantages and drawbacks . . . 160
7.4 Conclusion . . . 160

8 Validation 163
8.1 Proof-of-concept implementation . . . 163
8.1.1 VisualWorks for Smalltalk . . . 164
8.1.2 Model of first-class change objects . . . 165
8.1.3 Obtaining changes . . . 166
8.1.4 Change classification . . . 169
8.1.5 Feature composition . . . 172

8.1.6 ChEOPS supports the formal model 174
8.1.7 Intensional changes in ChEOPS 176
8.2 Validation: FOText 178
8.2.1 FOText design 178
8.2.2 FOText implementation 180
8.2.3 Feature composition 182
8.3 Lessons learned 188
8.4 Conclusion 190

9 Future Work **193**
9.1 Overcoming restrictions 193
9.1.1 Parellel development 193
9.1.2 True scalability 194
9.1.3 The meta-model 194
9.2 Classification strategies 195
9.3 Applications of first-class changes 195
9.4 Formalism 196
9.5 Deriving intensional changes 197
9.6 Feature refactoring 198
9.7 Ensuring design contracts 198

10 Conclusions **201**
10.1 Summary 201
10.2 Contributions 203

Bibliography **207**

List of Publications **217**

Biography **221**

List of Figures

2.1 FODA diagram of the buffer application 17
2.2 Mapping between feature diagram and dependency graph 19
2.3 Collaboration diagram of the graph traversal application (from [48]) 21
2.4 Composition by Mixin-layers (from [48]) 21
2.5 Composing objects by means of lifting functions (from [89]) 24
2.6 Application described by features B and H 25
2.7 Changebox implementation class diagram (from [28]) 34
2.8 The UML diagram of a simple figure editor. 37
2.9 Aspect weaving: composing an application using aspects and base program. 38
2.10 The AspectJ model 39

3.1 Class diagram of the chat application 49
3.2 First change: differentiating users 49
3.3 Second change: ensuring user privacy 51
3.4 Merging the changes 51
3.5 New way of merging the changes 52
3.6 Composable first-class changes design 54
3.7 ChEOPS Architecture 60
3.8 ChEOPS view on change list (ordered by time) 64
3.9 ChEOPS view on change list (ordered by affected entity) 64
3.10 ChEOPS view on change list (ordered by composition) 65
3.11 ChEOPS view on change list (ordered by intention) 65
3.12 ChEOPS change browser 66

4.1 Conception of the FAMIX model (based on [27]) 72
4.2 FAMIX model - Object (based on [27]) 74
4.3 FAMIX model - Entity (based on [27]) 75
4.4 FAMIX model - BehaviouralEntity (based on [27]) 76
4.5 FAMIX model - StructuralEntity (based on [27]) 77
4.6 FAMIX model - Association (based on [27]) 79
4.7 FAMIX model - Argument (based on [27]) 80
4.8 FAMIX model - Statement 81
4.9 Change Model Core 82

4.10 Composable first-class changes design 88
4.11 Push-down field in employee example 90

5.1 Source code (left) and change objects (right) of the `Buffer` 98
5.2 Source code of adding `Restore` (left), `Logging` (middle), `Multiple Restore` (right) 100
5.3 Change objects of `Restore` (left), `Logging` (middle), `Multiple Restore` (right) 101
5.4 Reconstruction of the change sets by means of the differentiation technique 106
5.5 Change model 108
5.6 Classification model 109
5.7 Manual classification 109
5.8 Semi-automatic classification 110
5.9 Automatic classification 111
5.10 Change specification of the buffer example 113

6.1 Buffer feature diagram 120
6.2 Change specification of the buffer example 130
6.3 Mapping optional changes to optional features 130
6.4 Tentative feature diagram representing the buffer change specification. 131
6.5 Buffer feature diagram resulting from the translation algorithm . . 134

7.1 Change specification of the buffer example (copy of Figure 5.10) . 141
7.2 Compositions based on first-class changes 142
7.3 A buffer with a functionality for maintaining statistics 144
7.4 Extended buffer code after adding the statistics feature 145
7.5 Extended logging changes after adding the statistic code 146
7.6 Extended change model 153
7.7 SOUL core 154

8.1 ChEOPS Architecture 164
8.2 Differentiation to obtain change objects 167
8.3 Change-oriented programming to obtain change objects 168
8.4 Change-oriented programming to obtain change objects (view 2) . 168
8.5 FODA diagram of FOText 179
8.6 Class diagram of FOText 180
8.7 FOText: List of logged changes 181
8.8 Composition of all features except for `Logging` 183
8.9 Composition of `Base` and `Save` 184
8.10 Composition of `Base`, `SaveAs` and `Compress` 185
8.11 Composition of `Base`, `Open` and `Compress` 186
8.12 Composition of `Base`, `Open` and `Logging` 188
8.13 Composition of all features and `Logging` 189

Listings

2.1 AHEAD Base feature 26
2.2 AHEAD Restore feature 26
2.3 AHEAD Logging feature 27
2.4 AOSD Base feature 42
2.5 AOSD Restore feature 43
3.1 Evolution scenario *user privacy*: change list by ChangeList 50
3.2 Change method body: extension 56
3.3 Change method body: intension 56
3.4 Chat application: change list by ChEOPS 61
3.5 Adding different users: change list by ChEOPS 62
3.6 Adding user privacy: change list by ChEOPS 62
3.7 Adding user privacy correctly: change list by ChEOPS 63
3.8 Adding user privacy correctly: (compositional) change list by ChEOPS 63
4.1 Change instantiation 87
4.2 Change instantiation example 87
4.3 Change definition example 1 89
4.4 Change definition example 2 89
4.5 Invocative dependency example 93
4.6 Accesive dependency example 94
7.1 Extension of `Logging` on $\{Buffer, Restore, Statistics\}$ 152
7.2 Extension of `Logging` on $\{Buffer, Restore\}$ 152

List of Algorithms

1 $validateComposition(F, CS)$ function 115
2 $minimal_feature_set(F, CS)$ subroutine 115
3 $minimal_change_set(F_{min}, CS)$ subroutine 115
4 $unwanted_change_set(F_{min}, C_{min}, CS)$ subroutine 116
5 $transitive_closure(C_{unw}, CS)$ subroutine 116
6 Constructing a maximal feature set 129
7 $addCandidates$ subroutine 129
8 Transforming a Cs to a feature diagram 133
9 $validateComposition(F, CS)$ function 157
10 $minimal_feature_set(F, CS)$ subroutine 158
11 $minimal_change_set(F_{min}, CS)$ subroutine 158
12 $find_intensional_dependencies(CS)$ subroutine 158
13 $unwanted_change_set(F_{min}, C_{min}, CS)$ subroutine 159
14 $transitive_closure(C_{unw}, CS)$ subroutine 159
15 Clustering change instances based on their timestamp 171

List of Tables

2.1 Analysis of the FOP approaches based on our criteria 29
2.2 Categories of changes 35
2.3 Analysis of the approaches based on our criteria 36
2.4 Aspect composition operators in EAOP 41

3.1 Problems handled by properties of change-oriented programming . 67

4.1 Relations between changes and meta-model entities 85

6.1 Cardinality as a way to describe feature decomposition 120

8.1 Statistics of the size of FOText 181

Chapter 1

Introduction

The subject of this dissertation is the construction of variations of object-oriented software systems. Concretely, we aim at a bottom-up approach in which individual building blocks of the system are first specified in an object-oriented way. These building blocks are then linked together to form larger subsystems, which then in turn are composed to form a complete system. This reflects the sub-domain of software engineering to which this research makes a contribution: the cross-section of object-oriented systems, feature-oriented systems and reengineering. The research behind this dissertation was driven by three observations. Together, these observations show that there is an opportunity to contribute in the research domain of the aforementioned cross-section.

One, feature-oriented programming (FOP) is crucial to the development of software product families. A growing trend in software construction advocates the encapsulation of software building blocks as features (at the producer side) which better match the specification of required functionality (at the consumer side). As a result, programmers find it easier to design and compose different variations of their systems. FOP is the research domain that targets this trend.

Two, the state-of-the-art approaches to FOP (a) are top-down, (b) provide limited control over feature modularisation and (c) require a specific development process. All these characteristics make FOP less easy to use. In a top-down application development method, the designer has to foresee feature modularisation from the start, which is not always possible in practice. The limited control over feature modularisation of the state-of-the art approaches to FOP often requires more effort from the developers to express certain feature modularisations. As feature modularisation is not mainstream, most developers have a development process that is different from the development process required to do feature-oriented software development. Moreover, in order to apply one of the state-of-the-art approaches to FOP, a specific development environment is usually required that is different from the standard one. The use of such environment enforces developers to alter their development habits. Usually, developers are reluctant to change development environment and habits.

Three, the application of FOP to systems that were not developed with feature modularisation in mind, is a tedious task. In order to restructure a legacy system – that was not developed with the separation of functionality in mind – reengineering techniques are needed. Few popular software development environments for object-oriented languages support automated techniques for reengineering to feature modularisation. Therefore, software engineers reengineer manually by browsing the source code. Reengineering code into modularised features involves a huge intellectual effort. Reading and understanding source code of object-oriented programs requires many context switches and a high concentration to keep a mental record of the recovered software architecture: the interrelationships between the architectural components and the rationale behind the software design.

The three observations reveal the need for an alternative approach to FOP for making this simpler: We propose one that captures modularisation information and uses that information to automatically restructure the application into feature-oriented modules. Although the solution to this problem seems obvious – let the developer record modularisation information that results from development actions – this is not easily achieved. First, the lack of proper notation and models for development actions is an impediment to capture the information hidden behind development actions. The prevalent notation and models for describing, building and reasoning about development actions deal with change in an implicit manner only. Second, the modularisation information extracted from development actions should be complemented with domain knowledge from the developer in order to be able to modularise the application correctly. Third, the intrusiveness of the extraction of the modularisation information in the development environment should be minimized. The thesis of this dissertation is that:

Software can be automatically restructured in feature modules if it is developed in a development environment that records fine-grained modularisation information resulting from development actions.

This dissertation introduces a bottom-up method for feature-oriented programming that consists of three phases. First, the change operations are captured into first-class entities. Second, these operations are classified into separate sets that each implement one functionality. Finally, those modules are recomposed in order to form software variations that provide different combinations of functionality. We identify three strategies for capturing change in first-class objects: This can be done by logging developers, by change-oriented programming or by differentiating source code versions. Several classification strategies are conceivable. In this work, three classification strategies are used to group the change objects: manual classification, classification based on common characteristics and automatic classification through development operation tagging. The recomposition of the separated modules is performed by producing a totally ordered set of changes that can subsequently be applied in order to produce the desired software variation.

1.1 Software modularisation

A "software module" is the basic unit of software development, maintenance and management. The main activity of the software design process is the partitioning of the software specification into a number of software modules that together satisfy the problem statement. To do this, programmers need criteria for defining the organization of modules. Major criteria for software modularization include cohesion, coupling and information hiding. All these criteria, however, are difficult to quantify as they mostly depend on specific requirements which are often contradictory (code reusability, code maintainability, system security, system performance, etc.).

Modular programming is a software design technique that increases the extent to which software is composed from separate entities – software modules. Decomposing software into modules allows a *separation of concerns*, and improves maintainability by enforcing logical boundaries between modules. Modular programming can be performed even where the language lacks explicit syntax or semantics to support modules. In fact, all programming paradigms support some level of grouping and encapsulation of concerns into separate, independent entities by providing new abstractions (e.g. functions, procedures, libraries, etc.) that can be used to represent these concerns.

It was only in the mid seventies, however, that programming languages started to isolate modularisation and incorporate specific language constructs for modular programming. Languages like Modula, Ada, and ML allowed the organisation of large-scale systems into software modules, or large-scale organizational units of code. In essence, module systems from the eighties were mostly based on generic programming constructs – generics being, in essence, parameterized modules.

1.1.1 Object-oriented programming

Object-oriented programming (OOP) is a programming paradigm in which programmers not only define the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure. That way, data structures become "objects" that include both data and functions. Moreover, programmers can create relationships between one object and another. For example, objects can inherit characteristics from other objects.

Modularity in OOP is also provided by separating the interface from the implementation. A module interface expresses the elements that are provided and required by the module. Elements defined in the interface are visible to other modules. The implementation, on the contrary, is typically not visible to other modules. It contains the working code that corresponds to the elements declared in the interface.

Simula is the oldest language that has the primary features of an object-oriented programming language. It was created for making simulation programs, in which what came to be called objects were the most important information representation. The first language that really talked about objects was Smalltalk, which was developed in the mid seventies.

One of the principal advantages of object-oriented programming techniques over procedural programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its functionality from existing objects. This makes object-oriented programs easier to modify.

The biggest inconvenience OOP has with respect to modularisation, is that it inherently suffers from the *Tyranny of the Dominant Decomposition* [102]: the program can be modularized in only one way at a time, and the many kinds of concerns that do not align with that modularization end up scattered across many modules and tangled with one another.

1.1.2 Component-based software engineering

Component-based software engineering puts the emphasis on the decomposition of software systems into functional or logical components with well-defined interfaces used for communication across the software components.

The basic building block of a component-oriented application is a *component*. Perhaps the best-known definition of a component is the one given by Szyperski in [101]. It states that "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*". This definition stresses that the components are subject to the black-box principle, that they are subject to (third party) composition and that they are subject to strict and contractually specified interfaces [109].

The biggest inconvenience of component-based software engineering with respect to modularisation, is that it also inherently suffers from the *Tyranny of the Dominant Decomposition* [102].

1.1.3 Aspect-oriented software development

Aspect-oriented software development (AOSD) is a research domain that investigates the increase of modularity by allowing a multi-dimensional separation of concerns. Some concerns defy the forms of encapsulation that are allowed by object-orientation and are called crosscutting concerns because they "cut across" multiple modules of a program. The identification, specification and representation of those concerns and their modularization into separate functional units as well as their automated composition into a working system is the subject of AOSD.

AOSD technology started with aspect-oriented programming (AOP) languages. The term AOP was introduced in 1997 by Gregor Kiczales and his team in [59]. As a follow-up on that paper, they introduced AspectJ: the first AOP language. It has gained a lot of acceptance and popularity from the Java developer community. AOP concepts are based on related research on software modularization, such as composition filters [1], adaptive programming [65] and subject-oriented programming [82] and appear as well in other research domains, such as context-oriented programming [49].

The main strength of AOSD and related approaches is that they all support a multi-dimensional separation of concerns which overcomes the problems that the tyranny of the dominant decomposition brings along. As such, those approaches support a clean modularisation of all the programming concerns.

A fundamental problem with many approaches to software modularisation is that they view systems from the perspective of producers, rather than consumers. Producers tend to specify their systems in terms of *software building blocks* while the consumers tend to specify requirements primarily in terms of *functionality*. This mismatch complicates variability, since there is no direct mapping between a composition of functionality and the software modules that implement that composition, even by using AOSD techniques. Recent research in software construction increasingly reflects a common theme: the need to realign modules around features rather than software building blocks [61].

1.2 Modularisation for variation

A software product family is a set of variations of the same software system. Developing all variations of a software product family can be addressed in an ad-hoc way by implementing one big system that contains all possible variations and which behaves differently depending on its configuration. In a procedural or functional programming style, the resulting code, however, would contain an *IF-THEN* control statement at every place where the program chooses which variant to produce. This kind of implementation lacks *modularity* and *reusability* [77]. An object-oriented implementation would typically use *polymorphism* to implement the variation points. Then, each *IF-THEN* control statement is replaced by instantiating different subclasses of a class which model the specification of each variation. This approach would still require a significant amount of manual labor [34]. The most important drawback of such approach, however, is that it suffers from a combinatorial explosion [70], as for every new funtionality, the number of variations of the product family is multiplied by two (the variations that include the new functionality and the ones that do not include the functionality).

1.2.1 Feature-oriented programming

A better alternative is to modularize a software system based on the functionality it provides. Modules which add functionality to the system are called *feature modules*, further abbreviated as *features*. *Feature-Oriented Programming* (FOP) is the discipline that centralises features as the main development module. Note that in FOP, features can also implement non-functional requirements (such as limited memory consumption). The idea of FOP is to produce software variations by composing feature modules. We find that the state-of-the-art approaches to FOP (E.g. Mixin-layers [98], AHEAD [10], Lifting Functions [89], Composition Filters [12], FeatureC++ [4] and approaches that stem from the AOSD community [59]) all suffer from three common inconveniences.

First, they all start from a *top-down approach* to FOP, in which the software system has to be designed upfront before the implementation is started. In such a method, the designer has to foresee feature modularisation from the start, which is not always possible in practice. This problem is even more apparent in a context of a development method that stresses a highly iterative and incremental development cycle. Such methodologies, such as *agile software development* [20], are often used in order to support *change* during the different phases of software engineering. As they encourage programmers to frequently inspect and adapt their software products, they appear incompatible with a top-down to FOP.

Second, in order to apply one of the state-of-the-art approaches to FOP, a *specific development process* is required that is different from the one exerted by developers. In FOP, code has to be modularised in features: a specific kind of software module. Since feature modularisation is not mainstream, most developers have a development process that does not include modularisation of code into features. Consequently, developers are obliged to alter their development process for applying FOP. As most developers are reluctant to change their development process, they avoid using the FOP programming technique, making it even less mainstream.

Third, we find that all approaches *lack control*. None of them allow the specification of feature building blocks with a granularity *below method level*. Next to that, none of them allow to express the *removal* of building blocks from a base. Examples of where such control is desirable include anti-features, the creation of a demo-application which consists of all features but only to a certain extent, or the customisation of certain features so that the software system copes with specific hardware requirements (e.g. limited memory or computation power). An anti-feature is a functionality that a developer will charge users not to include. E.g. while it may seem more difficult to produce a trial version of a software system that includes publicity, the producers will charge more for a version that does not include the publicity.

In some situations, these characteristics of the top-down approaches to FOP are undesirable as they make it less easy to use. We propose a novel approach to FOP, which is based on a programming style we call *change-oriented programming*. In the following section, we briefly introduce this style and show how it can be used to do bottom-up FOP. Afterwards we elaborate on this novel approach to FOP and hint at how it does not suffer from the three above-mentioned inconveniences.

1.2.2 Change-oriented feature-oriented programming

In [28] and [91], the authors propose to centralise change as the main development entity. We name this programming style change-oriented programming (ChOP). In order to develop a program in the ChOP style, the programmer applies changes which are automatically instantiated in first-class change objects. In order to do FOP by means of ChOP, we first provide the reader with a definition for a feature given by Batory et al. In [7], the authors define a feature as a function that is applicable to a base (a set of building blocks), rather than a set of building blocks itself (which is how features are specified by the majority of the state-of-the-art

approaches to FOP). According to that definition, a programmer can specify a feature by grouping first-class change objects into one function. The composition of features then boils down to the nested application of the functions that represent the features from the composition.

We see three advantages in the specification of features in terms of fine-grained first-class change objects which overcome the three inconveniences we found in existing approaches to FOP: *increased control*, a novel *bottom-up approach to FOP* which is *non-intrusive in the development process and environment*.

In comparison with state-of-the-art approaches to FOP, which allow the specification of features as a set of program building blocks that might extend or modify existing building blocks, our approach allows to specify features as sets of changes that add, modify or delete software building blocks. This approach *increases the control* of how features can be specified in two ways. (a) Features can express changes down to the statement level, which is more fine-grained than the state-of-the-art (only allowing the expression down to the method level). (b) Features can include the deletion of certain building blocks, which is not supported by the state-of-the-art.

The second advantage of specifying features by change objects is that it enables a method for a *bottom-up approach to FOP*. Instead of having to specify a complete design of a feature-oriented application before implementing it (top-down), our approach allows the development of such an application in an incremental way. Also top-down development or a combination of top-down and bottom-up approaches – in which the coarse-grained software structure is made in a top-down way and the detailed parts are constructed in a bottom-up way – is supported by our approach.

Thirdly, this approach is not intrusive in the development process and the development environment that has to be used. In ChOP, a programmer applies changes in his favorite programming language and environment. An external tool instantiates those changes behind the scenes and subsequently groups them in change sets that specify the different features. Moreover, by specifying features as sets of fine-grained change objects, features are not only aware of the building blocks they consist of. They also know how they can be added to a software composition (by applying the changes they consist of). This ensures that a programmer does not have to deviate from his development process in order to do FOP.

1.3 Bottom-up approach to feature-oriented programming

We propose to model features as transitions that have to be applied to a software system in order to add that feature's functionality to the system. This model matches development operations, which are also transitions of a software system. Concretely, this enables a bottom-up approach to feature-oriented programming and consists of three phases.

First, the development operations have to be captured into first-class entities that represent the changes applied for developing the software system. We distinguish between three ways of doing that: by developing the software in a *change-oriented* way, by *logging* the programmer when development is taking place or by *differentiating* two text files that contain the specification (for instance the source code or a UML class diagram) of two different versions of the software system.

Second, these operations are to be classified in separate sets that each implement one functionality. Several classification strategies are conceivable. In this work, three classification strategies are used to group the change objects. A *manual* classification requires a developer to classify the changes by hand. A *semi-automatic* classification technique is based on the principle of clustering changes together based on their common characteristics. The *automatic* classification strategy is capable of automatically grouping changes together based on the extra information that is provided when the system is developed.

Finally, those modules can be recomposed in order to form software variations that provide different combinations of functionality. The recomposition of the separate modules is done by producing a totally ordered set of change objects that can subsequently be applied in order to produce the desired software variation.

1.4 Scope of the dissertation

The main limitation of our approach is that it only scales up if all three phases are automated. In order to support the automatic classification, we require information that denotes which feature the development actions belong to. That information is usually in the developers head when he is producing the software system and is often lost as it is not made explicit. In case *change-oriented programming* or *logging* are used to capture the development actions in first-class changes, the IDE can be instrumented with a dialog that requests that information from the developer. That way, the *modularisation information is made explicit* and an automatic classification is possible.

In order to simplify the approach, throughout this dissertation we limit ourselves to a development setting in which we do not have *parallel development*: We do not allow more than one developer to work on the production of the same software system simultaneously. This restriction brings along two premises: (a) All the first-class change objects are part of a set that can be ordered by the time stamp of the changes. In other words, every change object has a unique time stamp. (b) The IDE can make sure that no conflicting changes are produced. Changes conflict if their application violates an invariant of the programming language or environment.

A final restriction of this dissertation is that it particularly targets feature-oriented modularisation of software systems that are developed by means of *class-based object-oriented programming*. While the concepts of our approach to FOP might be applicable to software systems that are implemented by means of other software development paradigms, we restrict ourselves to this paradigm for proving their applicability. We choose the class-based object-oriented paradigm because

many mainstream programming languages (like Java, C++ or Smalltalk) adhere to it. We feel confident, however, that all three restrictions may be dropped in the future and come back to this issue in Chapter 9.

1.5 Structure of the dissertation

The remainder of this text is structured in the following nine Chapters:

Chapter 2: Background In Chapter 2, we present the background material on the three research domains directly related to this work. First we elaborate on software variability (the ability of a software system to be changed, customized or configured for use in a particular context). Afterwards, we discuss software modularisation for variation and subsequently explain how this can be done by means of feature-oriented programming. We continue by presenting an overview of the related work on the instantiation of change into first-class change objects and conclude with some approaches to the modularisation of crosscutting concerns.

Chapter 3: Change-oriented programming In order to facilitate the recording of modularisation information, we propose a new style of programming, which we call change-oriented programming (ChOP). ChOP centralises change as the main development entity. ChOP developers have to instantiate and apply changes in order to develop their software systems. A software system in its turn, is specified by the sequence of changes that has to be applied to produce it.

Chapter 4: Model of first-class change objects While a model is an abstraction of phenomena in the real world, a metamodel is yet another abstraction, highlighting properties of the model itself. A model is said to conform to its metamodel comparable to a program conforms to the grammar of the programming language in which it is written. The goal of Chapter 4 is to establish a metamodel that allows expressing the evolution of a computer application as first-class objects. We start out from Famix: a metamodel which captures the common features of different object-oriented programming languages needed for software re-engineering activities [27, 30, 105]. We create a model of first-class change classes which is based on the Famix metamodel and incorporate the notion of dependencies between different change operations in the model.

Chapter 5: Feature-oriented programming through ChOP ChOP allows for a bottom-up approach to feature-oriented programming (FOP). In Chapter 5, we explain the details of this approach and clarify it with a small example: a Buffer. We present three techniques for capturing changes, a classification model, three classification techniques of classifying changes and an algorithm for composing and validating change compositions. We introduce the concept of a *change specification*: a definition of a software product

family based on our model of first-class changes. Throughout the chapter, we use the Buffer case to illustrate all aspects of the approach.

Chapter 6: Formalism for feature composition In Chapter 6, we present a formalism of the model behind ChOP. The formalism is based on basic set theory and presents the fundamental concepts and some properties that hold in the context of ChOP. Afterwards, we show that this formalism can be mapped to the better known formalism of feature diagrams (FDs) and present an algorithm that is capable of translating a change specification to an FD. This formal mapping opens up a broad range of applications that were verified in the FD research domain.

Chapter 7: Expressing crosscutting concerns This chapter explains the extension of ChOP with the notion of *flexible* features, which consist of at least one optional change that does not have to be included in a composition in order to make it valid. We show that flexible features provide more flexibility with respect to compositions and that they allow for more than one composition strategy. We expose a weakness of our change model which is caused by the fact that features always contain extensional descriptions of change objects. We present a solution for that issue, and call it *intensional changes*: a descriptive change which can evaluate to an enumeration of changes. We explain how such changes can be used to model crosscutting concerns and show how they increase the robustness to variability.

Chapter 8: Validation In order to validate our work, we implemented the metamodel for expressing ChOP. We instrument an interactive programming environment in such a way that first-class change objects can be instantiated by computer programmers. The tool suite ChEOPS, includes a graphical user interface which allows the visualisation and reasoning over the created change objects. The chapter includes a discussion of the implementation of a text editor called FOText, which we developed using our approach. We validate the thesis of this dissertation by showing that (1) ChEOPS is a development environment that records fine-grained modularisation information resulting from development actions and that (2) the FOText application can indeed be automatically restructured in feature modules.

Chapter 9: Future work There exist many lines of future research with respect to this dissertation. In this chapter, we suggest a handfull. We start by suggesting how the restrictions made to the research setting can be left out and on how that would probably affect the results of this work. Other lines of future work include the study of other classification strategies, other application domains for first-class changes, extensions to the formalism, the automatic deduction of intensional changes (which is similar to aspect mining), the restructuring of feature modules and the enforcement of design contracts.

Chapter 10: Conclusions In the final Chapter we summarize our contributions and make a critical assessment of its advantages over existing work. We

identify the strengths and the weaknesses of our approach and recapitulate our contributions.

Chapter 2

Background

As the title of this thesis already suggests, our work is tightly coupled with the research domain of program variation. Program variations are different versions of one program which provide different combinations of functionality. Constructing program variations is mostly a programming and programming language issue. Program variability itself, however, is also touched upon in the research domain of software evolution.

Starting with an overview of important literature, we summarize the key-characteristics of program variation and discuss how it is achieved by the different state of the art in programming styles. We subsequently discuss feature-oriented programming (FOP) and explain why we consider that programming style in the context of program variation. Afterwards, we evaluate the state-of-the-art approaches to FOP with respect to characteristics that are desired in a context of program variation. This evaluation exposes deficiencies in the current state of the art. We conclude with our own contributions in addressing these deficiencies.

The third part of this chapter briefly describes software evolution. As our contributions are based on expressing features as sets of first-class changes, we present an overview of the state of the art in first-class changes and discuss how those approaches satisfy the desiderata that emerge from the context of program variability. We observe that the changes of one feature module often affect the software building blocks of many other software modules and that their application is similar to the weaving process from aspect-oriented software development. These observations show that change objects exhibit a flavor of crosscutting concerns that is explicitly targeted by the research on aspect-oriented software development (AOSD).

This chapter concludes with a presentation of the state of the art approaches to AOSD. We compare those approaches to our approach in a context of program variation and discuss the similarities and differences. In Chapter 7, we exploit some of the concepts of the AOSD technology for introducing a new kind of change object that uses quantification to denote its impact.

2.1 Program variability

Software users are becoming more and more demanding and cost-conscious. They want specific products that exactly cope with their needs at the lowest possible cost. From the producers point of view, these two requirements are usually conflicting. The development of a specific product for every client takes a lot of time and will consequently be more expensive. The development of a more generic product is cheaper but usually does not exactly cope with the specific needs of the customer.

In order to find the good balance of both requirements, producers tend to use a business strategy called *product lining* [85]: offering for sale several related products of various sizes, types, qualities or prices. The more variations the product line offers, the more specific and expensive its products tend to get. The fewer variations the product line contains, the cheaper and less specific its products become. Adopting this business strategy, the producer's goal boils down to cover the entire scope of the product line, at the lowest possible production cost.

Software companies are the producers of either pure software products or products with an important software component (embedded systems). Driven by users' demand, they are also forced to increase variability of their products. Over the last decade, the management of this variability has become a major bottleneck in the development, maintenance and evolution of software products. Next to that, many companies do not even reach the desired level of variability or fail to do so in a cost efficient manner. An explanation of this can be found in the development approaches used by those companies.

A fundamental problem with many current development approaches is that they view systems from the perspective of producers, rather than consumers. Producers tend to specify their systems in terms of *software building blocks* while the consumers tend to specify requirements primarily in terms of *features*. This mismatch complicates variability, since there is no direct mapping between a composition of features and the software building blocks that implement that composition. Recent research in software construction increasingly reflects a common theme: the desire to realign modules around features rather than software building blocks [61].

Realigning modules around features requires programming constructs that can express the *modularization of features*. Moreover, this realignment often suffers from the tyranny of the dominant decomposition [102] as the program can be modularized in only one way at a time, and the many kinds of functionality that do not align with that modularization end up scattered across many modules and tangled with one another. For coping with that problem, we require a programming language that offers a *multi-dimensional separation of concerns*. In the following section, we elaborate how the state of the art programming paradigms cope with these two desiderata. We focus on FOP, as it claims to target multi-dimensional separation of concerns and feature modularisation at the same time [89].

Now let us present some state-of-the-art approaches related to FOP. While some of them explicitly target both problems, others provide helpful means that can help tackling one of them. We first present a set of criteria for analyzing these

approaches. This list presents the qualitative properties that we aim to find in an approach to program variability. They are motivated by the goals of our work (Chapter 1). Throughout this chapter, we will use these criteria to evaluate the state-of-the-art approaches to FOP.

1. *Granularity*: The granularity provided to express modules establishes the smallest/biggest unit of information that composes a module. Some approaches use native OOP entities, such as classes, instance variables, and methods, as the building blocks of a module. In that case, many of them establish the granularity at the level of methods. This implies that a module would not be capable of introducing a modification *within* a method, but it would be able to replace the entire method. The granularity establishes the kind of specifications that a feature can declare. For instance, an approach that sets its granularity at the level of methods is not capable of modeling a modification within the body of a method. It will consequently have to model such modification as a modification on the method level, which brings along a loss of detail.

2. *Supported operations*: The purpose of this criterium is to evaluate if the approach allows one to express modules by additions, modifications and deletions of program building blocks. Usually, the building blocks that compose a feature module are described in term of additions and modifications. In this dissertation, we also realise that deletion is a proper way for describing certain features such as anti-features [39]. Anti-features are features that remove a functionality from the system. Sometimes, the functionality affected by anti-features is scattered among several feature modules. Consequently, it should be possible to express the deletion of functionality that might be scattered over different feature modules for defining anti-features (and features in general).

3. *Dependency management*: In some settings one module might depend on other modules. Such dependency means that the former module can only be included in a composition that includes the latter modules. While some approaches do manage the dependencies between modules, others don't. Depending on the purpose of the approach, dependencies are managed. In a context of software variability, however, dependency management is desirable as it can be used to validate compositions.

4. *Customized deployment*: In a composition sometimes a feature cannot be deployed since some of its own building blocks have at least one unsatisfied dependency. Some modularisation approaches support a customised deployment of modules, which tackles this situation. Doing so, those approaches allow for compositions that otherwise would be invalid.

5. *Specific language support*: This criterium checks whether the modularisation approach requires the addition of new constructs to a programming language or whether it just uses the capabilities of a main-stream programming language.

2.2 Feature-oriented programming

Feature-Oriented programming (FOP) is a discipline that proposes to produce software instances by composing features. *Features* are well-delimited building blocks that encapsulate the code needed to satisfy a sole user requirement. Such requirement can consist of a functional concern (e.g. an extra functionality) or non-functional concern (e.g. speed optimisation). Since a software instance provides functionality to its users, it can be specified by the list of functionalities that it provides. The required software instance can then be constructed by composing the features that map to the specified functionality.

Note that a one-to-one mapping between features (which denote the system capabilities) and functionalities (which denote the system requirements) is not always feasible. As the mapping between features and functionalities is not the key concern of this work, we do not elaborate on this mapping in this dissertation. For the remainder of the text, we use the term functionality for referring to a requirement and the term feature for referring to a capability.

FOP takes modularization as a central concern, hence several aspects of software quality are improved. FOP increases software *maintainability* [2] by modularizing software. *Reusability* is defined as the ease with which software can be reused in developing other software, which in fact is the way how FOP proposes to develop software at all [22]. FOP proposes to create software variants by adding features to a composition and *reusing* the features that already exist. However, composing features is not free from issues, since there are interactions between features. These interactions make a feature depend on other features. These dependencies must be satisfied otherwise the composition is invalid.

2.2.1 FODA diagrams

Feature-Oriented Domain Analysis [54] (FODA) provides two mechanisms to formally describe applications in a feature-oriented way. It introduces *feature diagrams* to describe all possible combinations of features; and *composition rules* to express constraints that can occur among features. We discuss FODA diagrams because they provide common ground for describing the design of feature-oriented software systems and we use them throughout this dissertation when cases are presented.

We illustrate FODA with an example of a *Buffer* application as shown in Figure 2.1. It shows that a `Buffer` *must* have a `Base` functionality which *optionally* can be enhanced by `Logging` capabilities. The `Buffer` can *optionally* include a `Restore` functionality which can be a `Basic Restore` or `Multiple Restore`. The description is completed adding a *composition rule* which states that `Logging` requires `Restore` and `Base`.

The composition rules of FODA allows us to establish the interaction that can happen between features. Figure 2.1 shows that the `Base` and `Restore` features are required when a the `Logging` feature is included in a composition. This means that the `Logging` feature depends on the `Base` and `Restore` features. Although the number of combinations from a FODA diagram can lead to a combinato-

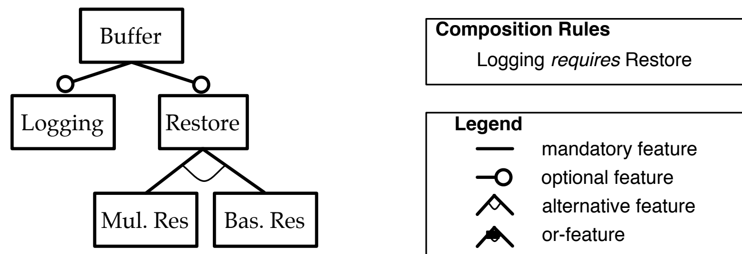

Figure 2.1: FODA diagram of the buffer application

rial explosion, the number of the instances that in fact can be instantiated is a minor proportion of them. In addition to decomposition operators, a feature diagram can be annotated by constraints in a textual language, such as propositional logic [9], that further restrain the set of allowed configurations. Given the industrial-strength *satisfiability solvers* on which most feature diagram implementations are based, this should not be a problem in practice [36].

1. *Granularity*: This is a general purpose model that does allows one to express product lines. In theory, one can use it to modularise methods or even code statements. The idea behind the FODA diagrams, however, is to use the diagrams to modularise on a more coarse-grained level.

2. *Supported operations*: Features can be added to or removed from the FODA diagrams. Some properties, such as features names or relations amongst features can be modified. The building blocks of the features are not relevant in this approach, what makes this criterium not applicable for this appraoch.

3. *Dependency management*: Dependencies between features can be declared in three ways: by imposing the structure of the feature hierarchy (a son depends on its parent), by the composition rules.

4. *Customized deployment*: It does not support a customized deployment of features. Features are fixed and can only be applied or omitted as a whole.

5. *Specific language support*: This criterium is not applicable as this approach only works on the level of feature diagrams.

In [54] Kang defines a FODA diagram as a specification by comprehension of a software product-line. It allows for the description of the relations between features and by that it can be used to derive product instances. However, dependencies between the building blocks that compose each feature are not covered by the feature diagram. We now recall an approach, in which feature diagrams are mapped to artifact dependency graphs, making explicit the dependencies between features and their building blocks.

2.2.2 Generic feature-based composition

In feature-oriented development, the *problem space* of a software product-line is represented by a FODA diagram which maintains all possible combinations that can be produced by a set of features, while the *solution space* is described by the dependency graph of the software building blocks that compose each feature. An artifact dependency graph is a directed acyclic graph, where nodes represent software artifacts while edges represent dependencies between them. Those dependencies can be derived by the abstract syntax tree of the language or introduced explicitly by the developer.

Van Der Storm [108] proposes a model where product instances are derived from a set of features within a software product-line. A feature captures elements of the problem domain while a product configuration consists in choosing features for instantiating a product. A product configuration maps features from the problem domain to the software artifacts that compose the features in the solution domain. We are particularly interested in this approach as it aims at filling the gap between the problem and solution space by linking artefacts in both domains.

The proposed model contains information about the validity of a feature composition, such as whether a feature is incompatible with another feature that is in the same composition. The proposed model is independent of the programming language used, as well as the software development method and the architecture. Filling the gap between problem and solution space model can be achieved by mapping the *problem space* modeled by feature diagrams and the *solution space* modeled by dependency graphs.

The composition of those two representations is proposed as a solution for program variability. By mapping every node of the feature diagram to one or several nodes of the dependency graph a general description of all possible variations is obtained. Figure 2.2 depicts a mapping between the features in a feature diagram – at the top of the picture – and the building blocks that compose features in the dependency graph – at the bottom of the picture. Besides the classic dependencies of the FODA diagrams, a diagram like in Figure 2.2 contains two other kinds of dependencies that are denoted by the arrows.

The dashed arrows denote the mapping between features (from the problem space) and their building blocks (in the solution space). The *Logging* feature for instance, maps to the addition of several $logit()$ statements and an addition of the $logit()$ method. The full arrows denote the dependencies in the solution domain. The $logit()$ method for instance, depends on the $Buffer$ class (that contains the method). Those dependencies can all be used to validate compositions declared within the problem domain.

1. *Granularity*: This is a general purpose model which addresses the gap between the problem space and solution space. The main concern of this approach is to specify the building blocks features consist of. This approach supports granularity down to the level of statements.

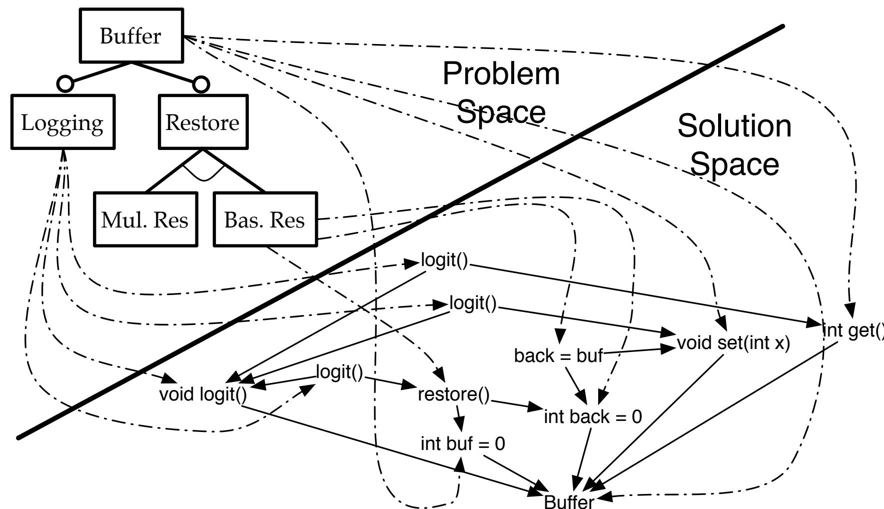

Figure 2.2: Mapping between feature diagram and dependency graph

2. *Supported operations*: This model describes features by the addition of software artifacts. This addition is addressed by mapping a set of software artifacts to the corresponding features.
3. *Dependency management*: This model explicitly targets the problem of composition validity. As such, dependencies between software artifacts are maintained and managed in a dependency graph (such as the one in Figure 2.2).
4. *Customized deployment*: In this model, features cannot be customized in such a way that they are deployed differently depending on the context.
5. *Specific language support*: This model does not require any specific language support. It uses feature diagrams and dependency graphs to describe dependencies between the different software artifacts.

Now we elaborated on two FOP approaches that target the problem space, and one that provides a mapping between the problem and solution space, we dive into the state of the art to FOP that targets the solution space. We start by an approach that targets the modularisation of software building blocks by means of code refinements. It is of particular interest for us, as it provides a basis for feature-oriented software development in a setting of object-oriented programming.

2.2.3 Mixin-layers

The approach proposed by Smaragdakis et al. [98] uses a collaboration-based technique for examining large-scale refinements. A *refinement* adds units of functionality to a software system. It can affect many implementation entities such

as classes, functions, etc. Software components are described by *fragments* of multiple classes which encapsulate fragments of multiple functions. They pursue reusability by using these kinds of components as building blocks of large-scale applications. Large-scale refinements exhibit a *crosscutting* behavior, since they normally impact on many classes at the same time.

A *collaboration* is a set of objects and protocol – which specifies allowed behavior – that defines how objects must interact. The object's *role* in a collaboration is the part of it enforcing the protocol. Usually, objects of an application can participate in many collaborations. A role can be seen as the part of an object that takes place within a collaboration. Thus, collaboration-based design describes applications by composing collaborations. Furthermore, a refinement of an object-oriented class is encapsulated by a subclass, hence it can add new methods and attributes, as well as override methods of the superclass. Class inheritance is not enough, however, to cope with large-scale refinements of a collaboration-based design. A mechanism for grouping behaviour that crosscuts the inheritance hierarchy is required. *Traits*, *multiple inheritance* and *mixins* are three such mechanisms.

In object-oriented programming languages, a mixin is a class that provides a certain functionality to be inherited by a subclass, but is not meant to stand alone. A trait is an abstract type that is used as a simple conceptual model for structuring object-oriented programs [96]. Essentially, both traits and mixins are parameterized sets of methods. They both allow classes to be organized in a single inheritance hierarchy, while they can specify the incremental difference in behavior of classes with respect to their superclass. Unlike mixins, traits do not employ inheritance as the composition operator. Instead, trait composition is based on a set of composition operators that are complementary to single inheritance. Multiple inheritance allows a class to inherit behaviour from more than one superclass. Mixin-based inheritance is a technique based on multiple inheritance that allows a class to inherit behavior from many mixin classes and from only one superclass.

In [98], Smaragdakis proposes to use the concepts of *mixins* and mixin-based inheritance [16] to support feature-oriented programming. Because a mixin only copes with one class while a collaboration contains many classes at a time, he introduces *mixin-layers*. These scale-up mixins so they can handle multiple smaller mixins that are collaborating.

Mixin-layers is a way for expressing large-scale refinements in a collaboration-based collaboration-based design. Since mixin-layers encapsulate refinements which can be applied to produce a program variation, it is a fertile field for software product-lines. Different product line instances might be produced by applying a sole mixin to different layers. Figure 2.3 shows an example taken from [48]. It presents a collaboration diagram which models a graph traversal application that defines three different algorithms on an undirected graph: `Vertex Numbering`, `Cycle Checking` and `Connected Regions`. The application has three classes: `Graph`, `Vertex` and `Workspace` and is decomposed into five collaborations.

Figure 2.4 shows a composition which expresses the development of a vertex numbering application as a series of refinements. Mixin-layers are represented as ovals and contain several mixins that refine the classes.

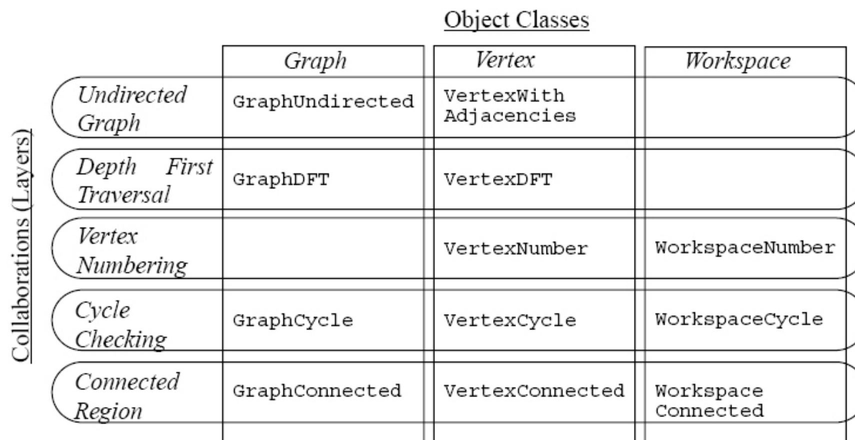

Figure 2.3: Collaboration diagram of the graph traversal application (from [48])

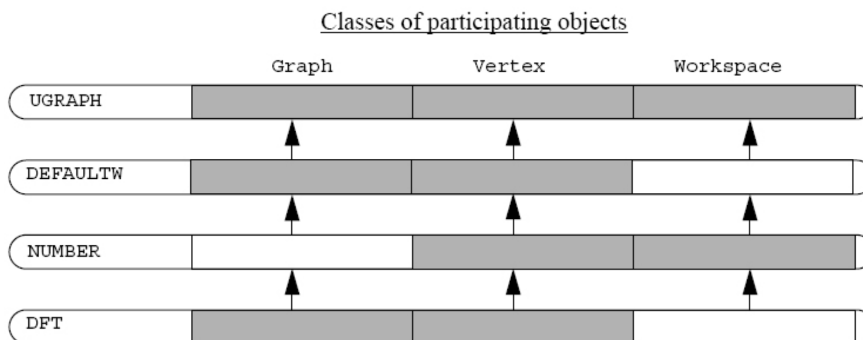

Figure 2.4: Composition by Mixin-layers (from [48])

1. *Granularity*: A *mixin-layer* is able to introduce modifications down to the level of statements. However, is not possible to introduce a statement in the middle of the body of a method. This model addresses method modification by overriding the method with a new definition. Even though this definition is able to invoke the old one (allowing one to maintain the old behavior in an encapsulated way), it does not cover all the possible modifications on the level of statements. A statement, for example, cannot be added somewhere in the middle of a method body.

2. *Supported operations*: This model provides *mixin-layers* as building blocks for building software. Since *mixin-layers* are based on refinements, they only allow for the addition and a specific kind of modification – the introduction of a statement in the middle of the body of a method, is for instance not allowed. It also does not support deletion.

3. *Dependency management*: Dependencies between *mixin-layers* are addressed explicitly.

4. *Customized deployment*: This model does not support a customized deployment of features.

5. *Specific language support*: They introduce *mixin-layers* as a way to encapsulate features. *Mixin-layers* extend an object-oriented language with a construct that models *mixins* and *mixin-layers*.

None of the approaches discussed above support customized feature deployment. Now let us discuss an extension to mixin-layers that uses notions of aspect-oriented software development (AOSD) in order to support that.

2.2.4 FeatureC++

Feature C++ is a language extension – proposed by Apel et al. in [4] – to support FOP and AOP. FeatureC++ implements features by means of *mixin-layers*. Apel et al. define a mixin-layer as a set of mixins that together implement a crosscutting concern. Mixins consist of *constants* or *refinements*. Constants are new software entities and refinements are increments on existing ones. In FeatureC++, mixin-layers are represented as file system directories. Mixins found inside a directory collaborate in the mixin-layer. FeatureC++ improves FOP in two ways: Firstly it enhances refinements with *multiple inheritance* to introduce new behavior to a class by making the class inherit from multiple classes; Secondly it improves refinements to produce generic transformations into classes. Using class and method templates, refinements can be parameterized improving variability. FeatureC++ addresses FOP issues related with crosscutting modularity by means of AOP. The following three concepts are introduced to that extent: *multi mixins*, *aspect mixin layers* and *aspectual mixins*.

Multi mixins. Traditionally, a mixin is able to modify only one mixin. This latter mixin is called the *parent* mixin. A multi mixin is a mixin able to refine a whole set of parent mixins. This is accomplished by introducing a wildcard % in the class or method name where the mixin is specified.

Aspect mixin-layers. The basic idea boils down to embed aspects into a mixin-layer. A mixin-layer contains a set of mixins and a set of aspects, allowing mixins to implement static/dynamic, homogeneus/heterogeneus and hierarchy/non-hierarchy-conform crosscutting behavior.

Aspectual mixins. These provide mixins with the power of AOSD. Aspectual mixins can contain pointcuts and pieces of advice, as well as common refinements and constants. We refer the reader to Section 2.4 for a detailed explanation of those concepts. The analysis of FeatureC++ with our criteria produced the following results:

1. *Granularity*: The level of granularity provided by mixin-layers allows one to manipulate source code at the level of a statement. Although a single statement can only be added to a method by redefining the method in a

mixin, invoking the original method in the new method and adding the new statement before or after that invocation. Statements can only be removed from a method by redefining the method and including all statements from the original method except for the one that has to be removed.

2. *Supported operations*: Features are represented as mixin-layers in the form of multi mixins, aspect mixin layers or aspectual mixins. Thus, a feature is able to introduce new attributes and methods and to modify methods in a limited way. This approach is not able to describe features that delete some building blocks. Consequently, deletion is not supported.

3. *Dependency management*: Feature dependency is addressed by the interaction between mixin-layers which is explicit.

4. *Customized deployment*: Aspectual mixin-layers and aspectual mixins are variants of mixin-layers which introduce notions of AOSD in feature modules. How a feature is deployed, can be addressed by AOSD. It permits deploying a feature in a different manner depending on the context. For instance, a mixin-layer that adds an invocation to all classes where its name starts with `set` would add a different number of invocations depending on the features that are being composed.

5. *Specific language support*: This model introduces the concept of a feature as a language extension for C++.

Software modularisation is not the end goal of feature-oriented programming. After a software system is modularised into feature modules, those modules need to be recomposed in order to construct variations of the software system that provide different combinations of functionality. As such, recomposition is a crucial part of FOP. Before we conclude the state of the art on FOP, we discuss two approaches that target the problems related to the recomposition of feature modules. While the lifting functions approach opts for a pragmatic approach, the AHEAD approach pursues a formal track.

2.2.5 Lifting functions

In [89], Prehofer introduces a model for flexible object composition from a set of features. He proposes a modular architecture for composing features with the required interaction handling, yielding a full object. Inspired by inheritance, the model resolves problems related to feature interactions by *lifting* functions of one feature to the context of the other using *method overriding*. Features are described as a language extension of Java, maintaining the same expressivity as classes. Feature composition is achieved by composing two features at a time, using its previously defined lifter which knows how each feature must be adapted for composing with another one. The process can be executed many times for composing several features. This model is presented as an extension to Java and provides two translations to Java, one via inheritance and another via aggregation.

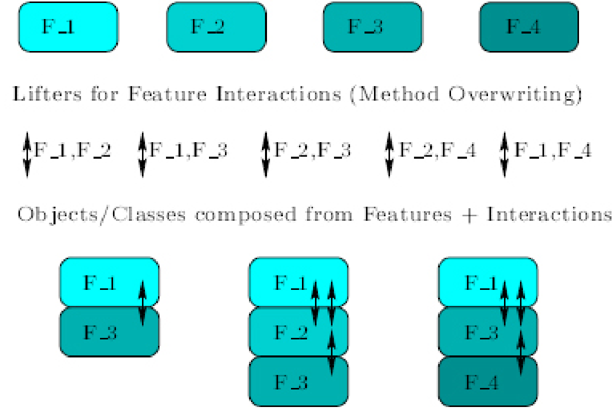

Figure 2.5: Composing objects by means of lifting functions (from [89])

Figure 2.5 was taken from [89]. It shows four features (`F_1`, `F_2`, `F_3` and `F_4`) and three compositions of these features (`F_1 - F_3`, `F_1 - F_2 - F_3` and `F_1 - F_3 - F_4`). To accomplish these compositions, five lifting functions are provided that handle the interactions between two features. In this example, they are: (`F_1,F_2`), (`F_1,F_3`), (`F_2,F_3`), (`F_2,F_4`) and (`F_1,F_4`).

This approach requires intensive human intervention since lifters need to be written for every pair of features which might be composed. Although it increases the accuracy of the resulting composition, the extra work required to construct a lifting functions to address every possible feature interaction seems awkward. With respect to our criteria, the lifting functions approach is summarised as follows:

1. *Granularity*: In this model the smallest entity that a feature can control is a statement. However, the ways in which a feature can add a statement to a method is very restricted. It is addressed by overriding the method with a new version and calling the old version of the method at a certain point. In general this model modifies entities by adding classes, methods and instance variables.

2. *Supported operations*: This model expresses features using the same building blocks as the OOP languages. Moreover, by overriding methods, it allows for the introduction of new behavior. It does not allow for the modification of a specific statement or deletion of any entity.

3. *Dependency management*: Lifters are defined for all pairs of features in a composition. In those lifters, dependencies are covered explicitly since they know which features are adapting the context of the others.

4. *Customized deployment*: This approach does not support the customized deployment of features. However, it offers to adapt the deployment of a feature manually by means of lifters.

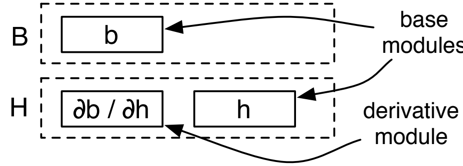

Figure 2.6: Application described by features B and H

5. *Specific language support*: This model extends Java by adding the notion of a feature. A feature introduces a lifting function which addresses the interaction of features.

Feature interactions are a key issue in feature-oriented designs. A feature interaction occurs when one or more features modify or influence other features. Liu, Batory and Nedunuri [67] presented a model which improves FOP by proposing an algebraic theory of structural feature interactions that models feature interactions as derivatives. The following section elaborates on that model.

2.2.6 AHEAD

Algebraic hierarchical equations for application design (AHEAD) is a unified formulation for FOP that integrates step-wise development, generative programming, and algebras [7, 10, 103]. This is based on step-wise development which proposes that a complex program can be built from a simple program (called a base program) by incrementally adding features. AHEAD models product-lines with a simple algebraic structure. A *base module* contains the definition of classes, members, and methods while a *derivative module* extends programs adding fragments of methods, classes, and packages. Thus, a feature is modeled by a composition of one *base module* and a set of *derivative modules*. Figure 2.6 (which was based on a figure of [67]) shows two features `B` and `H`. `B` consists of a *base module* `b` and feature `H` consists of a *base module* `h` and a *derivative module* $\partial b/\partial h$ which extends the *base module* `b`.

`GenVoca` is a method for creating application families and architecturally extensible software by expressing programs as sets of equations. It provides the same information that can be described by a *feature diagram* but adding *cross-tree relations* for denoting relations between features that are not expressed by the diagram. Since it provides a grammar, it can be used by applications in an automatic way.

The function $\bullet$ *weaves* a derivative into a base module. An *introduction sum* $+$ is a binary operation that aggregates base modules by a disjoint set union. Thus, an application composed by features B and H is expressed by:

$$[H(B)] = h + \partial b/\partial h \bullet b \tag{2.1}$$

```
class Buffer{
    int buf = 0;
    int get(){
        return buf;
    }
    void set(int x){
        buf = x;
    }
}
```

Listing 2.1: AHEAD Base feature

```
refines class Buffer{
    int back = 0;
    void set(int x){
        back = buf;
        buf = x;
    }
    void restore(){
        buf = back;
    }
}
```

Listing 2.2: AHEAD Restore feature

A derivative module which *refines* software artifacts – classes, members, and methods – is able to change the behavior of a method defining a new body for the method. This new body can call the execution of the original method at any point of the new method execution. A derivative module which extends fragments introduced by a derivative module – affecting many features – is called a *second derivative module*. If a feature J has a second derivative module which extends the module $\partial b/\partial h$ from H, it is denoted by $\partial^2 b/\partial J \partial h$.

Listings 2.1, 2.2 and 2.3 show a product-line example with three features: `Base`, `Restore`, and `Logging`. `Base` implements the base functionality that include the definition of a `Buffer` class with a `buf` instance variable, a `get` and `set` method for getting and setting the value of `buf`; `Restore` adds an instance variable `back` to store the old value of `buf` any time it is set and adds a method to do the restoration. `Logging` adds a method which prints the value stored in `buf` and `back` and adds invocations in the methods `get` and `set`.

The expression $[L(R(B))] = Logging \bullet Restore \bullet Base$ represents an application with the three functionalities. It could be desirable to compose an application with `Base` and `Logging` features. This is not possible, however, since the `Logging` feature would be making changes to the method `restore()` which would not exist since the `Restore` feature would be not included in the composition. This issue is called the `Feature Optionality Problem` [67]. It can be addressed by

```
refines class Buffer{
    void logit(){
        System.out.println(buf);
        System.out.println(back);
    }
    int set(int x){
        logit();
        super.set(x);
    }
    void get(){
        logit();
        super.get();
    }
    void restore(){
        logit();
        super.restore();
    }
}
```

Listing 2.3: AHEAD Logging feature

restructuring the features in such a way that the feature involved in the interaction is put in a separate feature. This solution allows one to produce an application with *Base* and `Logging` functionality, but has some limitations: Firstly decoupling of interactions requires a deep understanding of the feature implementation which in realistic cases could be difficult to address, secondly a feature can have interactions with many features at the same time.

1. *Granularity*: Features can modify previous software entities by *refinements*. The most fine-grained modification features can introduce by means of AHEAD is at the statement level. It is not possible, however, to modify the body of a method by introducing a statement in the middle. The only way of modifying method bodies, is by creating a new version of the method body and by calling the old body at some point in the new one.

2. *Supported operations*: Features can be described by *base modules* and *derivative modules*. The former allows us to introduce new software entities, the latter allows us to modify a previous entity.

3. *Dependency management*: A feature consists of a base module and a set of derivative modules. The latter can be empty in case that the feature only introduces new entities and does not modify any previous entity. Dependencies are made explicit by derivative modules. A derivative module knows which other module it is modifying and by that introduces a dependency.

4. *Customized deployment*: Although AHEAD can denote the part of a feature which makes a composition invalid and by that restructuring the feature and

encapsulating that part into a new one the adaptation must be addressed with human intervention. This makes it less useful for using in software product-lines.

5. *Specific language support*: AHEAD is a tool for implementing features which extends Java with some extra language constructs (e.g. refinements).

2.2.7 Discussion

Table 2.2.7 summarizes the analysis of all the presented approaches with respect to the provided criteria. It shows that, at the time of writing and to the best of our knowledge, there is no approach that fulfills all criteria. Although, most of the approaches support operations such as addition and a specific kind of modification, none of them allow for deletion. Note that this might be desirable when adding an anti-feature to a system that is already modularised into feature modules. Since the anti-feature might require the deletion of building blocks that were introduced by different feature modules, it must be able to express the deletion of building blocks, rather than the "non inclusion" of certain feature modules in a composition. We conclude that a new approach to FOP should be established that does address the delete operation.

Next to that, most of approaches set the granularity of the supported operations at statement level, but only support it in a restricted way (denoted by "Statement–" in Table 2.2.7). For instance, in AHEAD a programmer is not allowed to insert a statement between two statements of an existing method. With respect to the dependency management, we may conclude that the vast majority of the approaches to FOP, do provide a mechanism for maintaining dependencies between the feature modules. The two final criteria, however, are not covered in an adequate way, as only *FeatureC++* supports a customized deployment of feature modules, and none of the discussed approaches do not require additional language support.

After inspecting the state-of-the-art in feature-oriented programming we acknowledge that all approaches we encoutered intend to encapsulate features in separated modules, but that most of the approaches do it in different ways. There is an awareness that features are not sufficiently described by object-oriented language constructs – classes, methods, etc. That is why all approaches that target the solution space make language extensions. They need new means for expressing the concepts they introduce. It is difficult to imagine writing a program in a feature-oriented way where each feature is perfectly enclosed in a class. We realized a feature normally impacts many software entities at a time. A popular technique that addresses this issue is a *refinement* which is able to introduce modifications to previous language entities. However, refinements lack control, since they cannot describe particular modifications of a software entity. For instance, refinements do not allow for the deletion of a class, method or statement. They only allow for a particular kind of modification of methods based on calling the original method *before* or *after* certain statements of the new one.

	Granularity	Allowed operations	Depend. mgmt.	Custom. deploy-ment	Specific language support
FODA diagrams	Feature	n.a.	Yes	No	n.a.
Generic composition	Statement–	Addition	Yes	No	n.a.
Mixin-layers	Statement–	Addition & modification	Yes	No	Yes
FeatureC++	Statement–	Addition & modification	Yes	Yes	Yes
Lifting functions	Statement–	Addition & modification	Yes	No	Yes
AHEAD	Statement–	Addition & modification	Yes	No	Yes

Table 2.1: Analysis of the FOP approaches based on our criteria

A common trend in feature-oriented programming is step-wise development. That means creating a program by adding features in an incremental way. It has the benefit that all the dependencies of each feature are satisfied, since it will always be applied after the entities which it depends on. In case features are also allowed to include the deletion of building blocks, however, this premise is no longer guaranteed.

Nowadays, software is created by many developers at a time, all of them working in parallel and developing software artifacts that can depend on entities which will be implemented in the feature or in parallel by others co-workers. As such, we advocate that an approach to FOP needs to make such dependencies explicit and that it should manage them from within the features themselves. A feature must know on which other features it depends. Note that in this dissertation, we do not consider a setting of parallel development, but still propose to manage the dependencies in this way in order to allow for the approach to FOP to be applicable in a parallel development setting in the future (see Chapter 9).

In this dissertation, we propose a novel approach to FOP, in which features are specified by first-class change objects that model the development actions taken to implement the features' functionality. This allows features to express refinements that delete building blocks on a fine-grained level of granularity (the statement level). Next to that, this way of specifying features also supports a dependency management, a customized feature deployment without the need of introducing specific language constructs. In the following section, we elaborate on the approaches that also model development operations as first-class entities.

2.3 First-class changes

The goal of our research is to use first-class change objects as the building blocks of the features. For that, we need a model of first-class changes. This section analyzes the state-of-the-art approaches that successfully used first-class change objects in other research domains. As the goal of this section is to find an appropriate model of those first-class changes, we first establish a set of criteria to which we compare the state-of-the art on first-class changes.

2.3.1 Principles

A *change* is a record that captures the information about an adaptation of a software system. A change can be generated by monitoring the developer producing a software system. Such change can be encapsulated in a *first-class entity*, that can afterwards be used as a value in programs without restriction. Some characteristics of a first-class object are: being storable in variables, having an intrinsic identity, being passable as a parameter, being returnable as the result, being instantiatable at runtime, being printable, being readable and being transmissible among distributed processes [18]. Approaches modeling changes as first-class entities exploit these properties to ease the manipulation of changes and the storage of information related to them. A field where this kind of model of changes would be useful is *Software Generators* [8, 52, 5, 6]. Software generators are programs that build other programs.

In this chapter, we present the state-of-the-art approaches we found to model changes as first-class objects. We developed a set of criteria with the aim of analyzing and situating these approaches. These criteria emerge from the context in which our model of first-class changes is used. In a context of change-oriented feature-oriented programming, changes must be able to capture information about adding, removing or modifying the building blocks of a software system at the most fine-grained level possible. As we have seen in the related work on feature-oriented programming, it is also important to maintain dependencies in and between the problem and solution spaces in order to support the automatic validation of program variations. In order to be useable in our approach, the model must consequently be capable of expressing development operations (addition, deletion and modification) on software building blocks (down to the level of granularity of code statements) while maintaining the dependencies amongst those changes.

1. *Supported operations*. A change may consist of additions, modifications or deletions of entities. The aim of this criterium is to establish which operations can be captured by changes.

2. *Granularity*. The subject of a change is the entity that is affected by this change. This criterium analyzes which is the level of the granularity of subjects hat is provided by the approach. While a fine-grained approach allows changes to be expressed till the level of statements, a coarse-grained approach might only permit the expression of changes on classes and methods.

3. *Dependency management*. Changes are modeled as first-class entities, thus they can encapsulate information. This criterium verifies if the approach stores information about the dependencies.

2.3.2 VisualWorks: Change List

Smalltalk VisualWorks records and maintains all changes which are applied to the system in a *Change List* [97]. The *Change List* tool provides a wide variety of operations for reading change files, comparing the contents of a change file, to reorder, remove and replay the changes to the system. This process could end in an error since one operation might require that another operation is applied before it is applied. Some uses are: recovering from a crash, reverting to a prior version and merging several developments into a single environment. *Change List* provides a conflicts view to merge changes applied in different collections, and to construct a single file containing only the desired changes. It can also be an aid in crash recovery, by filtering older changes from a changes file. The Change List tool makes it easy to see the evolution of – and to examine the details of – the code at any stage of its development history. It is particularly useful when we need to see a prior version in order to restore the code.

This approach is of particular interest for us as it is the most basic approach we found to model change as first-class entities that is also integrated in a standard software development environment. In Change List, development actions are monitored and logged into change objects: instances of the `Change` class. That class has many subclasses each modeling a specific kind of development action.

1. *Supported operations*: Changes can represent additions, modifications and deletions depending on which entity is affected. For instance, if a statement is removed, the change produced would describe that a method has been changed.

2. *Granularity*: Changes in *Change List* can represent modifications at the level of methods. Although, it does not provide facilities for detecting changes at the statement level, that kind of changes can be detected by comparing the method before and after the change is applied.

3. *Dependency management*: This approach does not provide dependency management. The developer is responsible for establishing the dependencies by setting the correct order of the changes in the list.

2.3.3 SpyWare

Robbes and Lanza [91] propose a change-based approach to software evolution. In their approach, the interactive development environment (IDE) is instrumented with hooks that enable an IDE plug-in to monitor the developer and to create first-class entities that represent the actions taken by the developer. In their approach, the first-class change entities are objects that capture the history of a system in an incremental way. They are able to reproduce the software which they represent

the history of. When executed, they yield a representation of the program at a certain point in time. They contain additional information interesting for evolution researchers, such as when and who performed the change operation.

The main advantage of this approach in comparison to Change List is that it supports capturing development actions on the level of granularity of code statements. We study this approach as we require that level of granularity in our approach to feature-oriented programming.

As a proof-of-concept they developed *SpyWare*. Spyware is a Squeak[1] implementation which monitors developer activity by using event handlers located at IDE hooks and generates change operations – as first-class entities – from them. Doing so, it is able to provide graphical information for analyzing the evolution of a program.

1. *Supported operations*: Changes can represent additions, modifications and deletions of software entities.

2. *Granularity*: Spyware is a tool that allows for capturing the changes that a developer produces down to the level of statements.

3. *Dependency management*: This approach does not take into consideration change dependencies. It stores the entire evolution history of a program – while it is written – in an incremental way. Thus, the application of a change requires the application of all the previous changes in time. Changes are not intended to be used as modular units that can be applied.

2.3.4 CatchUp!

Henkel and Diwan [46] propose a model for capturing the changes that a developer produces while evolving software libraries. These changes can afterwards be replayed on the software libraries on the client's side reacting accordingly to the corresponding library evolution. In [73] the authors claim that most changes that a developer produces in the evolution of a library are refactorings. Moreover, they state that any change to a software program that preserves behavior can be understood as a refactoring. Using IDE hooks to catch the refactorings introduced to a library, they store that information within changes modeled as first-class entities.

The process starts with the recording phase which occurs when the developer is evolving the library. He introduces refactorings that produce changes which are logged in an incremental way. Then the developer can annotate the changes with semantic information. For instance, a change specifying it has been produced for `Renaming to avoid name clashes`. That is particularly interesting with respect to our approach, as we require an IDE that records such fine-grained modularisation information for validating our thesis. The outcome of this phase is a new version of the library and a file containing a list with all executed changes. In case a refactoring does not affect client code, the change which represents that refactoring can be removed from the list of changes. In order to integrate the new

[1] See http://www.squeak.org/ for more information about Squeak

library version into client code, the list of changes needs to be replayed on the client code.

CatchUp! is presented as a proof-of-concept. CatchUp! is an Eclipse plug-in written in Java. It provides a means for recording library refactorings which are stored as a list of first-class change objects. It allows one to replay the list of changes to a client code updating it for using a new library version.

1. *Supported operations*: This model records changes that produce additions and several kinds of modifications, such as: renaming classes, moving Java elements, etc. It is also capable of capturing deletions such as `remove parameters or remove exception types`.

2. *Granularity*: CatchUp! maintains a list with the kinds of refactorings that it is capable of capturing. It is able to record changes at the refactoring level which can be set at the method level depending on the implementation.

3. *Dependency management*: This approach does not take change dependency into consideration since library evolution is produced in a step-wise way. The list of changes is always replayed in the same way that it constructed.

2.3.5 Changeboxes

Denker et al. propose Changeboxes [28] as a general-purpose mechanism that encapsulates change as first-class entities in a running software system. Figure 2.7 was taken from [28] and shows how Changeboxes are modeled. A `ChangeBox` may specify three kinds of changes: definition, renaming or deletion. `Elements` are the target of a `ChangeSpecification` and model classes, methods or fields. A `ChangeSpecification` of a `ChangeBox` defines how one version of an `Element` may be transformed into another version of that `Element`.

In the context of our research, the most interesting property of the Changeboxes approach is that it has a change model in which the dependencies between change objects are maintained. As those dependencies provide a source of fine-grained modularisation information, we are interested in Changebox' model of changes.

Changeboxes capture the incremental evolution of a software system. They can be linked to `ancestors ChangeBoxes`, that represent prior `ChangeBoxes` to the same system. A `MergeStrategy` defines how two `ChangeBoxes` can be merged. A `ChangeBox` is defined within a `Scope` which allows one to execute multiple versions of a same entity at a time. The system is changed by introducing new Changeboxes which encapsulate a change specification. `CompiledMethod` and `Class` represent specific versions of classes and methods, that later can be used by the virtual machine to instantiate new objects.

A Changebox is an immutable entity that defines a snapshot of a system by encapsulating a set of change specifications, specifying a set of ancestor Changeboxes which these changes apply to and by providing a scope for dynamic execution. A Smalltalk implementation of Changeboxes was successfully evaluated [28]. It illustrates that bug fixes, new features and refactorings can be safely incorporated into a running system without impacting active sessions.

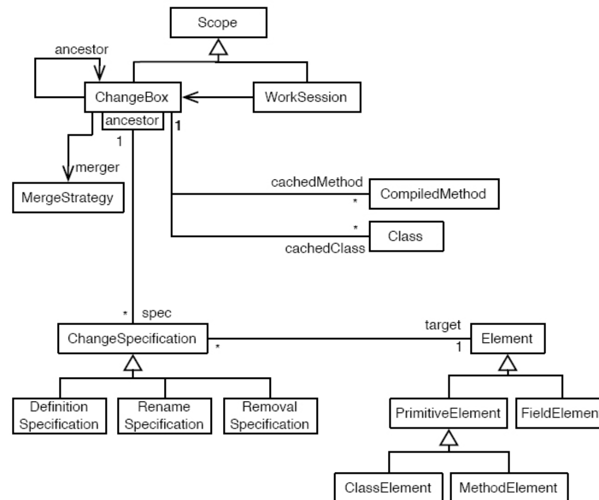

Figure 2.7: Changebox implementation class diagram (from [28])

1. *Supported operations*: This model allows one to define, rename and delete software elements.

2. *Granularity*: The granularity of this model goes down to the level of fields, methods and classes, as we can see in the model diagram of Figure 2.7.

3. *Dependency management*: Dependency is not modeled at the level of the elements that configure a Changebox, but at the level of a Changebox itself. Changeboxes can be related to other Changeboxes by the *ancestor* relationship.

2.3.6 Change-impact analysis

Ryder and Tip [94] elaborate on a collection of techniques for determining the impact that a set of source code changes has on the software system. Source edits are transformed into a set of changes. Table 2.2 presents the kinds of changes that are defined by this approach. For instance, an *LC* change is produced by any kind of source code change that affects dynamic dispatch behavior. A source change may trigger many changes, e.g., the addition of an empty method may imply several changes, of types *AM* and *LC*. They formalized the method dispatch process defining a *Lookup* function. The impact of edit actions on lookup can be monitored.

Just like Changeboxes, this approach also has a model of changes that supports the management of dependencies between change objects. While both models of changes seem very similar, the way changes to the method lookup are handled by this approach is different and the main reason why we present this approach. In contrast to Changeboxes, here, the changes to the method lookup can be detected

by means of dynamic analysis – as we explain below. This allows for the recovery of more accurate information [81] with respect to the dependencies between change objects.

Type	*Description*
AC	Add an empty class
DC	Delete an empty class
AM	Add an empty method
DM	Delete an empty method
CM	Change body of method
LC	Change Virtual method lookup
AF	Add a field
DF	Delete a field

Table 2.2: Categories of changes

In this model, dependency is defined as the interaction between changes that make a change need another one to ensure the compilation success. The introduction of categories for changes establishes a partial ordering between them. This computation is produced automatically.

A set of test drivers $\mathcal{T}$ is associated with a program $\mathcal{P}$. For each test driver t_i in the set $\mathcal{T}$, a *call graph* is produced. Each *node* is a method of the program $\mathcal{P}$ called from the test. Each *edge* corresponds to the calling relationship between the methods of $\mathcal{P}$. The same description applies for an edited program $\mathcal{P}'$. The model is completed with the definition of *AffectedTests* which is a function that returns the test drivers that are affected by a set of changes, and *AffectingChanges* for computing the changes that affect a specific test driver. These functions detect the impact of the changes by traversing call graphs $\mathcal{P}$ and $\mathcal{P}'$.

Although this approach does not state that it models changes as first-class entities, we believe it can be appropriate for maintaining all the information related with changes and to manipulate changes.

1. *Supported operations*: This model supports the addition, modification and deletion of entities.

2. *Granularity*: This approach captures changes related with classes, methods, fields and any modification that affects method lookup.

3. *Dependency management*: This approach takes into consideration the dependency between changes that ensure a correct compilation of the resulting program.

2.3.7 Discussion

The list of related work shows many approaches which successfully modeled changes as first-class entities. All acknowledge that there is a necessity for capturing the operations that are applied when a program is written. Doing so,

the construction of programs can be provided automatically by *software generators* [8, 52, 5, 6].

A first-class change must store information such as: when it was created, who created it, what dependencies it has and specific data related to its nature. It seems very useful to store that information within the change itself. Moreover, the first-class change object can be manipulated, assigned to variables, passed as argument to methods, returned as a result of operations, and so on. A convenient way for doing so is modeling changes as first-class entities.

	Operations	Granularity	Dependency management
Change List	addition, modification, deletion	method	No
SpyWare	addition, modification, deletion	statement	No
CatchUp!	addition, specific modification	statement	No
Changeboxes	addition, modification, deletion	class, method, field	Yes
Change -impact Analysis	addition, modification, deletion	method, field, method lookup	Yes

Table 2.3: Analysis of the approaches based on our criteria

Table 2.3 summarizes the analysis of each approach presented in this chapter based on the criteria we established for a model of first-class changes. In general, most approaches provide a fine granularity at the level of a statement. They also provide means to characterize changes as additions, modifications and deletions. However, only *Change-impact analysis* and *Changeboxes* provide an explicit dependency management model. To the best of our knowledge, we do not find any approach to match all the criteria. Consequently, we choose to develop our own model of first-class changes in which we support the expression of additions, modifications and deletions at the statement level and in which we manage the dependencies between the changes. This model is the subject of Chapter 4.

We observe that, in order to add one functionality to a software application, often changes have to be applied to different object-oriented modules. For instance, when a logging functionality is added to an object-oriented application, changes that add an invocation to a dedicated log method have to be applied to many methods that are scattered over the entire application. This observation shows that the changes of one feature often crosscut the standard object-oriented mod-

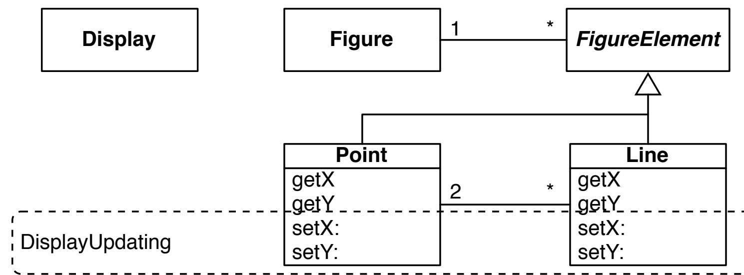

Figure 2.8: The UML diagram of a simple figure editor.

ularisation of an application, and that their application is similar to the weaving process. As the aspect-oriented software development research explicitly targets the modularisation of crosscutting concerns, we now elaborate on the state-of-the-art on aspect-oriented software development.

2.4 Aspect-oriented software development

Some aspects of system implementation, such as logging, error handling, standards enforcement and feature variations are notoriously difficult to implement in a modular way. The result is that code is tangled across a system and leads to quality, productivity and maintenance problems.

Aspect-oriented software development (AOSD) is the research domain that targets this problem, providing ways for a clean separation of crosscutting concerns. A concern is said to be crosscutting if it cannot be cleanly separated into a separate module because it is affecting several modules [37]. As such, AOP aims at a multi-dimensional separation of concerns, which entails breaking down a program into distinct parts called aspects.

Consider the UML class diagram for a simple figure editor described in [37] (see Figure 2.8). There are two concerns for the editor: keep track of the position of each figure element (data concern) and update the display whenever an element has moved (feature concern). The object-oriented design nicely decomposes the graphical element so that the data concern is neatly localized. However, the feature concern must appear in every movement method, crosscutting the data concern. The software could be designed around the feature concern; but then the data concern would crosscut the display update concern.

AOP aims at separating these different concerns into single units called aspects. An aspect is a modular unit with a crosscutting implementation. It encapsulates into reusable modules some behavior that affects multiple classes. Aspectual requirements are concerns that introduce crosscutting functionality in the implementation. Error checking and handling, synchronization, context-sensitive

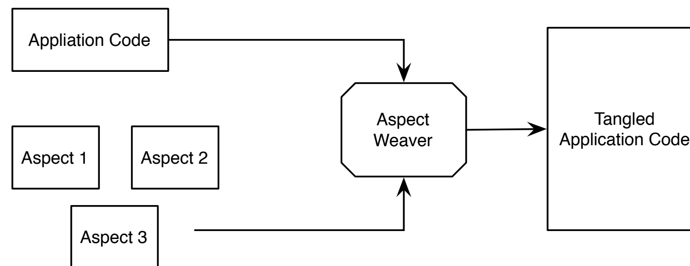

Figure 2.9: Aspect weaving: composing an application using aspects and base program.

behavior, performance optimizations, monitoring and logging, debugging support are all aspects.

Aspect-oriented programming (AOP) is the programming approach that corresponds to the ideas of AOSD. With AOP, each aspect can be expressed in a separate and natural form. After all aspects are declared, a *weaver* combines the aspects set and base-program files into the tangled application code (see Figure 2.9). As a result of this principle, a single aspect can contribute to the implementation of a number of procedures, modules, or objects, increasing reusability of the source code.

In order to enable automatic weaving of aspect code and base-program code, we need some extra entities. Every AOP language has three critical elements for coping with this matter: *a join point model*, *a means of identifying join points*, and *a way of affecting the implementation at join points* [37, 57]. A join point is a particular point in the program structure or the execution of the program.

The join point model provides the means to describe the joint points – the points where enhancements should be made. It also provides a mechanism to express sets and subsets of join points, to express common behavior. For that, each AOP language must have a join point definition syntax. After the join points are defined, we need to alter the implementation at those point in order to insert the crosscutting behavior. As we said above, this is also done by a weaver. We now present the best-known implementations of AOSD and discuss them in a context of software variability.

2.4.1 AspectJ

Currently *AspectJ* [58] is the most widely used AOP language. AspectJ is an aspect-oriented extension to Java that enables the modular implementation of a wide range of crosscutting concerns. Each of those crosscutting concerns is modularized by an *aspect*.

AspectJ adds to Java just one new concept, a *join point* – and that's really just a name for an existing Java concept (an event that occurs at runtime). It

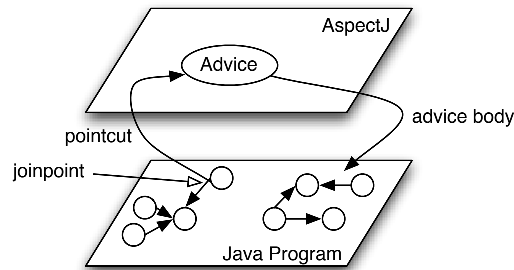

Figure 2.10: The AspectJ model

also adds to a few new constructs: *point cuts*, *advices*, *inter-type declarations* and *aspects*. *Join points* state the points in the base program where we want aspects to take control. They can be declared on operations like a method call, a class boundary, a method execution, an access or modification of member variable, exception handlers or on static and dynamic initialization.

Many join points can be assembled into a set of joint points – *point cuts*. To each of the point cuts, we then attach a piece of *advice*, which states the actions to take when one of the point cuts is reached. The body of the advice is specified in ordinary Java code. The *inter-type declarations* allow the programmer to modify a program's static structure, namely, the members of its classes and the relationship between classes. *Aspects* are the unit of modularity for crosscutting concerns. They behave somewhat like Java classes, but may also include point cuts, advice and inter-type declarations.

In AspectJ, aspect composition – or weaving – is done at load-time. Whenever a class is loaded into the VM memory, its byte is first adapted in order to incorporate the applicable pieces of advice. This results in modified byte code that contains both the class behaviour and the pieces of advice that impact on that behaviour. This modified byte code is actually loaded into the memory of the Jave VM. At runtime, when at a certain point in the execution, a pointcut holds, the attached piece of advice is executed, as we can see in Figure 2.10.

The base code and the aspect code must be written using an external Java editor. The AspectJ development tools for Eclipse (AJDT) is an editor that is written as a plugin for the Eclipse IDE and which supports the developer in doing AOSD in AspectJ. Instantiations of the AspectJ model where conceived for many other programming languages. *AspectR* [2], *AspectC* [3], *AspectC++* [41], *AspectS* [4] and *AspectS* [5] are AspectJ-like implementations of AOP for respectively Ruby,

[2]AspectR: http://aspectr.sourceforge.net/
[3]AspectC: http://www.cs.ubc.ca/labs/spl/projects/aspectc.html
[4]http://common-lisp.net/project/aspectl/
[5]AspectS: http://www.prakinf.tu-ilmenau.de/ hirsch/Projects

C, C++, Common Lisp / CLOS and Smalltalk. *Apostle* [6] is another AspectJ-like aspect-oriented extension to Smalltalk.

As we have stated before, implementations of aspect-oriented programming always incorporate two steps. The first is the decomposition of crosscutting concerns in aspects. The second is the recomposition of those aspects into one application. Though problems may rise doing this recomposition, as conflicts between the separate aspects may occur. When two or more aspects are impacting on the same point of the base program, an aspect composition should be defined in order to avoid conflicts. This composition states the way in which the aspects and the base program are supposed to work together in order to form an application with the desired behavior.

Support for such compositions is particularly interesting in the context of software variability, as program variations are to be produced by recomposing several separated aspect and class modules. This is where AspectJ falls short, as aspect reuse is limited, woven programs can have hard to predict behavior, modular reasoning using aspects is difficult, and step-wise development of programs is error-prone [68]. An AOSD approach with better support for aspect composition is discussed below.

2.4.2 EAOP

Event-based aspect-oriented programming (EAOP) [29] is another AOSD approach. In EAOP, a global monitor keeps track of all base-level *events*. That way, the monitor is always able to execute something extra when a certain event – operation – occurred. In such approach, it is also very easy to keep track of control flows because all information is centralized. The monitor can in that way be seen as one big metaobject which is responsible for managing the execution of the base program and the aspects. An aspect can be seen as an event transformer. In fact – in EAOP – an aspect is a Java object which takes an event as a parameter, performs some computation or action (which may modify the argument event), and waits for the next event.

In [29], the authors describe how a developer can resolve aspect composition conflicts. This is done using a three phase model. The first step is to write all the base and aspect code. The second phase holds the conflict detection. The last phase includes the conflict resolution. The second and third phase can be iterated on, till composition rules are specified that handle all composition conflicts.

A conflict occurs when two (or more) aspects interact with each other, i.e. if at least one of their crosscuts match the same join points. But in fact, this definition is too strong, because it is possible that there is no conflict at all, and that the aspects could be executed one after another without affecting each other. Therefore the authors distinguish between two kinds of dependence: strong and weak independence. Two aspects are said to be strongly independent if none of their crosscuts have common join points. Two aspects are said to be weakly independent if they have crosscuts with common join points but if the aspects

[6] Apostle: http://www.cs.ubc.ca/labs/spl/projects/apostle/

themselves can be composed to a single aspect. This composition can be done using one of the composers presented in Table 2.4.

Composition	*Semantics*
A_1 seq A_2	When both aspects have a common cross point, first A_1's actions and then A_2's actions will be applied.
A_1 fst A_2	Propagates the execution control to A_1, and, if and only if A_1 did not detect a crosscut, the execution control is given to A_2.
A_1 any A_2	Propagates the execution control to A_1 and A_2 in an arbitrary order.
A_1 cond A_2	Propagates the execution control to A_1, and, if and only if A_1 detects a crosscut, the event is forwarded to A_2.

Table 2.4: Aspect composition operators in EAOP

Another problem which is apparent in AOSD approaches like AspectJ and EAOP is that they suffer from pointcut languages where pointcut declarations result in a high coupling between aspect and base system. Additionally, these pointcuts are fragile, as non-local changes easily may break pointcut semantics. These properties are often described in literature ([100, 99, 60, 56]) and form an obstacle in the context of change-oriented feature-oriented programming, in which such changes are continuously applied to develop software products. This problem can be tackled by means of logical meta programming.

2.4.3 Logical meta programming

In [23, 26, 25, 24] a logical metalanguage (SOUL or TyRuBa) is presented that can be used to specify aspects. Logical rules are defined in order to detect the affected join points. When a join point is found, the code of the piece of advice of the aspect is inserted in the base code. When all rules have been applied on the base program, the new source code (extended with all aspects code) is available to be ran. Using this idea, one can express aspects as logical rules and compose them by producing new rules that call some of the logical rules in a certain sequence.

CARMA [45] is a language for AOP which makes use of logic meta programming for the specification of crosscuts. The use of a crosscut language based on logic meta programming provides CARMA with several advanced features. From logic programming it gets the use of unification as a more advanced wildcard mechanism than what is supported in other crosscut languages, the use of logic rules for writing reusable crosscut specifications, and the use of defining multiple rules for the same predicate for writing variants of a crosscut specification. From logic meta programming it inherits capabilities for writing crosscut specifications based on structural properties of the program being crosscut.

Intensive [56] is another AOP approach that uses logical meta programming. Intensive targets the documentation of software programs with design contracts and allows one to express model-based pointcuts. This model-based pointcut mechanism strives to decouple the actual pointcut definition from the implemen-

```
class Buffer{
    int buf = 0;
    int get(){
        return buf;
    }
    void set(int x){
        buf = x;
    }
}
```

Listing 2.4: AOSD Base feature

tation structure of the concepts in the source code which the pointcut relies on as well as to render the causal link between these concepts and the implementation structure explicit and verifiable. The mechanism allows for the expression of pointcuts in terms of a conceptual model: a first-class reification of the different concepts in the source code which a pointcut relies on. As such, design contracts can be imposed on the conceptual model, aiding in keeping this conceptual model synchronised with the source code.

2.4.4 Discussion

Of course there are many more implementations and technologies related to AOSD, but it is not our purpose to list all implementations, but just to give the reader an idea of the current success of AOSD and on its integration in different programming languages. Precisely that will be the challenge for all AOSD languages despite the fact that its solutions are likely to be platform specific. Figuring out how best to build usable and extensible tools is a problem that any serious programming language project faces – it is impossible to judge the usefulness of a programming language in the absence of real developers using it, and it is impossible to convince real developers to use a language without high quality tools [51].

AOSD introduces the concept of a *pointcut* as way to match the places of program code where the concern occurs. It also introduces pieces of *advice* as an artifact that specifies code to introduce behavior at the places matched by the pointcut. Advices are *woven* into the program at the point of the *pointcut*. Doing so, a crosscutting concern can be stored in a modular way making the program's code easier to understand and to maintain.

Listings 2.4 and 2.5 show an example of a `Buffer`, which is implemented with AspectJ [58]. It introduces a `Buffer` class that specifies a `buf` instance variable and methods `set` and `get`, and an aspect that adds a `back` instance variable, a method `restore` to restore the value of `buf` and that introduces a statement *before* the execution of method `set`. Note that the advise corresponds to the set of changes that needs to be applied in order to add the aspect to the base code and that the pointcut refers to where those changes have to be applied.

```
public aspect Restore{
    int back = 0;
    void restore(){
        buf = back;
    }
    public pointcut set(int x): target(x)
        && execution(void Buffer.set(*));
    void before(int x); set(x){
        back = buf;
    }
}
```

Listing 2.5: AOSD Restore feature

Although AOSD is a general purpose approach, we believe it is an appropriate technique to modularise features. Aspects can implement features and the weaving phase implements the composition of the features to the base program. Moreover, the composition language and framework of the EAOP approach, show that it is possible to generate program variations by composing class and aspect modules.

When applying AOSD to do feature-oriented programming (FOP), programs end up consisting of aspect-oriented programming artifacts such as classes and aspects. Doing so, a set of classes and aspects can specify a feature. FOP requires composing features to produce a software instance. Although AOP provides means to weave aspects with classes and by that produce a valid application, it does not provide means to specify the set of features that is desired to be composed. From a FOP point of view, we realize that AOSD always composes all features that are specified in the environment, not allowing one to select a subset from them. This hinders variability, which is required in the context of our research. Now we discuss our findings with respect to the desiderata that we identified in the beginning of this chapter.

1. *Granularity*: In general AOP is able to introduce modifications at any level of program entity. However, there are AOP implementations, such as AspectJ [58], that do not provide a granularity beyond the level of methods.

2. *Supported operations*: Features implemented by means of aspects can express addition and restricted modification to a base program. Only very few AOP approaches (like CARMA) allow for the specification of aspects that modify the body of a method. Most approaches, however, only support modifications *before*, *after* or *around* a method and not within a method.

3. *Dependency management*: The feature dependency in terms of dependency between aspects and the base program is addressed by the pre-compiler which weaves aspects into the base program. Consequently, not all approaches provide a dependency management.

4. *Customized deployment*: Pointcuts allow for the matching of places of a program by pattern matching and quantification. Doing so, the places where an aspect may introduce behavior depend on the elements present in the program at the moment that the aspect is woven. Thus, features by means of aspects provide a customized deployment.

5. *Specific language support*: AOP enhances OOP by introducing new concepts. Many implementations provide AOP capabilities for existing OOP languages.

2.5 Conclusions

We started this chapter by explaining that the context of this dissertation is program variation. We advocate feature-oriented programming (FOP) as the right development technique for modularising software in recomposable modules as it aims at modularising software systems in feature modules that each implement a different functionality [89]. While features refer to entities within the solution domain (software capabilities), functionalities refer to entities in the problem domain (software requirements).

We elaborate on what properties we require from approaches in order to support program variability. From our analysis of related work in previous sections, three important deficiencies were found in current systems for providing program variability:

- *Strictly top-down approach:* None of the existing approaches supports the development of software systems that are modularised in a bottom-up way. In other words, all approaches require a developer to design the system in a modular way before the implementation is actually started.

- *Specific development process:* All of the existing approaches require a specific development process in which the program is first designed in an aspect-oriented or feature-oriented way, after which it is modularised into base entities, aspects and features. As such, the developers are required to alter their development habits. Moreover, most of the approaches enforce the use of a specific development environment in order to be able to use it.

- *Limited control over feature modularisation:* None of the existing approaches supports the expression of a feature that deletes some building bocks from an application. Also most of the approaches only allow a feature to contain building blocks at the level of classes and methods and do not allow a feature to express building blocks at the level of statements.

In this dissertation, we propose an alternative approach to program variation, which aims at overcoming the deficiencies we found in the state of the art in approaches to feature-oriented programming. It is based on expressing features as sets of first-class change objects. For that, the second part of this chapter contains an overview of approaches that already provide models of first-class changes. We compare those approaches and discuss the granularity at which they allow changes

to be expressed, which operations they provide and whether or not they maintain the dependencies between the change objects. We conclude that none of the state of the art approaches that propose first-class changes satisfy all three criteria. For that, we decide to work out our own model of first-class changes. This model is presented in Chapter 4.

We observe that the first-class changes of which features consist often impact the building blocks of multiple other software modules. Hence, our features modules exhibit a crosscutting behaviour. We elaborate on the state of the art on aspect-oriented software development (AOSD) as that is the research domain that explicitly targets the problem of modularising crosscutting concerns. We conclude that – just like the AOSD approaches – our approach provides a mechanism for modularising crosscutting behaviour and that the application of features corresponds to the AOSD weaving process. One difference between our approach and AOSD is that the vast majority of AOSD approaches supports quantification for declaring impact points for the pieces of advice, while the expression of features as change sets does not include quantification. In Chapter 7 we expand our approach with quantification, and as such allow for the specification of a change that quantifies its impact points.

Chapter 3

Change-oriented programming

The thesis of this dissertation is that the software development environment should provide support for recording modularisation information that results from development actions, so that software can be automatically restructured into recomposable feature modules. In order to facilitate the recording of modularisation information, we propose a new style of computer programming, a new programming paradigm: change-oriented programming (ChOP).

ChOP targets software evolution and centralises changes as the main entities in the software development and evolution process. In ChOP, software programs are developed by applying *changes* and can afterwards be evolved in the same way: by applying changes to them. Some examples of developing code in a change-oriented way can be found in most interactive development environments (IDE): the creation of a class through interactive dialogs or the modification of the code by means of an automated refactoring. ChOP goes further, however, as it requires all building blocks to be created, modified and deleted in a change-oriented way (e.g. adding a method to a class, removing a statement from a method, etc).

This chapter is structured as follows. Section 3.1 presents in what setting ChOP should be considered and what its main benefits and drawbacks are. In Section 3.2, we present the programming paradigm and show what we mean by centralising change as the main development entity. Section 3.3 shows that we need both a model of first-class changes and a change management system in order to enable ChOP. Change- and Evolution-Oriented Programming Support (ChEOPS) is a proof-of-concept implementation of ChOP and is introduced in Section 3.5. A discussion of ChOP and its related work can be found in Section 3.6. We conclude this chapjter in 3.7.

3.1 Context

Some interactive development environments (IDEs) include a tool to manage the changes applied to a software system. VisualWorks for Smalltalk, for instance, contains *ChangeList* [50, 107]. Using this tool, programmers can inspect, compare, edit and merge changes applied to classes or methods. Each Smalltalk image[1] holds one single change list which records all the performed changes on that image. Hence the user may recover from a crash by backtracking to the most recent non-erroneous state of the image and reapplying changes listed by the ChangeList tool.

Throughout this section, we consider the ChangeList tool as a case to demonstrate the need for change-oriented programming. In ChangeList, changes are modelled as standard Smalltalk objects and as such they can be referenced, queried and passed along. Every time a system is modified, the Smalltalk IDE logs this modification by creating a change object for it, linking this object to the project, and adding the object to the list handled by ChangeList.

Change objects of the ChangeList tool properly satisfy the notion of first-class changes for software evolution defined by Robes and Lanza [91]. According to them, change objects provide more accurate information about the history of a program than *file*-based and *snapshot*-based versioning change management systems. Timestamps or instance, are more precise as they are not reduced to commit times. Change objects also better represent the incremental and entity-based way in which the evolution of a program occurs (changes are expressed in terms of system entities and not over text). However, change objects still suffer from a number of shortcomings which we illustrate in the following scenario.

3.1.1 Evolution scenario

Consider the scenario of software evolution for the chat application depicted in Figure 3.1. This application originally consists of two classes, `User` and `Chatroom`, which respectively maintain a reference `cr` and `users` to one another. A user can subscribe to a chatroom using the `register` method and exchange messages with the rest of users of the chatroom using the `send` and `receive` methods. Messages sent to the chatroom are propagated to all the registered users.

Assume that there are two developers working on new features for this application. The first developer is responsible for introducing *types* of users so that it can be possible, for instance, to differentiate between registered and guest users. The second developer is responsible for ensuring the privacy of the users, which in this case corresponds to encrypting and decrypting the messages when they are sent or received.

For the first change the developer adds two subclasses of `User` to the application program, `RegisteredUser` and `Guest`. In this example, the only difference between these two types of users is that the registered users can be identified by

[1] Most Smalltalk systems represent the application code (for example classes) together with the application state (objects) in a single memory region called the *image*. Images can be saved and loaded by the Smalltalk environment as a snapshot of the current code and state of a program.

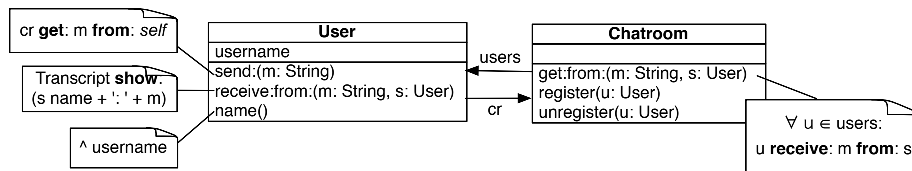

Figure 3.1: Class diagram of the chat application

their name in the chat room whereas the guests cannot: Accordingly, the `username` attribute of `User` class is moved to the `RegisteredUser` class. Figure 3.2 shows this first feature added to the application.

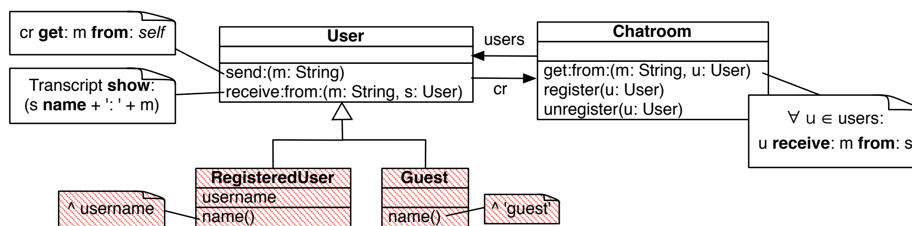

Figure 3.2: First change: differentiating users

To ensure user privacy, the second developer adds two methods to the `User` class, `encrypt` and `decrypt`, which are called from the `send` and `receive` methods respectively. Figure 3.3 shows this second feature added to the application.

After implementing these two new features in the application, the developers merge their changes, resulting in the class diagram showed in Figure 3.4.

Listing 3.1 illustrates how these three steps are registered by the ChangeList tool in Smalltalk. Each line in this list corresponds to a change required for the implementation and merging of the two new features described above. Each of these changes is stored in a change object. We observe a series of problems in this representation of the evolution of the chat application:

Restricted level of granularity The entities contained in the change list are restricted to a granularity of classes and methods. Additions or removals of attributes or statements within method bodies are not captured by dedicated change objects. The changes within a method body result in a change that redefines the complete method. This is what happens, for instance, with the `send:` method which is first defined and then modified. As changes within methods are not logged by dedicated changes on the statement level,

```
Created package ChatApp
  define User
  doIt User organisation addCategory:#messaging
User receive:from: (change)
  define User
  define User
User send: (change)
  doIt User organisation addCategory:#accessing
User name (change)
  define Chatroom
  define Chatroom
  doIt Chatroom organisation addCategory:#messaging
Chatroom get:from: (change)
  doit Chatroom organisation addCataegory:#registering
Chatroom register: (change)
Chatroom unregister: (change)

define Guest
  doIt Guest organisation addCategory:#messaging
Guest name (change)
  define RegisteredUser
RegisteredUser name (change)
User name (remove)
  define User
  define RegisteredUser

  doIt User organisation addCategory:#encryption
User encrypt: (change)
User decrypt: (change)
User send: (change)
User receive:from: (change)
```

Listing 3.1: Evolution scenario *user privacy*: change list by ChangeList

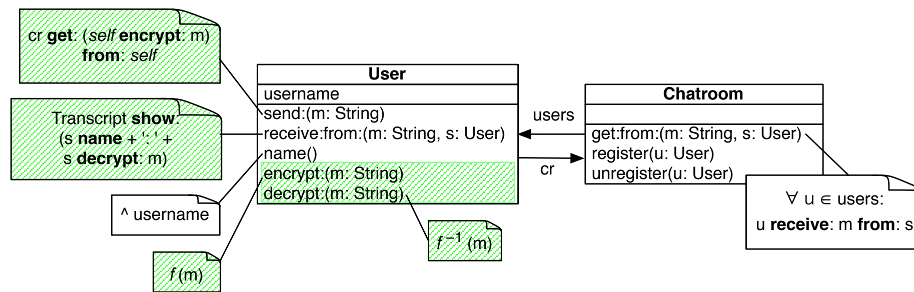

Figure 3.3: Second change: ensuring user privacy

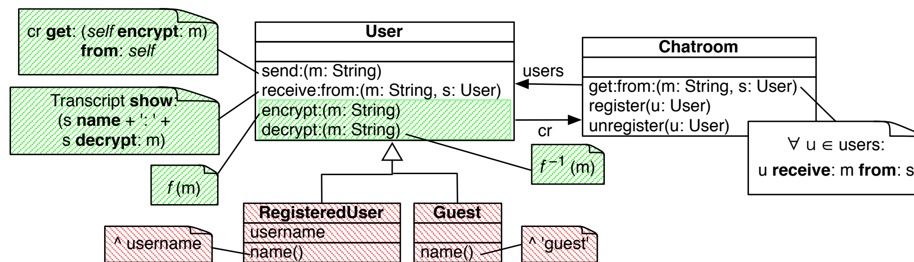

Figure 3.4: Merging the changes

these two different kinds of changes result in two occurrences of `User send: (change)` on lines 7 and 30.

Term overloading The definition of a change object is in some cases ad hoc and inconsistent. The same kind of Smalltalk change object can represent several kinds of modifications. For instance, a `ClassDefinitionChange` object is required to add a class to the program (for example to add the `User` class) but also to add or remove attributes (for example to add the `username` attribute in the `User` class). As a result of this term overloading, the change `define User` appears four times in the change list (lines 2, 5, 6 and 24). This hinders the understanding of the changes in the list.

Lack of high-level changes The ChangeList tool does not allow the explicit monitoring of high-level changes which better represent the intention of the developers. In this evolution scenario, the two intentions of the developers (introducing user kinds on lines 18-25 and ensuring user privacy on lines 27-31) are concatenated in the final change list. This may become a problem if

the developers need to change the way in which the two features are merged, for instance, to only enable registered users to benefit from encrypted communication (see Figure 3.5). The developers have to manually recompose their implementations from the change list.

No exploration facilities The three shortcomings described above illustrate how difficult the change management can become. After making several modifications to a program, the change list can contain large amounts of change objects. In such a case, the exploration of the change lists is cumbersome and error-prone.

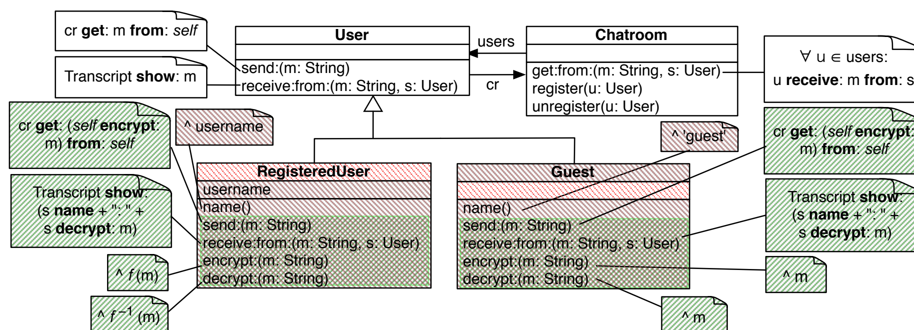

Figure 3.5: New way of merging the changes

In summary, the first-class change objects featured by the Smalltalk Change-List tool provide information about the history of each entity of a program. However, this information has a restricted level of granularity, overloads terminology, lacks high-level changes and does not facilitate exploration. We claim that these inconveniences result from the fact that change is treated as a side-effect from development actions rather than the main development entity. In the next section, we propose to centralise change as the central development entity and call this style of programming, change-oriented programming.

3.2 Change as the central development action

We first introduced *change-oriented programming* (ChOP) in [31]. ChOP is a style of programming (a programming paradigm) which centralises change. In order to create a piece of software in the ChOP style, a programmer does not need to write large streams of source code, but rather uses the interactive development environment (IDE) to *apply changes* to his source code.

Most mainstream IDE's already partially support ChOP. An example of this support can be found in the creation of a new class in Eclipse, where the programmer can create a class by right-clicking a package in order to add a new class

to that package. The IDE then provides the necessary dialogs to interact with the programmer in order to obtain all parameters of the class. Finally, it is the IDE that produces the source code of the newly created class and that inserts this code in the correct locations. Another example is the refactoring[2] [17] support of the VisualWorks IDE. When a programmer decides to *pull up* a method (move it to the superclass), he right-clicks the method and selects the 'pull-up' menu item after which the IDE performs the actual modifications to the source code in order to execute the method pull up.

The idea of ChOP is that the entire software system is developed as illustrated above: by making the IDE execute changes. It is clear that the IDE should support all possible kinds of changes to source code and provide the dialogs to collect the information it needs to perform them. Next to that, the evolution scenario of Section 3.1.1 revealed a need to support user-defined changes. The following section elaborates more on those and other requirements for ChOP.

3.3 Requirements for ChOP

ChOP centralises change in order to overcome the issues presented in the evolution scenario of Section 3.1.1. ChOP has two axes: the change model and the change management. The change model represents the various kinds of changes that can be applied by the programmer. Change management consists of the creation, application and storage of changes. In this section, we subsequently present some desired properties for the change model and management system behind ChOP.

3.3.1 First-class changes

First-class changes are objects that represent change and can be referenced, queried and passed along [33]. They were shown to provide useful information about the evolution history of software programs (see Section 2.3). As we want to pass changes around and reference them from within change lists, they need to be first-class. As an example of a model for first-class changes, we take the model behind the ChangeList tool. We now present four additional properties of first-class changes, which are desirable in order to overcome issues with the ChangeList model (presented in Section 3.1.1).

Fine-grained changes

The different kinds of changes in the ChangeList tool are structured in a hierarchy of change classes. This hierarchy eases extending the set of changes, stimulates reuse and improves maintainability. We support all these principles and therefore propose to preserve the idea of structuring the types of changes in a hierarchy in which subtypes inherit common functionality from their super types.

[2]In software engineering, refactoring source code means improving it without changing its overall results, and is sometimes informally referred to as "cleaning it up"[40].

The first-class changes of the ChangeList tool express changes about packages, classes, attributes or methods. The statements of a method body are not explicitly considered as a subject of change. The decomposition of a method body, however, always reveals more detailed information that can be used to study the evolution of the concerned program. The fact that a method statement includes an invocation of another method, implies that there is a relationship between both methods, and that the former method could be affected when the latter is changed.

Another example of such a restriction of granularity can be found in the modification of attributes. Changes to method parameters are also not explicitly modeled by the the ChangeList tool. Assume a method call that has a complex expression as an argument. This expression can contain method invocations which reveal links between the caller and the callee.

In our model, we propose to introduce dedicated change objects for all possible associations between program entities. An `AddInvocation` change, for instance, describes the addition of an invocation of a specific method and maintains a reference to the method it invokes. Keeping this reference avoids the need to recover it later, which is even not always possible due to programming language features such as polymorphism, dynamic binding or especially behavioral and structural reflective capabilities [66].

Enabling changes at this level of granularity allows the change list to contain information which can be used to recover the invocations of a method.

Composable changes

A programmer usually needs to apply several changes in order to introduce a new functionality into a software system. For example, ensuring privacy for users requires four changes (lines 28-31 in Listing 3.1). Every one of those four changes has the same intent: ensuring user privacy. As such, they could be grouped together based on their common intent. We propose to distinguish between two kinds of changes: atomic and composite changes, as depicted in Figure 3.6.

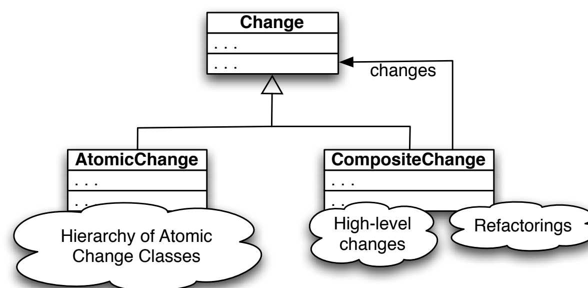

Figure 3.6: Composable first-class changes design

Atomic changes are operations that directly manipulate the abstract syntax tree of a particular program. These operations consist of adding or removing an entity (for instance a class) as well as changing the properties of an entity (for instance the name of a class). Like all changes, atomic change operations are captured in first-class entities that are (re)applicable. By reapplying the changes of the complete change list, a developer can reproduce each development stage a program went through during its evolution.

Composite changes consist of multiple changes which can in turn be composite or atomic changes and which are *all* carried out whenever the composite change is executed. They group some changes in a composition, which can be more expressive than the atomic changes on their own. A `ChangeMethod`, for instance, could be expressed by a composition of a `RemoveMethod` and an `AddMethod` change. In some cases, if one of them is undone, the other should be undone as well. The information that represents this relationship is captured in the `ChangeMethod` change that surrounds both the `RemoveMethod` and an `AddMethod` change.

Composing changes also improves comprehension of the change list. A system history consisting of only atomic operations leads to an enormous and poorly organized amount of information. Therefore all change operations are composable: They can be grouped into higher-level operations with a more abstract meaning.

A final application of composite changes can be found in domain-specific changes. Imagine that we envision the addition of lots of new kinds of users to our Chat application. In that case, it probably would be better to define a dedicated change type `AddUserClass`, which in its turn adds a subclass to the `User` hierarchy and add methods for instanciating that new class. As such change types are targeting a specific a specific problem domain, we call them domain-specific. By means of such domain-specific changes, the level of abstraction of changes is raised, which leads to better readable (and more understandable) change lists.

Dependent changes

In our model, every change has a set of preconditions that should be satisfied before a change is applied. Such preconditions are related to system invariants imposed by the programming language (usually defined by the meta-model of the language). For example, methods can only be added to existing classes. Preconditions enable expressing dependency relationships between changes. In the scenario of Section 3.1.1, for instance, the change that adds the method `name` to the `RegisteredUser` class depends on the change that added the `RegisteredUser` class to the system, as the latter is the creational change for the subject of the former.

There are many kinds of dependencies between changes. In Section 4.4, we classify the dependencies based on their origin. An example is the creational *dependency*.

A change is always applied to a building block, which we refer to as the *subject* of change. *Creational changes* are changes which have as subject a new entity that they produce. A change c_1 is said to *creationally depend* on a change c_2 if that is the creational change of the subject of c_1

```
Change the name of method "name" in class RegisteredUser         1
        to "username"                                            2
Change the name of method "name" in class Guest to "username"    3
Change the invocation "s name"  in method receive:from: from     4
        User to "s username"                                     5
```

Listing 3.2: Change method body: extension

```
Change the name of method name in all subclasses of the class    1
        User to "username"                                       2
Change every invocation of method "name" to an invocation of     3
        method "username"                                        4
```

Listing 3.3: Change method body: intension

Intensional changes

A set of changes can be specified extensionally – by listing them – or intensionally – by adding quantification. Assume a change in which we rename the `name` method of the users of Figure 3.5 to `username`. This evolution step can be implemented by the following two algorithms which are depicted in Listings 3.2 and 3.3.

Both algorithms 3.2 and 3.3 represent the same modification to the system: a “change name refactoring” [40] to the method `name`. There is an important difference between both, however. Combining this change with the addition of another invocation of method `name` would result in an inconsistency when the extensional list is applied – the other invocation of `name` would not be changed. As the intensional change finds *every* invocation of `name` by definition (Listing 3.3), such conflicts are avoided. We name the changes that use an intensional description intensional changes. Note that we need a way for expressing qualifiers like *every* or *exists* for specifying intensional changes. A logic-based declarative language was already used in the passed for such matters [72].

3.3.2 Change management

The second axis of requirements for change-oriented programming lies in the management of the first-class changes. A change first needs to be produced, then maintained and finally used. We identify two ways of producing changes: by instantiating the change kind by means of an interactive dialog provided by the IDE, or by monitoring the programmers actions in the IDE and producing the change objects that correspond to those actions. The maintenance of changes consists of a book keeping of all changes that were produced, in such a way that they can easily be retrieved and queried later on. In order to support ChOP, specific IDE support is necessary which allows developers to develop (and evolve) their software by means of changes. In fact, from this point of view, developing is not different

from evolving software. The following sections subsequently present how the IDE should support the production, the management and the use of changes.

Extensible change hierarchy

An IDE which enables change-oriented programming should incorporate a change taxonomy, which represents all kinds of changes that can be executed by a programmer in order to develop a software system. For every change in the hierarchy, the IDE should provide the adequate dialogs it can use to obtain extra information from the programmer in order to execute the concerned change. The change hierarchy should be extensible, so that a programmer can add self-defined (domain-specific or general composite) kinds of changes.

Verification of pre-conditions

A change has pre-conditions which must hold before it can be executed. Those pre-conditions can be of a semantic or a structural kind and are introduced by the meta-model of the concerned programming language. In a class-based object-oriented programming language, for instance, a method can only be added to a class, if that class already exists. The IDE should enforce that the pre-conditions of the changes are verified before they can be carried out. A good starting point to do that, is to enable only the kinds of changes that can be performed in a certain context of the IDE. For instance, when a class is already added and selected, a programmer can execute a change to add a method to that class.

Composition of changes

As we have explained in the previous section, changes can be composed in order to form new change types. The IDE should include the functionality to support this process. Concretely, the programmer should be able to compose scripts of existing changes and capture them in new change types. User dialogs must be generated that can query the developer for the information that specifies those changes.

Another aspect of the composition of changes is the grouping of change instances based on a common property (the user that produced those changes, their intent (= their reason of existence), the time on which they were applied, etc). Such grouping can be used to classify changes in change groups, which can afterwards be used to reason about the changes on a more abstract level.

Declaration of intensional changes

In the previous section, we explained that some changes might require an intensional description. The IDE should support the programmer in the declaration of intensional changes. For that, two things are needed. First a specific language is needed, which can be used to express the intentional descriptions of the changes. Second, an evaluator is required, which is able to interpret the intensional specifications of changes and which can evaluate them to the extension that corresponds to that intension. Both these matters are addressed in Chapter 7.

Maintenance of changes and references

The maintenance of changes requires three separate tasks. First, the different kinds of changes have to be maintained in order to allow the programmer to instantiate them for developing software in a change-oriented programming style. Second, the change instances have to be maintained in a repository in order to be able to use them afterwards. Finally, change management also consists of maintaining the link between the program building blocks (like classes, methods, etc.) and the changes that created and affect them. This link can be used to reveal the intent of the concerned program building block.

Development support

The idea behind ChOP is that a developer does not write code himself, but rather that the code is generated for him when changes are executed. The IDE should provide support for executing those changes. Concretely, all kinds of change should be invokable from within the IDE. Moreover, the IDE should provide dialogs to interact with the programmer in order to obtain the information required to specify the change. In fact, some IDE's already provide some support to that regard. Eclipse [104] and VisualWorks [50] for instance, both provide an interactive way of adding a new class to a system. Both do this by means of graphical dialogs which request the desired information from the developers. Such interactive support to apply other kinds of changes, like adding methods or more fine-grained changes, however, is not included in those IDE's.

In practice, though, pure ChOP does not seem realistic. We do not believe a programmer will actually execute a dialog for adding a statement to a method body. For that, we propose that the IDE is instrumented with a logging mechanism which is capable of monitoring the actions of a programmer and which instantiates the changes that correspond to those actions. While this does not correspond to pure ChOP, we believe it makes ChOP more applicable and useable, while it does not take away the main benefits of ChOP, which we discuss in the following section.

3.4 Advantages of ChOP

This section presents the advantages of the ChOP paradigm. In this dissertation, we highlight the two main advantages of ChOP and subsequently discuss them below.

3.4.1 Incremental change management

Change management is a technique for storing and managing the changes of evolving software systems. An *incremental* change management – in contrast to a *snapshot-based* change management – consists of continuously storing changes to software systems in a central repository. A *snapshot-based* change management stores snapshots of the software system in a repository. In fact the latter rather stores the results of changes but not the changes themselves. Consequently, the

latter only records the evolutionary information at the *explicit request* of developers [91] while the former *implicitly* records the evolutionary information: changes are recorded as they happen.

Such a change management system is integrated into the *Interactive Development Environment* (IDE) and continuously captures changes as they are applied by the developer(s). Hence developers do not carry the responsibility of committing versions. Each captured change increments the system history stored in the repository enabling to reconstruct a software version at any point of time.

In [91], Robbes et al show that incremental change management provides a lot of useful information about the evolution of the software systems. Hence, such change management is a valuable source for software evolution researchers. Another advantage of incremental change management is that it allows a bottom-up approach for feature-oriented programming (as we will elaborate on in Chapter 5).

3.4.2 Combination with other paradigms

In order to explain the fact that ChOP can be combined with any other programming paradigm, we first explain the difference between a *model* and a *meta-model*. A model is an abstraction of real world phenomena, while a meta-model is another abstraction, highlighting properties of the model itself. A model conforms to its metamodel in the way that a computer program conforms to the programming language in which it is written. Following these definitions, a programming paradigm is specified by a meta-model, which can be used to produce software programs which are models (instances of the meta-model).

In one point of view, ChOP is not different from other programming paradigms (like object-oriented or aspect-oriented programming) as it is also specified by a meta-model. From another point of view, ChOP is different as it is always *applied* to the meta-model of another programming paradigm. ChOP is specified by a meta-model which consists of the *addition*, *modification* and *deletion* of the building blocks of the programming paradigm which it is applied to [3]. In fact, ChOP adds dynamics (changes) to a static view of a programming paradigm and can consequently be seen as an extension of the meta-model of the programming paradigm which it is applied to.

An advantage of ChOP is that a programmer can develop software in a change-oriented way using his favorite programming style and programming language. Indeed, from the moment the IDE supports the above defined criteria for ChOP, the programmer is able to program in a change-oriented way in his favorite programming language. While the concepts of ChOP are applicable to any programming language and programming paradigm, we chose to apply it to the class-based object-oriented programming paradigm to which languages such as Ada, C++, Java and C# adhere. Consequently, in the remainder of this dissertation, ChOP is considered in combination with class-based object-oriented programming.

[3] The building blocks of class-based object-oriented programming, for example, are packages, classes, methods, etc. For aspect-oriented programming, on the contrary, the building blocks also contain aspects, pointcuts, etc.

3.5 Tool support

ChEOPS (the change- and evolution-oriented programming support) is an IDE plugin for VisualWorks, which we created as a proof-of-concept implementation of ChOP. ChEOPS implements a model of changes that is compliant with all requirements described above and does not fall back on ChangeList [107] – a change management tool included in most Smalltalk IDEs. Reasons for this are elaborated on in Section 3.1.

Figure 3.7 presents the architecture of the ChEOPS kernel. It consists of nine parts. The *Change Creator*, the *Change Importer* the *Differentiator* and the *Change Logger* are four modules that can produce first-class change instances. While all of them are capable of producing change instances, they all gather the information concerning the change in a different way. The logger obtains it by monitoring the development actions. The creator requests it from the developer by means of dialogs. The differentiator uses a differentiation strategy between two versions of the source code of a system. The importer is capable of importing change objects that were created elsewhere.

After their creation, the changes are stored in the *Change Repository*: a repository of all first-class change objects. It is the *Change Classifier* that is capable of classifying changes into sets of changes that are related. Classification can be based on logic formula, which are evaluated by the *Logic Kernel*. Finally, changes can be inspected and composed. This is done respectively by means of the *Change Browser* and the *Change Composer*. For more information about the implementation of ChEOPS, we refer the reader to Section 8.1.

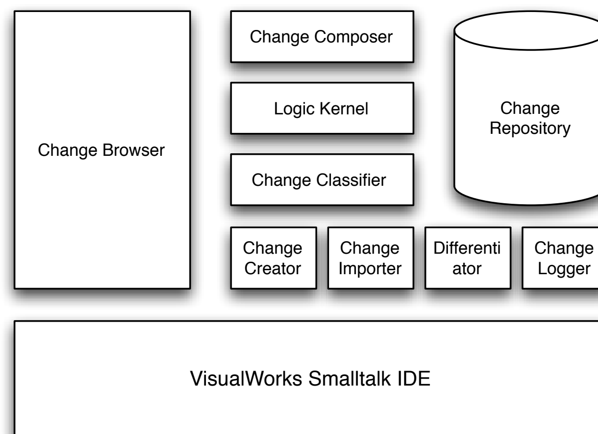

Figure 3.7: ChEOPS Architecture

ChEOPS fully supports change-oriented programming but also has the capability of logging developers producing code in the standard OO way. Therefore,

```
Changes to ChatApp package:                                              1
  Add new class "User"                                                   2
  Add new instance method "receive: m from: s" to class "User"           3
    -> Invocation tree added                                             4
  Add new instance variable "cr" to class "User"                         5
  Add new instance variable "username" to class "User"                   6
  Add new instance method "send: m" to class "User"                      7
    -> Invocation tree added                                             8
  Add new instance method "name" to class "User"                         9
    -> Add Read Access                                                   10
  Add new class "ChatRoom"                                               11
  Add new instance variable "users" to class "ChatRoom"                  12
  Add new instance method "get: m from: s" to class "ChatRoom"           13
    -> Invocation tree added                                             14
  Add new instance method "register: u" to class "ChatRoom"              15
    -> Invocation tree added                                             16
  Add new instance method "unregister: u" to class "ChatRoom"            17
    Add invocation "users remove: u"                                     18
```

Listing 3.4: Chat application: change list by ChEOPS

ChEOPS instruments the IDE with hooks and uses them to produce fine-grained first-class change objects that represent the actions taken by the developer. Those objects can later be grouped into a change set that specifies a transition which might be applied to a base in order to extend the base with the feature expressed by that change set.

The following goes back to the evolution scenario from Section 3.1.1 and shows how the support provided by ChEOPS overcomes the four identified issues. Listing 3.4 shows the list of changes of the basic 2-class chat application, as they are logged by ChEOPS. Listing 3.5 and 3.6 respectively show the changes of adding different user categories and ensuring user privacy in the way that ChEOPS logged them.

Contrary to Listing 3.1, the ChEOPS change list allows a clear separation between an addition of a new class (line 2 of Listing 3.4) and the addition or removal of instance variables to a class (lines 5 and 6 of Listing 3.4). This is a consequence of *not overloading* the `AddClass` change, but separating every change into a different kind of change class. This improves the understandability of the change list.

In the ChEOPS change list, every modification at the method level is represented not only by a statement such as `User send:(change)` (lines 7, 30 of Listing 3.1). Instead, ChEOPS distinguishes between the addition (line 7 of Listing 3.4) and the removal of a method (line 10 of Listing 3.6). Next to that, ChEOPS does not log changes only at the level of methods, but also logs more *fine-grained changes* at the statement level of the method bodies. This overcomes the problem of *the restricted granularity* which was identified in Section 3.1.

```
Changes to ChatApp package:                                              1
  Add new class "Guest"                                                  2
  Add new instance method "name" to class "Guest"                        3
    -> Invocation tree Added                                             4
  Add new class "RegisteredUser"                                         5
  Add new empty instance method "name" to class "RegisteredUser"         6
  Add read access to behavioral entity "name" >> return value            7
      of  variable "username"                                            8
  Remove instance method "name" from class "User"                        9
    -> Invocation tree Removed                                           10
```

Listing 3.5: Adding different users: change list by ChEOPS

```
Changes to ChatApp package:                                              1
  Add new instance method "encrypt: m" to class "User"                   2
    -> Invocation tree Added                                             3
  Add new instance method "decrypt: m" to class "User"                   4
    -> Invocation tree Added                                             5
  Remove instance method "receive: m from: s" from class "User"          6
    -> Invocation tree Removed                                           7
  Add new instance method "receive: m from: s" to class "User"           8
    -> Invocation tree Added                                             9
  Remove instance method "send: m" from class "User"                     10
    -> Invocation tree Removed                                           11
  Add new instance method "send: m" to class "User"                      12
    -> Invocation tree Added                                             13
```

Listing 3.6: Adding user privacy: change list by ChEOPS

When both extensions need to be merged in order to allow only registered users to send and receive encrypted messages, we actually want to obtain a change list similar to Listing 3.7. This shows that just concatenating the changes of both programmers does not do the job. In fact, the four changes of the second programmer, which were originally applied to the single class `User` now need to be applied to both the `RegisteredUser` class and the `Guest` class. As such, we want to group these changes by their intention and parameterize them with the class which they need to be applied to. Additionally, the two classes differ in what kind of encryption and decryption functions are required. Consequently, these functions have to be expressed as additional parameters to this composition of changes. The change list of Listing 3.8 shows the same ChEOPS change list as Listing 3.7, but in stead of listing all atomic composite changes, it contains two high-level composite changes of the type `AddEncryptionChange`. Such a change is a composite change that adds encryption to a class.

This does not only solve the merging problem of the evolution scenario, but also brings along improved support for reusing changes. This high-level change

```
Changes to ChatApp package:                                              1
  Add new instance method "encrypt: m" to class "RegisteredUser"         2
    -> Invocation tree Added                                             3
  Add new instance method "decrypt: m" to class "RegisteredUser"         4
    -> Invocation tree Added                                             5
  Remove instance method "send: m" from class "RegisteredUser"           6
    -> Invocation tree Removed                                           7
  Add new instance method "send: m" to class "RegisteredUser"            8
    -> Invocation tree Added                                             9
  Add new instance method "encrypt: m" to class "Guest"                  10
    -> Invocation tree Added                                             11
  Add new instance method "decrypt: m" to class "Guest"                  12
    -> Invocation tree Added                                             13
   Add new instance method "send: m" to class "Guest"                    14
    -> Invocation tree Added                                             15
   Add new instance method "receive: m" to class "Guest"                 16
    -> Invocation tree Added                                             17
```

Listing 3.7: Adding user privacy correctly: change list by ChEOPS

```
Changes to ChatApp package:                                              1
  Add encryption for class "RegisteredUser" with function                2
    "f(x)= encrypt(x)" and "f(x)= decrypt(x)"                            3
     -> composite changes                                                4
  Add encryption for class "Guest" with function "f(x)=x" and            5
    "f(x)=x"                                                             6
    -> composite changes                                                 7
```

Listing 3.8: Adding user privacy correctly: (compositional) change list by ChEOPS

can now be applied on all kinds of classes that understand the `name` message and have access to an instance variable `cr` which behaves like a `Chatroom`. Next to that, it also improves the readability of the change list, as the extensive list of composite changes is abstracted away behind the high-level change.

Exploring the change list of ChEOPS is easier and more user-friendly than in the traditional ChangeList tool. This is a consequence of exploiting the *improved exploration support* brought by the first-class change model behind ChEOPS. By means of the ChEOPS change browser, changes can be looked at from four perspectives: ordered on time, grouped by affected entity, grouped by composition and grouped by intension.

- The first view (depicted in Figure 3.8) is similar to the traditional ChangeList approach, with the difference that ChEOPS organises the changes in a tree structure where the fine-grained changes beyond the method-level are hidden in the branches of the method-changes.

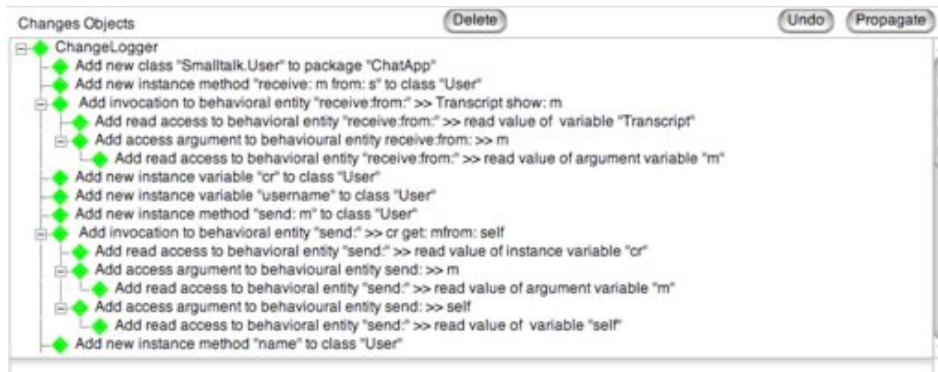

Figure 3.8: ChEOPS view on change list (ordered by time)

- Figure 3.9 shows how the second view groups the changes by the entity they affect. This entity can be any of the building blocks of the chosen meta-model For ChEOPS supports ChOP for class-based object-oriented programming it supports changes on building blocks such as classes, methods, attributes, etc. This view also contains a tree of changes, where the dependent changes are hidden in the branches of the creational change of their subject. For example, the dependent changes of the creational change of `User`, are structured in a branch below that change.

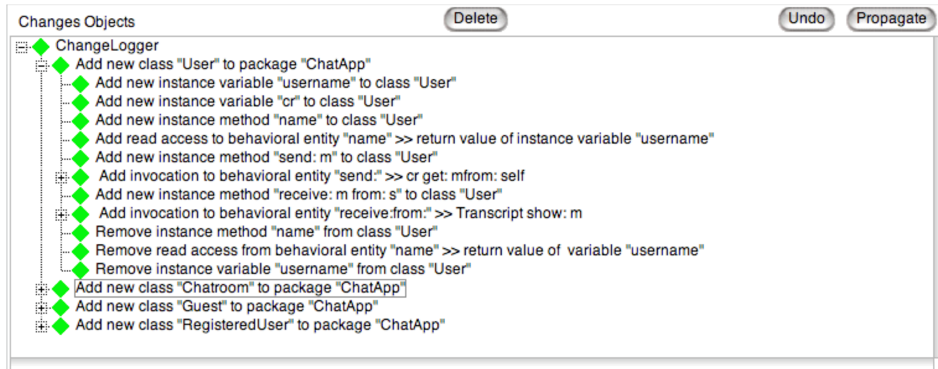

Figure 3.9: ChEOPS view on change list (ordered by affected entity)

- A third view which is included in ChEOPS groups the changes by their composition. Figure 3.10 shows the composite `AddAncryptionChange`, and

the atomic changes it contains. This view dramatically decreases the number of changes to be displayed.

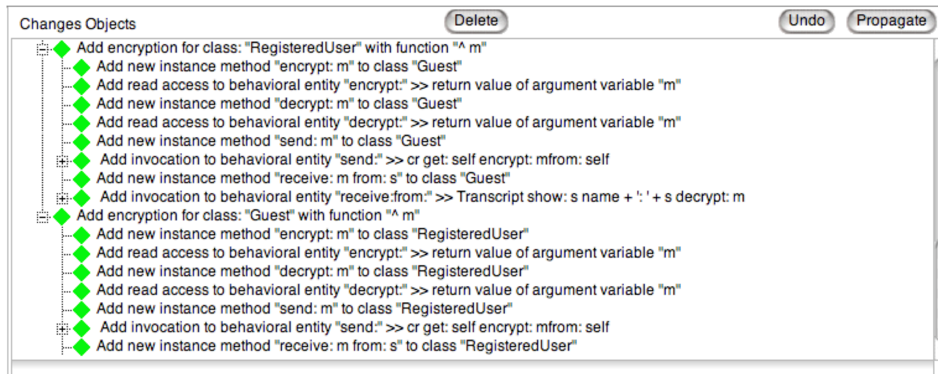

Figure 3.10: ChEOPS view on change list (ordered by composition)

- Yet another ChEOPS view can present the changes grouped by their common intent. An intent of a change is the reason of existence of that change; the reason why that change was instantiated (a bug fix, the introduction of a functionality, a refactoring, etc). The nodes of this tree view contain all the intentions that are detected amongst the complete change set. Those nodes can be expanded to present all the change objects in the same way as the second ChEOPS view.

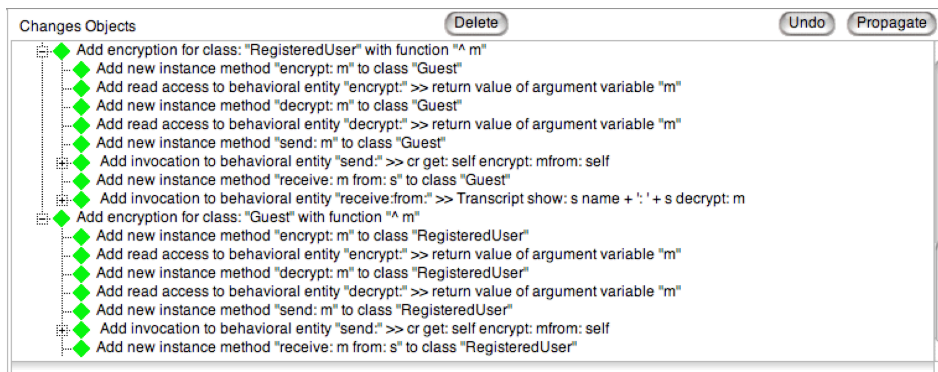

Figure 3.11: ChEOPS view on change list (ordered by intention)

Figures 3.8 to 3.11 present the different views on changes that ChEOPS can produce. In ChEOPS, those views are actually part of the change browser, that

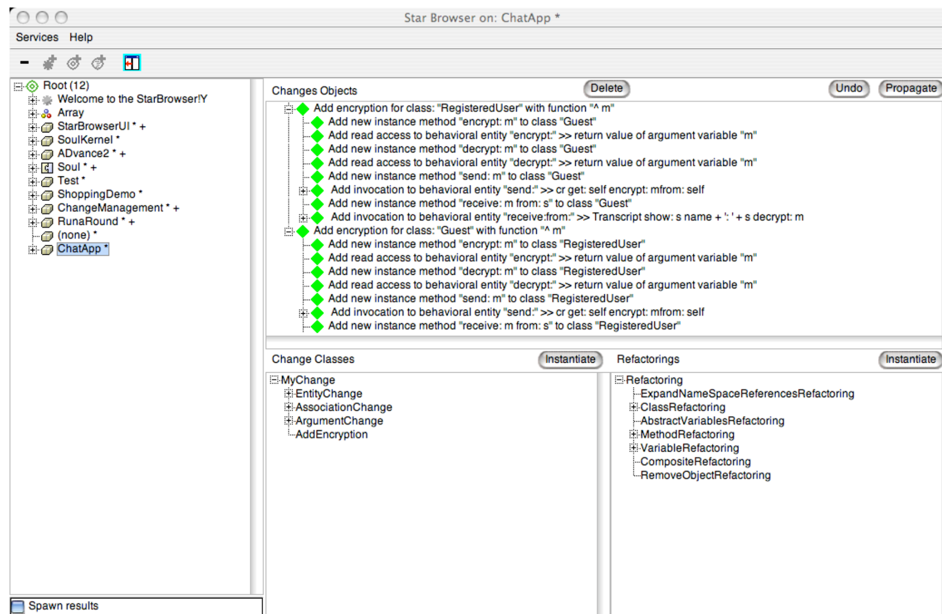

Figure 3.12: ChEOPS change browser

we implemented in ChEOPS. Figure 3.12 depicts a complete view of the change browser. The left pane contains the Smalltalk packages on which the developer is working. In the upper right pane, we find the views we just elaborated on. In the bottom right pane, a hierarchy is presented with all the change kinds ChEOPS supports: Composite changes are found on the right, while atomic changes are found on the left. The developer can execute those changes by expanding the tree and selecting the type needed. ChEOPS then presents a dialog that interacts with the programmer in order to obtain all the required information for instantiating that change.

3.6 Discussion

Table 3.1 shows an overview of the four problems we identified in Section 3.1. The vertical axis shows the requirements that we propose for ChOP (Section 3.3). Cells containing an `x` denote that the property of the cell's row helps to overcome the problem of the cell's column. The rest of this section discusses how this is achieved:

Restricted level of granularity The restricted level of granularity is noticeable by the lack of both very fine-grained and very coarse-grained changes. Our

Property	Restricted granularity	Term overloading	Lack of high-level changes	No exploration support
Fine-grained changes	x	x		
Composable changes	x		x	x
Dependent changes				x
Intensional changes			x	x
Extensible change hierarchy	x	x	x	
Verication of pre-conditions				x
Maintenance of changes and references				x
Development support				x

Table 3.1: Problems handled by properties of change-oriented programming

model includes not only coarse-grained changes (such as the addition of classes or methods), but also more *fine-grained changes* such as method invocations and accessors. The possibility to compose changes into *composite changes* allows the definition of more coarse-grained changes, which can be used to abstract away from the fine-grained level. These extensions provide the granularity required to reason about and understand the evolution history of programs. These extensions are supported by the IDE in the form of an *extensible change hierarchy.* As programmers can define their own kinds of changes, they can broaden or tighten the granularity spectrum.

Term overloading Our model provides a different change for every building block of the chosen meta-model. As such, every change to such a building block is captured by a specific change. This avoids overloading change classes to capture different kinds of changes. We can conclude that in our model, the definition of a change is unique. The fact that the programmer can define his own change types and incorporate them into the *extensible change hierarchy*, also assists in avoiding term overloading.

Lack of high-level changes Our model enables defining high-level changes that better represent the developers' intentions. The model is extensible so that developers can define their own domain-specific changes. This is achieved (1) by the use of an *extensible change hierarchy* in which all the change types reside

and (2) by the composite design pattern (distinguishing between atomic changes and *composite changes*). High-level changes can also be defined as *intensional changes*, which describe a pattern of change. The application of high-level changes is conditioned by the fulfillment of their preconditions. This can be supported by the *IDE*, which guides the programmers in defining new kinds of changes, applying their changes, undoing changes and verifying the preconditions to ensure the application consistency.

No exploration facilities All the notions of first-class change described in this work are implemented in ChEOPS, which supports program exploration by providing different views on the changes. The views structure the changes based on the *dependencies* between them, on their *composition*, on their *intension* or on the time on which the changes were made. Different views on changes can be used for different goals. While the dependency view seems interesting as a means to undo and redo changes, the intentional view seems more interesting for understanding the changes. Exploration of changes in these views is only possible if the *IDE supports the development* in a change-oriented way and if the *changes*, their *references* and their *pre-conditions* are maintained by the IDE

Other researchers pointed out the use of encapsulating change as first-class entities as well. In [91], Robbes shows that the information from the change objects provides a lot more *information about the evolution* of a software system than the central code repositories. In [28] Denker shows that first-class changes can be used to define a scope for dynamic execution and that they can consequently be used to *adapt running software systems*. We refer to Section 2.3 for a more complete overview of related approaches.

3.7 Conclusions

In this chapter, we presented change-oriented programming (ChOP): a programming paradigm that targets software evolution and that centralises changes as the main entity in the development process. The subject of the change refers to the building block of the programming language in which the program is being developed. As such, ChOP builds on top of another programming paradigm in which those building blocks are written. In this dissertation, we build on class-based object-oriented programming and take the chosen meta-model as a model for describing the programs that are to be changed.

By means of an evolution scenario, we show the need for the centralisation of change. We then introduce ChOP and explain that it has two dimensions: the *model of first-class changes* and the *management of the change instances*. We first show why first-class changes are desired to program in a change-oriented way. We identify four issues with respect to the model of first-class changes, as it is presented in a related approach. The restricted level of granularity in the different types of changes, the overloading of change types, the lack of high-level changes and the lack of program exploration facilities hinder good software evolution support. This

explains the need to extend the model of first-class changes in such a way that these problems are overcome. With respect to the management of changes, we present six requirements for a change management system that can enable ChOP.

Advantages of ChOP include the fact that it enables an incremental change management, which has been proven to be beneficial over a snapshot-based change management system with respect to the amount of evolutionary information it contains. In the context of this dissertation, which also includes software variation, ChOP enables a bottom-up approach to feature-oriented programming (presented in Chapter 5). Another advantage of ChOP is that it is actually applied to another programming paradigm and that it consequently must – but also can – be used in combination with any other programming paradigm and programming language.

We briefly present ChEOPS, a proof-of-concept implementation of ChOP which is implemented as a plugin of the VisualWorks for Smalltalk IDE. It is based on the existing implementation of first-class changes, but extended with solutions for the requirements that were identified. The implementation of the evolution scenario in ChEOPS shows that an implementation of ChOP – that takes into account all requirements identified above – indeed overcomes the four problems we identified.

Chapter 4

Model of first-class change objects

In the context of software evolution, many researchers have provided their view on how evolution could be interpreted. Mittermeir for instance, believes that *software evolution* should only be considered as the changes that were applied to an existing software system [76]. In [64], Lehman studies different interpretations of software evolution and elaborates on the theory and practice behind them. He defines software evolution as

> "the dynamic behaviour of programming systems as they are maintained and enhanced over their life time." [63, 93]

In this chapter, we aim to establish and describe an *evolution model*: a model that is capable of representing the evolution of software systems. We show that an evolution model depends on a *meta-model*: a model that describes the building blocks of the software systems *model*. As an example, consider the software system of Section 3.1.1. The model of that software system consists of *users* and *chatrooms*. The corresponding meta-model is composed of *classes*, *methods*, *instance variables*, etc.

We focus on an evolution model which is capable of capturing the evolution of object-oriented class-based systems in first-class change objects. The term first-class refers to the property that changes are entities which can be queried, adapted and passed along. The resulting evolution model can then be used to express the evolution of any software system developed in a programming language that adheres to that model.

As a meta-object model for software systems, we consider the Unified Modeling Language (UML) meta-model [78, 79], the Common Object Requesting Broker Architecture (CORBA) [47] and the FAMOOS Information Exchange model (FAMIX) [27]. All were created to support the information exchange about software systems that are implemented in different programming languages. They all provide a meta-model for class-based object-oriented programming which can be

used to write a system's blueprints. UML or CORBA, however, do not provide ways to specify intra-method dependencies (such as method invocations or variable accesses). Since that information is required to express the evolution of software systems, we choose the FAMIX model.

FAMIX is a meta-model which Java, Ada, C++ and Smalltalk adhere to. This means that the FAMIX model can be used to express the models of software programs written in one of those languages. FAMIX, however, also provides extension hooks that can be used to cover other programming languages. In the following section, we discuss the FAMIX model and expose its extension hooks. In Section 4.2, we discuss an extension to the FAMIX model that captures code statements. In Section 4.3 we elaborate on another extension to FAMIX, which consists of the addition of entities that model the *changes* of the building blocks specified by FAMIX. As such, we provide the necessary facilities to model the evolution of software systems that are developed in a programming language adhering to the FAMIX meta-model.

4.1 The FAMIX model

FAMIX stands for *FAMOOS Information Exchange Model* and was created to support information exchange between interacting software analysis tools. FAMIX captures the common features of different class-based object-oriented programming languages and models the ones that are needed for software re-engineering activities [27, 30, 105].

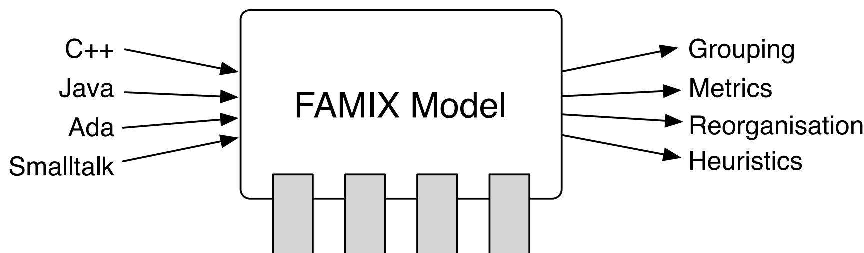

Figure 4.1: Conception of the FAMIX model (based on [27])

Figure 4.1 shows a conceptual view of the FAMIX model. On the left hand side we find different programming languages that were used to implement several case studies of the FAMOOS project. The right hand side of the figure lists various experiments conducted by several software analysis tools on the provided case studies. In the middle, there is the information exchange model that only captures the common features of class-based object-oriented programming languages such as classes, methods or the inheritance relation. To cope with language specific features, the FAMIX meta-model can be extended by using the provided hooks.

Those are represented by the grey bars at the bottom of the figure. The extended meta-model takes as input the source code of the different case studies which in its turn is provided as input to several software analysis tools.

- *Support for multiple languages*: the FAMIX meta-model is designed to model software systems at the level of source code independent of the implementation language. To achieve the support for multiple languages, the FAMIX model only captures common features of many different class-based object-oriented programming languages and omits language specific features. For instance, FAMIX models inheritance in a very generic way, so that both multiple inheritance and single inheritance can be expressed. This is necessary, since both kinds of inheritance are asserted by the programming languages targeted by FAMIX (e.g. multiple inheritance of C++, single inheritance of Smalltalk).

- *Extensibility*: the FAMIX model provides three extensibility mechanisms. First, users have the possibility to define *new concepts* (new meta-model elements). Second, users may add *new attributes* to existing concepts in order to store additional information. Third, just like in UML, FAMIX allows the user to *annotate* any model element by attaching extra information.

- *System invariants*: The FAMIX specification can be used to derive system invariants: clauses that must hold for an instantiation of the FAMIX meta-model, for that instantiation to be valid. A method, for instance, must always belong to a class.

- *Information exchange*: the FAMIX meta-model was created to *support information exchange between tools.*

We now discuss the complete FAMIX meta-model, subsequently its basic data types, classes, attributes and relationships.

4.1.1 Basic data types

The data types of FAMIX for variables and return types can be split up in two categories: primitive and non-primitive data-types.

- *Primitive data types*: There exist two primitive data types in FAMIX (`String` and `Integer`).

- *Non-primitive data types*: FAMIX defines three extra data types that may be used in the model:

  - `Name` is a `String` that gets its semantics from the model of the developed software system. A $uniqueName$, for instance, may be used to refer to an object from within the model.

- `Qualifier` is a `String` that also has semantics outside the model of the developed software system. The *sourceAnchor* of an object, for example, contains the location of the source code of that object. This location can be used by any software application to retrieve the source code of the object.
- `Index` is an `Integer` that represents the position in some sequence. As a *parameter* is used in a sequence, it has a position in that list. This position is of type index.

The naming conventions used in FAMIX are very compliant with UML [15, 44]. For more information about the FAMIX naming conventions, we refer the reader to [27]. We subsequently discuss all the building blocks of FAMIX and illustrate their structure by means of UML class diagrams. We see that all the building blocks are classified in an inheritance hierarchy with a root called `Object`.

4.1.2 Object

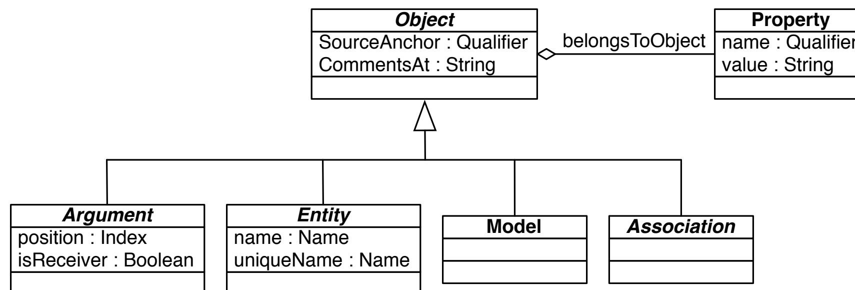

Figure 4.2: FAMIX model - Object (based on [27])

The `Object` class is the root class of the FAMIX model. As shown in Figure 4.2, `Object` is an abstract class [1] that does not inherit from any superclass but which acts itself as a superclass for all other classes presented in the model (except for the `Property` class). An `Object` holds a `sourceAnchor` determining the location of the source code of the concerned object. A typical example of a source anchor consists of the filename and start and stop indices where a class is physically stored. Developers have the possibility of commenting model elements, these comments are stored in the `commentsAt` attribute.

The `Property` of an object holds annotations that can be made to the FAMIX objects. As such, properties provide an extension hook that can be used to extend the FAMIX model. A `Model` represents information concerning the particular software system being modeled (e.g. name of the publisher or the used programming

[1] An abstract class is a class that cannot be instantiated. It is typically used to group the shared behaviour of all its subclasses.

language). In the FAMOOS project, the models were used by software analysis tools when investigating the provided case studies. The remaining three subclasses (`Entity`, `Association` and `Argument`) are explored in the following three subsections.

4.1.3 Entity

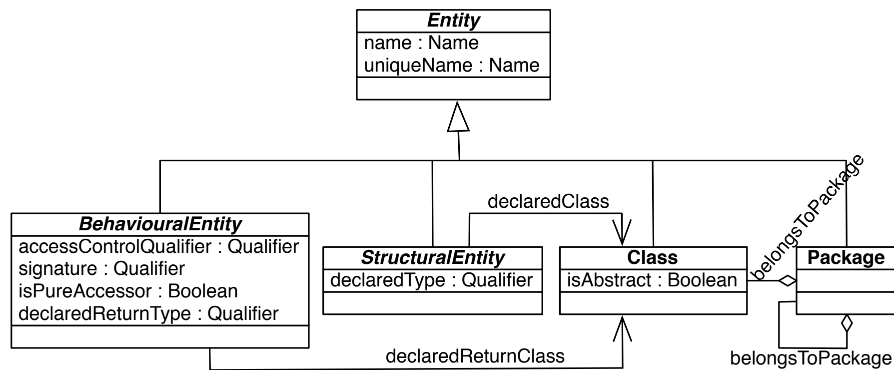

Figure 4.3: FAMIX model - Entity (based on [27])

Figure 4.3 shows the `Entity` class which represents the different mechanisms that can be used in an object-oriented programming language to manipulate the static structure, behaviour and state of the implemented system. As shown in Figure 4.2, `Entity` is an abstract class inheriting from the `Object` class. An `Entity` stores a `name` and a `uniqueName`, which must be unique for all entities in the model of the software system. This is usually ensured by development tools such as an IDE or a compiler.

The `Class` and `Package` classes inherit from the `Entity` class and form the static structure of a software system. `Class` and `Package` respectively model a class and a package in the context of object-oriented programming. A class defines the structure and behavior of all the instances of that class. A `Package` is a container that can be used to group source code. As such, packages organise the software system in smaller subsystems. It is the `belongsToPackage` relationship that denotes this decomposition. The following two subsections elaborate on the `BahaviouralEntity` and `StructuralEntity` classes.

Behavioural entity

Figure 4.4 shows the `BehaviouralEntity` class which represents the definition of a behavioural abstraction in a software system. When invoked, a behavioural abstraction executes one or more actions defined in its body (e.g. calling other

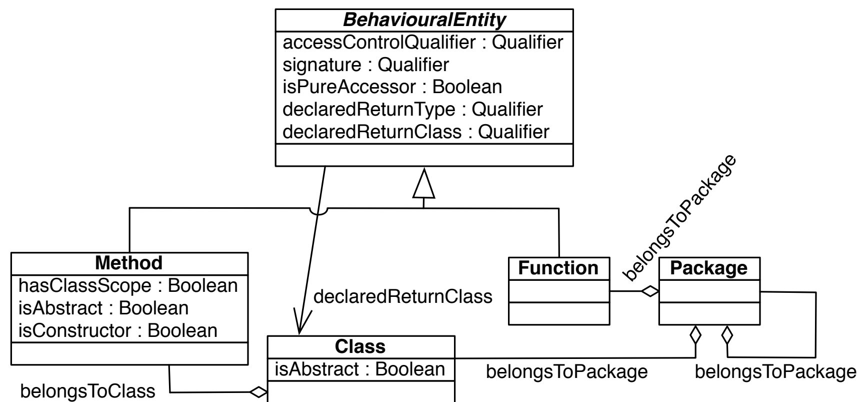

Figure 4.4: FAMIX model - BehaviouralEntity (based on [27])

behavioural entities or instantiating classes). Typical examples of behavioural abstractions are methods and functions. As shown in Figure 4.3 `BehaviouralEntity` is an abstract class inheriting from the `Entity` class.

Every behavioural entity has a `signature` which consists of a name, the number and possibly the type of its parameters. While the signature has to be unique within a class, it does not need to be unique within the complete model because many object-oriented programming languages allow for the definition of different behavioural entities in different classes for the same signature (*method overriding*). A method lookup mechanism determines at runtime which of the overridden methods has to be invoked.

By setting an access control qualifier (`accessControlQualifier`), one can define who is allowed to invoke the behavioural entity. In case the behavioural entity returns an object, the information concerning the returned object is maintained by the `declaredReturnType` and the `declaredReturnClass`. While the former contains the type of the returned object[2], the latter contains a reference to the class implicitly held in `declaredReturnType`. Note that in some programming languages, the return type of a behavioural entity does not need to be statically declared and that in some programming languages, some behavioural entities do not return anything. In such situation, both the `declaredReturnType` and the `declaredReturnClass` are left empty.

Each subclass of the `BehaviouralEntity` class represents a mechanism for defining a specific behavioural entity. FAMIX provides two such mechanisms:

- `Method`: a class used to represent the definition of some behaviour specified within a certain `Class` (as expressed by the `belongsToClass` relationship). The `hasClassScope` attribute denotes wether the method is a class method

[2]The return type is typically a class, a reference to an object (*pointer*) or a primitive type.

(static methods in Java) or an instance method. The `isAbstract` attribute is true for abstract methods (= methods without a body).

- `Function`: A `Function` represents a closure (= a piece of behaviour that is evaluated in an environment containing one or more bound variables). When called, the function can access these variables by using its scope. The scope is an enclosing context in which values and expressions are associated. The scope of a function can be global or local to the place where the function is defined. If provided, the value of the *belongsToPackage* attribute denotes the local scope of the function. Smalltalk blocks are well-known examples of a `Function`.

Structural entity

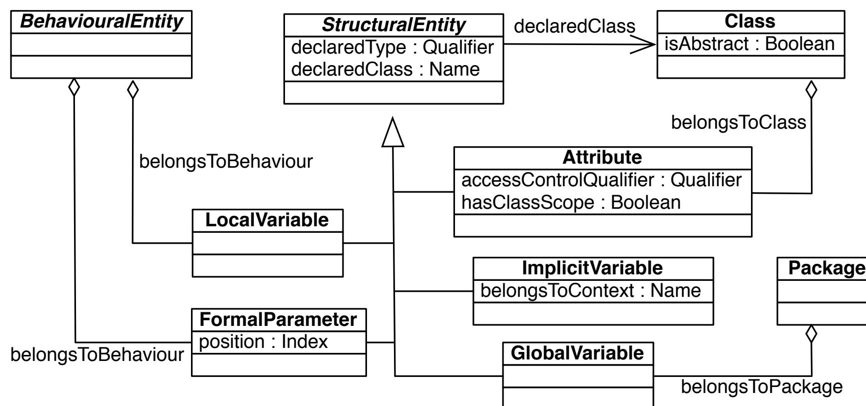

Figure 4.5: FAMIX model - StructuralEntity (based on [27])

Figure 4.5 shows the `StructuralEntity` and the relations it has to other FAMIX entities. A `StructuralEntity` class represents the definition of an entity that concerns the structure of a system (e.g. an attribute specified in a certain class to which a particular value can be assigned). Structural entities are characterized by their scope: Some are visible only within the entity in which they are defined (e.g. attributes are only accessible from within the class). Others have a scope equal to that of the entire running system (e.g. global variables).

Every structural entity is of a certain kind, which is denoted by its type. The `declaredType` Is a qualifier that refers to the declared type of the structural entity. Typically this will be a class, a pointer or a primitive type (e.g., int in Java). `declaredType` is empty if the static type is not known or the empty string (i.e., “”) if the `StructuralEntity` does not have a static type. Note that this is the case in programming languages which are not statically typed (such as Smalltalk).

The declared class is stored in the `declaredClass` attribute, which indicates the unique name of the class that is implicit in the `declaredType`, with the goal of capturing the dependency to the corresponding `Class` instance in a model. The `declaredClass` contains the name of a class, or null if it is unknown if there is an implicit class in the `declaredType`, and the empty string(i.e., “”) if it is known that there is no implicit class in the `declaredType`. The `declaredClass` attribute holds a pointer to the class implicitly held in `declaredType`. What exactly is the relationship between `declaredClass` and `declaredType` is a language-dependent issue. Each subclass of the `StructuralEntity` class represents a possible variable definition, Figure 4.5 reveals five such mechanisms:

- `Attribute`: a variable declared within a class. The `hasClassScope` attribute indicates whether the attribute is defined at instance-level or at class-level. Instances of a class have their own copy of instance-level attributes so that they can maintain a separate state. The scope of a class-level attribute is bound to the class in which it is specified. That class is represented by the `belongsToClass` relationship.

- `GlobalVariable`: a variable with a scope equal to that of the entire running system. The `belongsToPackage` association determines which package the global variable is defined in. Consequently, it can be used to limit the scope of a variable to that package. An empty `belongsToPackage` means that the variable has a global and public scope.

- `ImplicitVariable`: a context-dependent reference to an entity which can only be bound at runtime (e.g. `this` in C++/Java or `self` in Smalltalk). The result of that reference depends on the behavioural entity in which it is invoked. The scope of the variable is denoted by the `belongsToContext` attribute. An empty `belongsToPackage` means that the variable has a global scope.

- `LocalVariable`: a variable defined locally within a behavioural entity, represented by the `belongsToBehaviour` attribute. Another widespread term for referring to a local variable, is the term *temporary variable*. A *closure* can be used to associate a behavioural entity with a set of local variables, which persist over several invocations of that behavioural entity. The scope of a local variable is bound to the behavioural entity in which it is defined.

- `FormalParameter`: is the declaration of the parameter(s) expected by a behavioural entity. The `belongsToBehaviour` attribute keeps a reference to the behavioural entity the parameter belongs to. What exactly constitutes such a definition is a language-dependent issue. In the absence of static typing for instance, it does not make sense to declare the formal parameters. The behavioural entity stores the formal parameters that belong to its signature. The formal parameters maintain their position in the behavioural entity’s parameter list. Formal parameters have a scope equal to the scope of the method or function in which they are defined.

4.1.4 Association

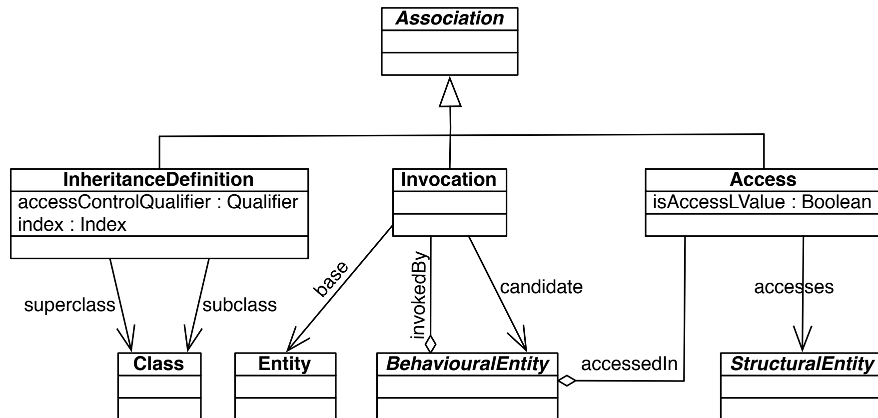

Figure 4.6: FAMIX model - Association (based on [27])

An `Association` defines a relationship involving two entities. As expressed by Figure 4.2, the `Association` class is an abstract class inheriting from the `Object` class. Figure 4.6 shows the `Association` class and its subclasses. Each subclass denotes a different type of relationship, FAMIX provides three such types:

- `InheritanceDefinition` is a representation of the inheritance relation between two classes: a `subclass` which inherits from a `superclass`. The `InheritanceDefinition` class keeps a reference to both subclass and superclass. By setting `accessControlQualifier`, one can define how subclasses access their superclasses. In order to support multiple inheritance, one subclass is able to have more than one superclass. For that, a subclass might maintain a list that contains its superclasses. The `index` attribute of `InheritanceDefinition` refers to the position of the superclass in such a list.

- `Invocation` denotes the invocation of a behavioural entity by another entity (the `invokedBy` relation). An `Invocation` maintains a list of behavioural entities that are possibly called (`candidate`). Static typing information can be used to reduce the size of the candidates at compile time. It is the method lookup mechanism, however, which reduces these candidates to one actual behavioural entity at runtime. The `base` relationship refers to the entity that defines the invoked behavioural entity.

- `Access` is used to represent the access of a structural entity. It is always a behavioural entity (the `accessedIn` attribute) that accesses a structural entity (the `accesses` attribute). The `isAccessLValue` attribute indicates

whether the concerned access corresponds to a getter action (which returns the value of a certain structural entity) or a setter action (which assigns a value to a certain structural entity). When `true`, it denotes a setter action.

4.1.5 Argument

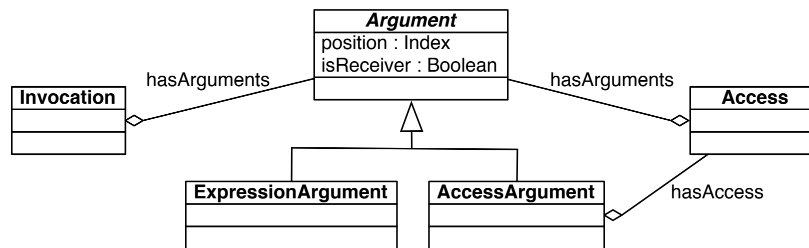

Figure 4.7: FAMIX model - Argument (based on [27])

An `argument` expresses the entity that is passed when invoking a behavioural entity (e.g. function or method) or when accessing a structural entity. As shown in Figure 4.2, the `Argument` class is an abstract class inheriting from the `Object` class. Figure 4.7 shows the `Argument` class and its subclasses. Every `Argument` holds its `position` in the argument list which it belongs to. This list is maintained by the invocation or access by its `hasArgument` attribute. The `isReceiver` is a a predicate that tells whether this argument plays the role of the receiver in the enclosing invocation.

An `ExpressionArgument` models an argument that is an expression. In the main-stream object-oriented programming languages, every argument is an expression: a sequence of variables and constants (the receivers) separated by the messages that are sent to them. How these are structured is a language-dependent issue. In FAMIX, the expressions themselves are not modeled. The invocations and accesses the expression consists of, however, are modeled in FAMIX. An `AccessArgument` models an argument that is a reference to a structural entity. This is only used by certain programming languages (such as C++) to support call by reference. The `hasAccess` attribute denotes the `Access` instance that models the access of a `StructuralEntity` made by the argument.

4.2 Code statements in FAMIX

In Section 3.3, we have seen that it is desirable to express changes at a very fine-grained level of detail. In order to support the modeling of such changes the meta-model has to model entities at the same level of detail. The current version of FAMIX, however, does not model code statements, which we need if we want to express changes at the level of code statements. In this section, we extend FAMIX

with a new entity called `Statement`, which models a code statement. A code statement is a sequence of expressions and keywords (such as `return`). We model the contents of a code statement as a String. In order to make this extension to FAMIX, we use the extension hooks that are provided by its inheritance hierarchy.

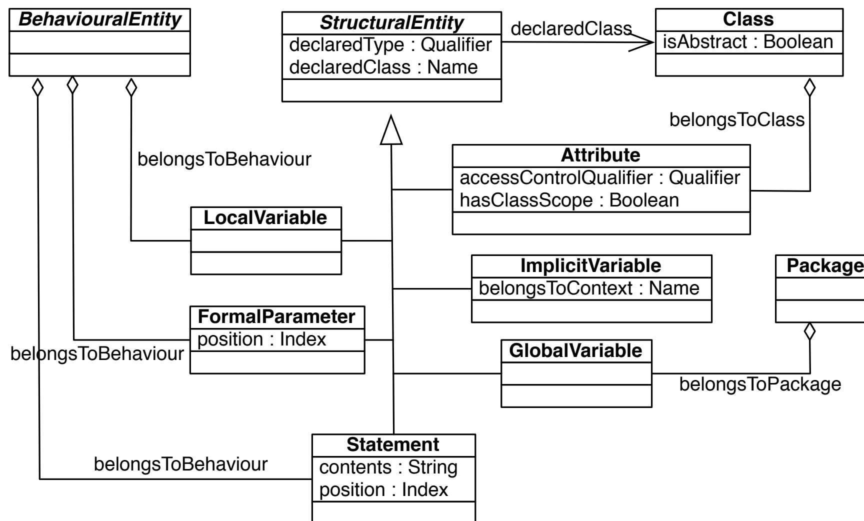

Figure 4.8: FAMIX model - Statement

Figure 4.8, shows how we subclass the FAMIX `StructuralEntity` class in order to introduce a dedicated entity modelling a code statement. Every instance of the `Statement` class is a code statement of which the source code is maintained in its `contents` attribute. Every statement is included in a behavioural entity (hence the `belongsToBehaviour` relation to `BehaviouralEntity`). In fact, every behavioural entity is an ordered collection of statements. The `position` attribute maintains at what index the statement is located in that collection.

4.3 A model of changes

Now we have established a fine-grained meta-model that can be used to model class-based object-oriented software systems, we establish an *evolution model*: a model that can represent the evolution of software systems that are implemented in a programming language that adheres to that model. Our theory starts from the premise that every evolution of a software system can be expressed as a sequence of changes to the building blocks of that software system. Other research, such as [91], confirms that this premise holds.

An evolution model consists of the building blocks that are subject of the evolution and the actions that can be taken to evolve those building blocks. As

the building blocks are modeled by the meta-model (like the one described in the previous two sections), that meta-model is part of the evolution model. The central entity of the evolution model is `Change`: A class that represents an evolution action. The FAMIX building blocks (all inheriting from the `FamixObject`) form the `Subject` of the change. We identify three kinds of evolution actions that can be applied to those subjects: addition, removal and modification. We model those actions with the classes `Add`, `Remove` and `Modify` respectively. Together, they form the concrete commands of the *Command* design pattern [42]. The `Atomic Change` class plays the role of the `abstract command` class in the *Command* design pattern. Next to that, it also fullfils the responsibilities of the `leaf` participant in the *Composite* design pattern [42]. A `Composite Change` is composed of `Changes` (which can in turn be of any change kind). Note that the inclusion of those design patterns in the evolution model provides an extension hook that can be used to introduce new kinds of evolution actions. We elaborate on the difference between atomic and composite changes in subsection 4.3.2.

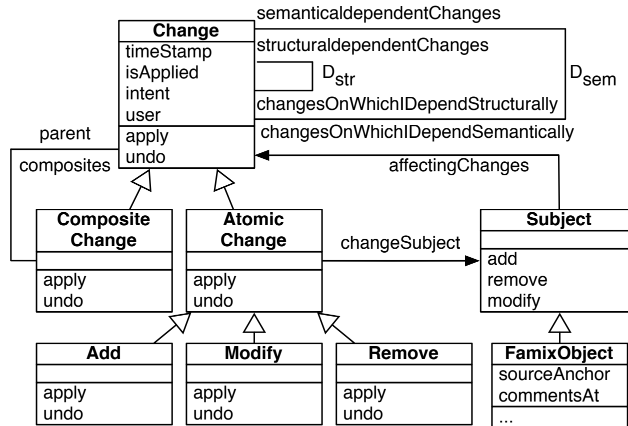

Figure 4.9: Change Model Core

The UML class diagram of the core of the evolution model is presented in Figure 4.9. The diagram also shows two dependency relationships between the change objects: D_{str} and D_{sem}. The details of the dependencies amongst the changes are discussed in Section 4.4. Notice also that a subject maintains a reference to all of the changes that affect it (expressed by the `affectingChanges` relation). This information is redundant but allows for optimisation.

Figure 4.9 shows that we distinguish between two kinds of changes: atomic and composite changes. *Atomic changes* are operations that manipulate the building blocks of a particular program. As was stipulated in the previous section, these operations consist of adding or removing an entity (for instance a class) as well as modifying the properties of an entity (for instance the name of a class). By calling the `apply` method to an atomic change, the change will be applied. By iterating

over the changes and re-applying them, one can reproduce each development stage that a program went through during its development or evolution.

Composite changes consist of an ordered sequence of changes (which can in turn be composite or atomic changes). They group changes in a composition, which can be more powerful than the atomic changes on their own. The `RenameMethod` refactoring, for instance, can be expressed as a composition of a `Modify` change of a method `m` and a `Modify-Invocation` change for every invocation of the original method `m`.

The following two subsections discuss the atomic (subsection 4.3.1) and composite changes (subsection 4.3.2) respectively. Subsection 4.4 explains the dependency relations between change objects.

4.3.1 Atomic changes

In order to explain the atomic changes, we first introduce the different kinds of relations between atomic changes and subjects (building blocks of the software meta-model). From the point of view of the building blocks of the software system, we can say that every building block is created by exactly one change, which we call the *creational* change (C) of that subject. After its creation, every subject can be modified by a variable number of changes. The changes that modify an existing subject are called the *adaptive* changes (A) of that subject. Finally, a subject can be destroyed by exactly one change: the *destructive* change (D) of the subject.

Every subject in the change model maintains an `affectingChanges` attribute: an ordered collection of changes that affect it. Such a list consequently starts with the subject's creational change, possibly followed by a number of adaptive changes, possibly concluded by one destructive change. Note that if there is a destructive change in the ordered collection, it is always the last change. The re-creation of a subject, leads to a new subject with its own `affectingChanges` list.

We identify two kinds of atomic changes: changes to the entities (entity changes) and changes to the relation between those entities (association changes). We subsequently explain both.

Entity changes

From the point of view of changes we identify four different kinds of relations: Changes can be *creational*, *adaptive*, *destructive* or *invariant* with respect to a certain subject. A change `c` is said to be *creational* w.r.t. a building block `b` if the application of `c` adds `b` to the software system. A change `c` is said to be *adaptive* w.r.t. a building block `b` if the application of `c` adapts one of the attributes of `b` in the software system. A change `c` is said to be destructive w.r.t. a building block `b` if the application of `c` removes `b` from the software system. A change `c` is said to be *invariant* w.r.t. a building block `b` if the application of `c` does not affect `b`.

Consequently, one change can be one of four kinds, depending on the building block we relate it to. A change that adds a class `C` to a package `P` for example, is a creational change w.r.t. class `C`, is an adaptive change w.r.t. package `P` and is invariant w.r.t. another (unrelated) package `Q`. It is the meta-model of the programming

language that specifies the relations between the different building blocks. FAMIX defines packages to contain classes that contain methods and attributes. Methods in turn contain statements, which consist of invocations or accesses.

The horizontal axis of Table 4.1 lists the different entities which we identified on the extended FAMIX model: Package, Class, Method, Function, Statement and all the structural entities. The vertical axis contains the different kinds of changes that can be applied to those entities. Note that each of these change kinds is actually a combination of a change and its subject. An `AddMethod` change for instance, corresponds to an instance of the `Add` class with a `Method` as its subject. For the remainder of this text, we use this notation for change kinds. [3]

The table shows that an `Add` change is creational (C) at the level of granularity of its subject and adaptive (A) at all more coarse-grained FAMIX entities that contain the subject of that change. A `Remove` change is destructive (D) at the level of granularity of its subject and adaptive (A) at all more coarse-grained FAMIX entities that contain the subject of that change. A `Modify` change is adaptive (A) at the level of granularity of its subject and also adaptive (A) at all more coarse-grained FAMIX entities that contain the subject of that change. All voids in the table consist of invariant relationships, as the corresponding change is invariant w.r.t. the corresponding entity.

Association changes

As a side effect of applying an atomic change to one of the software system's entities, there might be a modification of the relation between those entities. Such modifications are modeled by the *association changes* and are categorised in three groups: changes to the inheritance definition, changes to the method invocation and changes to the entity access. We subsequently discuss all three groups.

Changes to the inheritance definition: A FAMIX class needs to have at least one *superclass* and might have one or more *subclasses*. Consequently, when a class is created, an `AddInheritanceDefinition` change has to be created for every subclass of the new class. Every `AddInheritanceDefinition` instance models the inheritance association between the new class and its superclass. When one of the superclasses or subclasses of a class is modified, the `Modify` class is instantiated, resulting in a change that models the adaptation of the corresponding modification in the `InheritanceDefinition` subject. Similarly, when a class is removed, a `Remove` change is created for each inheritance relation that class was a part of. Note that in C++, it is allowed for a class not to have a superclass. FAMIX supports these classes by means of a dummy super class.

Changes to the method invocation: A FAMIX entity might *invoke* a behavioural entity. When such an invocation is added to an entity of the software

[3] Currently, we do not support changes regarding special language constructs such as *for loops* of *while loops*. Note that it is not necessary to model such special constructs in Smalltalk, as in that language such constructs are implemented by ordinary messages sent to a Boolean instance.

	Package	Class	Method	Function	Statement	Attribute	Implicit Variable	Local Variable	Global Variable	Formal Parameter
AddPackage	C									
ModifyPackage	A									
RemovePackage	D									
AddClass	A	C								
ModifyClass	A	A								
RemoveClass	A	D								
AddMethod	A	A	C							
ModifyMethod	A	A	A							
RemoveMethod	A	A	D							
AddFunction	A			C						
ModifyFunction	A			A						
RemoveFunction	A			D						
AddStatement	A	A	A	A	C					
ModifyStatement	A	A	A	A	A					
RemoveStatement	A	A	A	A	D					
AddAttribute	A	A				C				
ModifyAttribute	A	A				A				
RemoveAttribute	A	A				D				
AddLocalVariable	A	A	A	A	A		C			
ModifyLocalVariable	A	A	A	A	A		A			
RemoveLocalVariable	A	A	A	A	A		D			
AddImplicitVariable	A	A	A	A	A			C		
ModifyImplicitVariable	A	A	A	A	A			A		
RemoveImplicitVariable	A	A	A	A	A			D		
AddGlobalVariable	A								C	
ModifyGlobalVariable	A								A	
RemoveGlobalVariable	A								D	
AddFormalParameter	A	A	A	A	A					C
ModifyFormalParameter	A	A	A	A	A					A
RemoveFormalParameter	A	A	A	A	A					D

Table 4.1: Relations between changes and meta-model entities

system, the `AddInvocation` change is instantiated. It models the addition of an invocation relation. Certain language features (e.g. polymorphism, the absence of static typing or the presence of reflection) increase the calculation effort for retrieving the correct invocation subjects. That is why in FAMIX `Invocation` instances maintain a `candidates` attribute, that models a list of all possible targets. This is an overestimation, but can be narrowed down by static and dynamic analysis techniques [112, 19]. As that is not the core of this dissertation, we do not elaborate on this topic. The corresponding `ModifyInvocation` and `RemoveInvocation` changes respectively model the modification and removal of an invocation relation. They are instantiated when an invocation is respectively modified or removed.

Changes to the entity access: A FAMIX behavioural entity might *access* a structural entity. Assignments are instances of such accesses that modify a structural entity. When such an access is added to an entity of the software system, the `AddAccess` change is instantiated. In contrast to the invocation, FAMIX presumes that the specification of an access always reveals exactly one candidate entity that is accessed. This, however, is not always the case (for instance in Java). The same technique as with the method invocations can be used to cover this situation, but is not considered for the sake of simplification. The modification and deletion of accesses are modeled by `ModifyAccess` and `RemoveAccess` changes respectively.

All the associations between FAMIX entities model relations between those changes. They will form the basis of the dependencies that exist between changes. We elaborate on this issue in Section 4.4 but first finish the discussion of all change types by explaining the composite changes.

Change instantiation

In order to develop or evolve a software system, the above explained atomic changes need to be *created* and *carried out*. The creation of a change is done by instantiating the corresponding concrete operation class from the core model (Figure 4.9). Instantiating an `AddMethod` for instance, is done by instantiating the `Add` class with a `Method` as its subject. In order to carry out a change, it is sent the `apply` message. The change itself is responsable of carrying itself out. For that, it needs to contain all the information that specifies it. More concretely, in our model, a change is specified by:

- *what* its kind (`Add`, `Modify` or `Remove`) and subject (the affected FAMIX building block) are
- *when* it gets instantiated (`time`)
- *who* instantiated it (`user`)
- *why* it got instantiated (`intent`)

Throughout this dissertation, we use the declarative notation of Listing 4.1 to reference a change instance. *ChangeName* is the name of the change kind (e.g. `AddClass`), `parameter_list` is a list of the information that specifies the

```
ChangeName(parameter_list, time, user, intent)                    1
```

Listing 4.1: Change instantiation

```
AddClass((Buffer, Object, BufferPackage),                         1
        122233565.554,                                            2
        "Peter",                                                  3
        "BufferBaseCode")                                         4
```

Listing 4.2: Change instantiation example

change subject (e.g. name, superclass, package), `time` is an object that represents the time, `user` is an object that represents the user who created the change and `intent` is an object that denotes the *raison d'être* of the change. In a simple variation of this model, both the `intent` and `user` contain a String that describe the intent and the user respectively. In Chapter 5 we present a more powerful variation of this model in which these values are more abstract.

Listing 4.2 presents an example of a change instance. It can be read as follows: the change stands for the creation of the class `Buffer` as a subclass of `Object` in the `BufferPackage` and is instantiated on `122233565.554` by `Peter` in order to produce the `BufferBaseCode`.

4.3.2 Composite changes

In some cases, developers need to apply the same pattern of changes over and over. Examples of such reoccurring change patterns include code refactorings [40]. A "push-down field" refactoring, for instance, pushes down an attribute of a superclass to one of its subclasses. The change pattern that corresponds to this refactoring consists of first adding the attribute to the subclass and afterwards removing it from the superclass. In order to automate the application of this code refactoring, one can composite changes.

A composite change consists of an ordered collection of component changes which can in turn be a composite or atomic change. The component changes of a composite change are carried out as a transaction whenever the composite change is applied. Figure 4.10 presents two possible uses of composite changes. We now discuss those uses and afterwards explain how such changes are defined and instantiated.

Uses and advantages

Composite changes can be used to model domain-specific changes, which are specified by the developers in order to facilitate a domain-specific task. Consider that one envisions the addition of lots of new kinds of users to the chat application (from Chapter 3). In that case, it probably would be better to define an `AddUserClass`

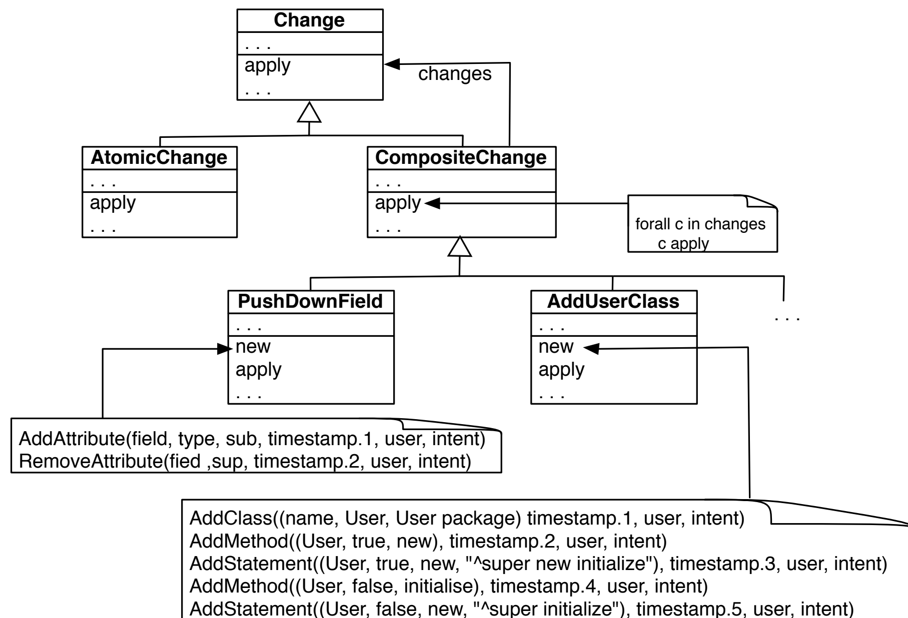

Figure 4.10: Composable first-class changes design

change, which will in turn add a subclass to the `User` hierarchy and add methods for instantiating that new class. Another well-known example of domain-specific changes are the code refactorings [40] which are used to increase the maintainability and evolvability of software systems. Every refactoring can be modeled by a composite change, that evaluates to a sequence of atomic changes whenever applied. In fact, composite changes are *higher-order* changes: changes that evaluate to a sequence of changes, whenever they are applied.

Advantages of the composite changes include that they raise the level of abstraction, and stimulate the reuse of changes. The composition of changes into composites, however, also improves the comprehensibility of the change list. A system history consisting of only atomic operations leads to an enormous and poorly organized amount of information.

Change definition

Developers can define their own domain-specific changes for the domain in which they are working. This is done by producing a subclass of the `CompositeChange` class. Figure 4.10 already depicts two examples of such definitions. The `PushDownField` is a refactoring that relocates a attribute that is only used by

```
PushDownField((attribute,sup,sub),timestamp,user,intent) if
  AddAttribute((attribute,attribute declaredType,sub),
     timestamp.1,user,intent)
  RemoveAttribute((fied,sup),timestamp.2,user,intent)
```

Listing 4.3: Change definition example 1

```
AddUserClass((name),timestamp,user,intent) if
  AddClass((name,User,User package),timestamp.1,user,intent)
  AddMethod((User,true,new),timestamp.2,user,intent)
  AddStatement((User,true,new,"^super new initialize"),
     timestamp.3,user,intent)
  AddMethod((User,false,initialise),timestamp.4,user,intent)
  AddStatement((User,false,new,"^super initialize"),
     timestamp.5,user,intent)
```

Listing 4.4: Change definition example 2

some subclass to that subclass. It consists of two atomic changes: one that removes the attribute from the super class and one that adds it to the subclass.

Listing 4.3 presents the definition of a new change kind which models a push-down field refactoring [40]. The listing shows that, in order to create a new instance of the `PushDownField` change, one needs to specify the concerned attribute `attribute`, super class `sup` and subclass `sub`. The composite change consists of two atomic changes, that are responsible for removing the attribute from the superclass and adding it to the subclass. The declarative definition of composite changes describes *what* the change should accomplish, rather than describing *how* it should accomplish it.

Listing 4.4 presents the definition of a new change kind which models a domain-specific change that is responsible of adding a new kind of user to the chat application of Figure 3.2. The listing shows that, in order to create a new instance of the `AddUserClass` change, one only needs to specify the name of the new kind of user. The composite change consists of five atomic changes, that are responsible of creating the new class representing the new kind of user as a subclass of `User` and by producing the methods for initialising that class [4].

Change instantiation

A composite change can be instantiated in the same way as atomic changes. If a `PushDownField` change is instantiated, the model ensures that an `AddAttribute` and a `RemoveAttribute` change are instantiated according to the definition.

[4]The second parameter in the AddMethod change is a predicate that denotes wether the method is added as a class method or as an instance method of the class. If the predicate is true, the change is adding a class method.

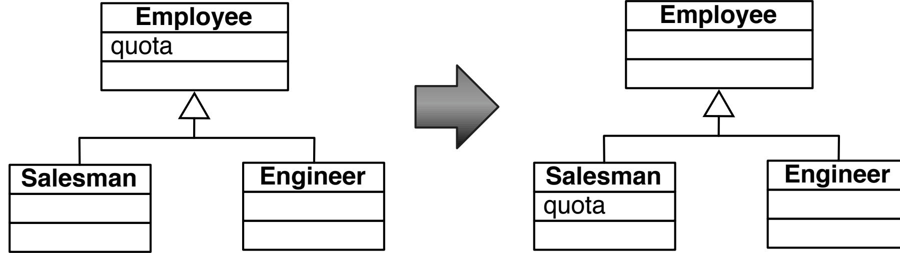

Figure 4.11: Push-down field in employee example

The `PushDownField` composite change is instantiated on the software system behind the class diagram on the left-hand side of Figure 4.11 in the following way:

$$PushDownField((quota, Employee, Salesman), 554453, Peter, "CleanUp")$$

It results in the two change instances:

$$AddAttribute((quota, String, Salesman), 554453.1, Peter, "CleanUp")$$

and

$$RemoveAttribute((quota, Employee), 554453.2, Peter, "CleanUp")$$

which, when applied, produce a software system as in the right-hand side of Figure 4.11.

When those changes are produced, they receive a time stamp. Every time stamp must be unique. In order to accomplish that, we make the time stamps of the changes follow a recursive model of nested numbers. If a change has a time stamp x and a new change is created that should receive the same time stamp x, we instead give it a time stamp $x.1$. The semantics of a time stamp $x.y$ are as follows. A change c with time stamp $x_c.y_c$ happens after all changes with a time stamp where $x < x_c$, but before all changes with a time stamp where $x > x_c$. The change happens before changes that have a time stamp where $x = x_c$ and where $y_c < y$ and after changes with $y_c > y$. The same principle is applied recursively in order to ensure that the uniqueness of the time stamps is also preserved for nested time stamps. In case a change would be assigned an already given time stamp $x.y$, it is assigned $x.y.1$ in stead.

4.4 Dependencies between change objects

In our model, every change has a set of preconditions that should be satisfied before a change is applied. Such preconditions are related to system invariants

imposed by the programming language (usually defined by the meta-model of the language). For example, methods can only be added to existing classes. Preconditions enable expressing dependency relationships between changes. In the scenario of Section 3.1.1, for instance, the change that adds the method `name` to the `RegisteredUser` class depends on the change that added the `RegisteredUser` class to the system, as the latter is the creational change for the subject of the former.

More generally, a change object c_1 is said to depend on another change object c_2 ($c_1 \rightarrow c_2$) if the application of c_1 without c_2 would violate the system invariants. In this dissertation we distinguish between *structural* and *semantical* dependencies between changes depending on the reason why the dependency holds. Structural dependencies are implied by the meta-model and can consequently be logged automatically when the changes are instantiated.

Semantical dependencies can only be derived from semantical information. This consists of programming knowledge and domain knowledge. Programming knowledge is the knowledge an expert programmer uses when writing, interpreting or debugging programs [14]. Domain knowledge is knowledge about the environment which the target system operates in. Domain knowledge is important but more difficult to obtain, because it usually must be acquainted from software users in the domain (as domain specialists/experts), rather than from software developers. An example of a semantical dependency can be found in Figure 3.3 of Chapter 3. The changes that add an `encrypt:` an `decrypt:` method to the `User` class are semantically dependent on each other as they have to be the inverse of one another. The modification of one of them, implies that the other has to be adapted as well. Semantical information is more difficult to derive and more difficult to verify. In software testing, debugging and maintenance, one is often interested in the following question to find ripple effects:

> "When can a change in the semantics of a program statement impact the execution behaviour of another statement?"

This question is undecidable in general [84]. Dependency analysis, like data flow analysis, avoids problems of undecidability by trading precision for decidability. During dependency analysis, programs are represented by defines/uses graphs, which contain limited semantical information but are easily analyzed. Dependency analysis allows semantical questions to be answered approximately, because a programs dependencies partially determine its semantical properties. Semantical and structural dependencies also provide such information, and can consequently assist in recovering semantical properties.

4.4.1 Structural dependencies

A structural dependency is a dependency that is needed to ensure the compilation of a program. Examples of a structural dependency are: a change that adds a method *depends on* the change that created the class where the method is added, or the change that adds an invocation to a method *depends on* the change that

added the invoked method. Dependencies are relations between changes and are consequently implied by the relations between the entities of the evolution model. In this subsection, we explain the seven kinds of structural dependencies we identified in our FAMIX-based evolution model.

Five kinds of structural dependencies can be derived from the FAMIX model. We name them: *hierarchical* dependencies, *accessive* dependencies, *invocative* dependencies, *genetic* dependencies and *typological* dependencies. Two kinds of dependencies can be derived from the relations between the entities in the change model itself. We call those: *transactional* dependencies and *creational* dependencies. We now itemize the seven kinds of structural dependencies that can exist between the change instances of our evolution model and illustrate them.

- A change c_1 is said to *creationally depend* on a change c_2 if c_2 is the creational change of the subject of c_1.

- A change c_1 is said to *transactionally depend* on a change c_2 if c_2 is contained in a transaction before c_1. Note that transactions of changes are specified by composite changes (see Subsection 4.3.2) in our evolution model.

- A change c_1 is said to *hierarchically depend* on a change c_2 if the subject of c_1 is in a `belongsTo` relation with the subject of c_2. FAMIX specifies this by means of a `belongsToObject`, `belongsToPackage`, `belongsToClass`, `belongsToBehaviour` or `hasArguments` relationship. If, for instance, c_1 adds a method to a class that was added by c_2, c_1 is said to hierarchically depend on c_2.

- A change c_1 is said to *accessively depend* on a change c_2 if c_2 is the creational change of the structural entity accessed by the subject of c_1. Note that instances of the FAMIX `Access` class can be used to model such dependencies. Take for instance a change c_1 that adds an access to an attribute that was added by a change c_2. In that case, c_1 is said to accessively depend on c_2.

- A change c_1 is said to *invocatively depend* on a change c_2 if c_2 is the creational change of the behavioural entity invoked by the subject of c_1. Note that instances of the FAMIX `Invocation` class can be used to model this kind of dependency. Consider for instance a change c_1 that adds an invocation to a method that was added by a change c_2. In that case, c_1 is said to invocatively depend on c_2.

- A change c_1 is said to *genetically depend* on a change c_2 if c_2 is the creational change of the superclass of the subject of c_1. Note that such dependency can be modeled by an instance of the FAMIX `InheritanceDefinition` class. As an example, consider that c_2 adds a class A and c_1 adds a class B that is a subclass of A. Than, c_1 is said to genetically depend on c_2.

- A change c_1 is said to *typologically depend* on a change c_2 if c_2 is the creational change of the type of the subject of c_1. FAMIX specifies this by means of a `declaredClass` and `declaredReturnClass` relationship between entities

```
AddClass((A,Object,P),1,Peter,"InvocationTest")                   1
AddMethod((A,false,bar),2,Peter,"InvocationTest")                 2
AddClass((B,A,P),3,Peter,"InvocationTest")                        3
AddMethod((B,false,bar),4,Peter,"InvocationTest")                 4
AddClass((C,Object,P),5,Peter,"InvocationTest")                   5
AddMethod((C,false,foo:b),6,Peter,"InvocationTest")               6
AddStatement((C,false,foo:b,"b bar"),7,Peter,"InvocationTest")    7
```

Listing 4.5: Invocative dependency example

and classes. As an example, consider that c_1 adds the class A and c_2 adds a global variable a of type A. Than, c_2 is said to typologically depend on c_1. Note that this kind of dependency only gets instantiated in case entities have a declared type. In Smalltalk, for instance, that is not the case.

Hierarchical, creational, genetical, typological and transactional dependencies between changes can be established with an absolute certainty from the moment the changes are instantiated. For example, a change that adds a method m to class C certainly depends on the change that added class C to a package. Invocative dependencies cannot always be established with a complete certainty due to language features (like polymorphism – a basic property of Object oriented programing). This is illustrated by the change history of Listing 4.5. It introduces class A with a method bar, a subclass B, with another implementation of the method bar and a class C with a method foo, that has a statement in which it invokes bar.

The actual method that is invoked depends on the type of b, the object that is passed to the foo method at runtime. The exact type of that object is only known at runtime. Consequently, at compile time, we cannot be 100% sure that the `AddStatement` change of line 7 invocatively depends on the `AddMethod` of line 2 or the one on line 6. Static type information can be used to narrow down the set of possibile candidates, since all candidates should be of the static type of b or of a type that inherits from or implements the type of b. Recent work shows that even in the absence of static type information a similar optimisation is possible [112].

In our evolution model, we start from the premise that accessive dependencies can be established at compile time with an absolute certainty. Note, however, that the FAMIX meta-model does not forbid the redefinition of an instance variable. Consequently, an existing accessive dependency between two changes can cease to exist because a new change object is introduced. This is illustrated by the example in Listing 4.6

Listing 4.6 first introduces a class A with an instance variable x, then adds a subclass of A, called B with a method foo that accesses the instance variable x of the A (the superclass of `B`). This change history contains an accessive dependency: the `AddStatement` of line 5 accessively depends on the `AddAttibute` of line 2 since the latter is the creational change of the attribute accessed by the former. Now imagine we add a new instance variable x to the class B (line 7), the accessive

```
AddClass((A, Object ,P) ,1 , Peter , "AccessTest")                    1
AddAttribute((A,x) ,2 , Peter , "AccessTest")                         2
AddClass((B,A,P) ,3 , Peter , "AccessTest")                           3
AddMethod((B, foo) ,4 , Peter , "AccessTest")                         4
AddStatement((B, false , foo , "^x") ,5 , Peter , "AccessTest")       5
AddAttribute((B,x) ,2 , Peter , "AccessTest")                         6
```

Listing 4.6: Accesive dependency example

dependency between the `AddStatement` of line 5 and the `AddAttibute` of line 2 ceases to exist, as the attribute added by the `AddAttibute` of line 7 will be accessed from now on by the statement that is added by the `AddStatement` of line 5.

4.4.2 Semantical dependencies

Semantical dependencies come from the domain knowledge. Hence, the developer is responsible for establishing these dependencies between the changes that semantically depend on one another. A semantical dependency is a relation between changes where the software application does not exhibit the desired behaviour whenever the the depending change is applied without the change it depends on. An example of a semantical dependency is when the addition of an invocation to method `m` only exhibits correct behaviour if the body of method `m` is modified in a specific way. A semantical dependency exists, for instance, between the changes that add the functionality of viewing the contents of a log file and the changes that output the logging results to that log file. If the latter changes are not applied, the former do not make sense, as they would only allow to view an empty log file.

The D_{sem} relation from Figure 4.9 is used to model the semantical dependencies between the changes. While the non-violation of structural dependencies ensures a compilable software product, the non-violation of semantical dependencies has to ensure a correctly behaving system. As it is impossible to automatically ensure the correct behaviour of an application (Rice's theorem [90]), we start from the premise that in case all semantical dependencies are satisfied, the program does behave correctly. It is up to the developer, however, to ensure that all semantical dependencies are made explicit in a software system.

4.5 Conclusion

In this chapter, we established a model that is capable of expressing the evolution of software systems to a certain extent. The model centralises change as a first-class entity. Although our evolution model is not language-specific, it targets a specific set of programming languages. More precisely, it can be used to model the evolution of all building blocks of programming languages that adhere to the FAMIX meta-model: class-based object-oriented programming languages (e.g. Java, Smalltalk, C++, Ada, etc.).

The FAMIX model, however, is not capable of modelling code statements. We extend the FAMIX model with this notion in order to obtain a finer-grained model. The FAMIX model and the extension that we applied to it are the subject of Sections 4.1 and 4.2. The evolution model itself is presented in Section 4.3. It is a generic model that is capable of modeling the evolution of any software system written in a language adhering to FAMIX.

In Section 4.4, we discuss the different kinds of dependency relations between change objects. We distinguish between structural and semantical dependencies and explain how structural dependencies can be discovered automatically as they are derived from the meta-model. Based on the relations between the entities of our evolution model, we identify seven kinds of structural dependencies. Hierarchical, creational, genetical, typological and transactional dependencies between changes can be established with an absolute certainty from the moment the changes are instantiated. Invocative and accessive dependencies cannot always be established with a complete certainty due to language features like polymorphism, late binding or reflection. Semantical dependencies can only be derived with the help of domain experts and software developers. While structural dependencies can be asserted to ensure program compilation, semantical dependencies are a help in establishing correctly behaving software products.

The evolution model is capable of modeling the *addition, modification and deletion* of software building blocks at a level of granularity down to the *level of statements*. *Dependencies* between the change objects are also modeled in our evolution model. We conclude that our evolution model is the first state-of-the-art model of first-class changes that has all characteristics required for it to be usable as a basis for feature-oriented programming (Table 2.3 on page 36). The following chapter elaborates on how this is done.

Chapter 5

Feature-oriented programming through change-oriented programming

In Chapter 2 we already elaborated on what feature-oriented programming (FOP) is. An interesting observation is that different researchers have been proposing different views of what a feature is or should be. Most state-of-the-art approaches to FOP specify a feature as a set of building blocks. An alternative, already pointed out by Batory in [10], is to see a feature as a function that modifies a program, so that feature composition becomes function composition. Indeed, a feature would be seen as a function $prog \rightarrow prog$ that takes a program, modifies it by adding the functionality that implements the feature's requirements, and returns the modified program. The application of a feature to a program yields a new program, extended by that feature, which in turn some other feature can be applied to. In that view, a system is obtained by applying a sequence of features to a base program. The AHEAD toolchain [10] was recently formalised this way [3].

In this chapter we work towards a novel approach to FOP, which overcomes three issues we identified in the related work: the lack of control for feature modularisation, the undesired side-effect that FOP requires a developer to alter the development process and the non-existence of a bottom-up approach to FOP. The approach is based on change-oriented programming (ChOP) that we presented in Chapter 3. We first illustrate the principles of our approach by means of an example and introduce the concept of a change specification. Afterwards, we show how this approach overcomes the flaws we identified in the related work to FOP.

5.1 Principles

5.1.1 Features as functions

In this dissertation, we follow the intuition of Batory and see a feature as a function that modifies a program in order to increase the capabilities of that program with the capability modeled by that feature. A capability can be of a functional or non-functional nature (see Section 2.2 for more information). We define a feature as a *structure that has to be applied to a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision or to offer a configuration option* and we show how ChOP enables a bottom-up approach to FOP that provides more control over feature modularity. In the following subsections, we use a `Buffer` base program in order to present the building blocks of the approach: the changes and the dependencies between them.

5.1.2 Changes as feature building blocks

In ChOP, software systems are developed and evolved by instantiating changes of a change model (illustrated in Chapter 4). Those instances actually represent all the development operations that were carried out to implement the complete software system. Consequently, the application of all those changes, results in a complete version of the software system. However, one can also apply *only a part of* those changes in order to get another version of the same software system. Such software system is actually a *variation* of the original software system – one with less functionality. This technique is the core of our approach to FOP.

Consider a `Buffer` program that follows the *value object* pattern [42] in order to provide the functionality of a buffer. The left hand side of Figure 5.1 presents the Java code of the that buffer software application. It does not show what development actions were performed in order to actually produce that code. Actually, first the `Buffer` class was added by an `AddClass` change (B1). Afterwards, an instance variable called `buf` was added by an `AddAttribute` change (B2). Finally, the methods `get` (B3) with body (B4) and `set` (B5) with body (B6) were added by twice respectively instantiating and applying an `AddMethod` and an `AddStatement` change.

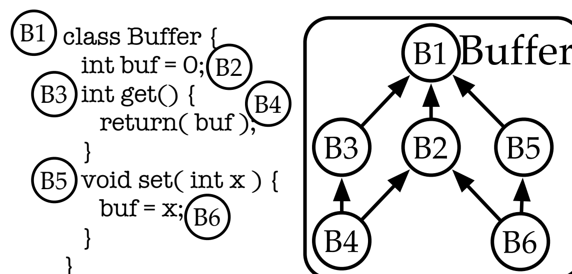

Figure 5.1: Source code (left) and change objects (right) of the `Buffer`

The right part of Figure 5.1 presents all the first-class change objects that resulted from the development operations taken to produce the buffer. Next to that, the figure also shows that all these changes are grouped into a set of changes that represent that buffer. Conceptually, the drawing in the right part of Figure 5.1, presents a feature as a set of all the development actions that were taken in order to add the functionality of that feature (denoted by a name and a line surrounding all the concerned change objects).

The edges between the change objects represent the dependencies between the change objects that were introduced in Section 4.4. The following subsection explains in more detail how these dependencies are modeled and used in the context of FOP.

5.1.3 Dependencies

In our model of ChOP, a change object c_1 is said to depend on another change object c_2 ($c_1 \rightarrow c_2$) if the application of c_1 without c_2 would violate one of the system invariants. We differentiate between structural dependencies (ensuring structural invariants) and semantical dependencies (ensuring behavioural invariants). While the former are automatically derived from the meta-model, the latter depend on domain knowledge and must be instantiated manually. Both kinds, however, are represented in the same way in Figure 5.1: A dependency between a change c_1 and a change c_2 is represented by an edge from c_1 to c_2.

A software system often consists of more than one functionality. In ChOP, a software system is always implemented incrementally. Consequently, the features of a software system are added one after the other. We extend the buffer application with a `Restore`, a `Logging` and a `Multiple Restore` feature which respectively add the functionality of restoring the value of the buffer, logging the values of all instance variables whenever a method of the buffer is executed and allowing the buffer to restore more than one value. Figure 5.2 presents the code of the adapted application. From left to right, the features Restore, Logging and Multiple Restore are implemented. The corresponding change objects are presented in Figure 5.3.

The dependencies between changes are not always confined to a single feature, but can reach changes of other features as well. For example, in Figure 5.3, changes within the `Restore` feature depend on changes inside the `Buffer` feature. Based on those dependencies one can state that the `Restore` feature depends on the `Buffer` feature. Indeed, the dependencies that hold between certain of these change objects can be propagated to the feature level. The dependency of a change $R1$ of a feature $Restore$ on another change $B1$ from a feature $Buffer$ implies that $Restore$ is dependent on $Buffer$. Hence, feature dependencies are modeled by means of fine-grained dependencies between change objects.

```
class Buffer {
  int buf = 0;
R1 int back = 0;
  int get() {
    return( buf );
  }
  void set( int x ) {
    back = buf;  R3
    buf = x;
  }
R2 void restore() {
    buf = back;  R4
  }
}
```

```
class Buffer {
  int buf = 0;
  int back = 0;
  int get() {
    logit(); L2
    return( buf );
  }
  void set( int x ) {
    logit(); L3
    back = buf;
    buf = x;
  }
  void restore() {
    logit(); L4
    buf = back;
  }
L1 void logit() {
    print(back); L5
    print(buf); L6
  }
}
```

```
class Buffer {
  int buf = 0;  M1
  Stack back = Stack new();
  int get() {
    logit();
    return( buf );
  }
  void set( int x ) {
    logit();  M2 M3
    back.push(x);
    buf = x;
  }
  void restore() {
    logit();  M4 M5
    buf = back.pop();
  }
  void logit() {  M6
    print(back.top());
    print(buf);
  }
}
```

Figure 5.2: Source code of adding `Restore` (left), `Logging` (middle), `Multiple Restore` (right)

5.2 Mathematical properties

We now discuss the mathematical properties of the dependency relation between change objects. We define the structural dependency relation to be irreflexive, asymmetric and transitive and define the semantical dependency relation to be irreflexive and transitive. We wrap up this section by introducing a dependency graph: a mathematical graph consisting of change objects (as the nodes) and dependencies (as the relation).

5.2.1 The dependency relation

We differentiate between the two kinds of dependency relations and elaborate on their mathematical properties. For that, we first introduce the dependency operator, $c_1 \rightarrow c_2$ – an alternative notation for $(c_1, c_2) \in D$ – meaning that the change c_1 depends on c_2. This means that c_1 needs c_2 to be applied first.

Structural dependencies

In Section 4.4.1, we already elaborate on the different kinds of structural dependencies. All these structural dependencies have similar properties, which we now explain in detail. Given the changes c_1, c_2, c_3 which belong to the set of change objects C and the relation $D_{str} \subset C \times C$ with $(c_1, c_2) \in D_{str}$ (in the alternative no-

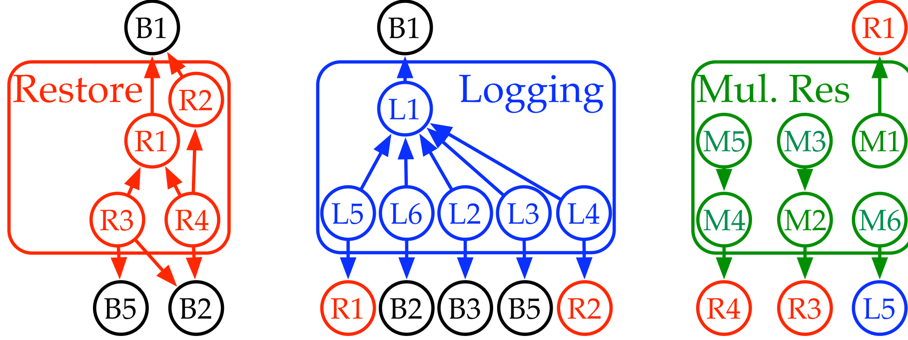

Figure 5.3: Change objects of `Restore` (left), `Logging` (middle), `Multiple Restore` (right)

tation, this boils down to $c_1 \stackrel{str}{\rightarrow} c_2$). Then, we can say that D_{str} is a homogeneous, binary endorelation over C. An endorelation is a binary relation where the domain and codomain are the same. The relation D_{str} has the following properties:

- D_{str} *is irreflexive*:

$$\forall c_1 \in C \bullet (c_1, c_1) \notin D_{str} \tag{5.1}$$

  In our change model, a change can only be applied if the changes which it depends on have already been applied. A change can never structurally depend on itself. In case a change c_1 would hierarchically depend on itself for instance, c_1 would have to affect a subject that is in a `belongsTo` relation with itself. The evolution model from Chapter 4 ensures that that is never the case. In fact, it is the FAMIX core that ensures the irreflexive property for hierarchical, accessive, invocational, genetical and typological dependencies, while the change model ensures it for transactional and creational dependencies. Expressed by means of the dependency operator, this property is $c_1 \stackrel{str}{\not\rightarrow} c_1$.

- D_{str} *is asymmetric*:

$$\forall c_1, c_2 \in C \wedge (c_1, c_2) \in D_{str} \Rightarrow (c_2, c_1) \notin D_{str} \tag{5.2}$$

  If a change c_1 structurally depends on a change c_2, c_2 may never structurally depend on c_1. This is enforced by the FAMIX and change model. Take for instance that the dependency from above is a genetical one, that implies that the subject of c_1 is a subclass of the subject of c_2. FAMIX ensures that the subject of c_2 can not be a subclass of the subject of c_1. In fact, FAMIX ensures that this property holds for the four first kinds of dependency, while

the change model ensures this for the latter two kinds. A shorter notation of this property is: $c_1 \overset{str}{\rightarrow} c_2 \Rightarrow c_2 \overset{str}{\not\rightarrow} c_1$.

- D_{str} *is transitive*:

$$\forall c_1, c_2, c_3 \in C \wedge (c_1, c_2) \in D_{str} \wedge (c_2, c_3) \in D_{str} \Rightarrow (c_1, c_3) \in D_{str} \tag{5.3}$$

It means that if c_1 structurally depends on c_2 and c_2 structurally depends on c_3, then c_1 structurally depends on c_3. In other words, if c_1 requires c_2 to be applied and c_2 requires $c3$ to be applied, c_1 also requires c_3 to be applied. As an example, take a change c_3, that adds a attribute buf to a class $Buffer$, a change c_2, that modifies the name of that attribute to $buff$ and a change c_1, that adds a statement which accesses the attribute $buff$. The FAMIX and change model ensure that c_1 accessively depends on c_2 and c_2 creationally depends on c_3. Consequently, c_1 is said to depend structurally on c_3 since $buff$ cannot be accessed if buf was not first added to the class (and afterwards renamed to $buff$). By means of the dependency operator, this property is expressed as: $c_1 \overset{str}{\rightarrow} c_2 \wedge c_2 \overset{str}{\rightarrow} c_3 \Rightarrow c_1 \overset{str}{\rightarrow} c_3$.

Consequently, we can state that the structural dependency relation D_{str}, as defined above, is a *strict partial order* (a binary relation that is irreflexive, transitive, and asymmetric). Since C is a set with a strict partial order relation defined on it, C is said to be a *strictly partially ordered set*. Strictly partially ordered sets are useful because they can be easily mapped to directed acyclic graphs (DAGs): Every strict partial ordered set is a DAG, and the transitive closure of a DAG is both a strict partial ordered set and also a DAG itself. This property is exploited in Section 5.2.2, in which we introduce dependency graphs over change objects. Let us first describe the mathematical properties of the semantical dependencies.

Semantical dependencies

A semantical dependency is a relation between changes where the software application does not exhibit the desired behaviour whenever the depending change is applied without the change it depends on. Consider for instance, the changes $R1$ and $R2$ from the *Restore* feature. Each of those changes can be applied independently from the other without the resulting software application violating the meta-model. The resulting software application, however, would not contain the restore functionality if either of the changes would not be applied. Consequently, there is a semantical dependency between $R1$ and $R2$. In fact, there is a semantical dependency between every two changes of the restore feature, since omitting one of the changes would result in an application with an undesired behaviour.

A semantical dependency also exists between the changes that add the functionality of viewing the contents of a log file and the changes of our *Logging* feature, which output the results of the logging to a log file. If the latter changes are not applied, the former do not make sense, as they would only allow viewing

an empty log file. It still makes sense, however, to include the changes of the *Logging* feature without including the changes to view the log file.

Given the changes c_1, c_2, c_3 which belong to the set of change objects C and the relation $D_{sem} \subset C \times C$ with $(c_1, c_2) \in D_{sem}$ modeling the semantical dependency relation "c_1 needs c_2 to be applied together" (in the alternative notation, this boils down to $c_1 \overset{sem}{\rightarrow} c_2$). Then, we can say that D_{sem} is an homogeneous, binary endorelation over C with the following properties:

- D_{sem} *is irreflexive*:

$$\forall c_1 \in C \bullet (c_1, c_1) \notin D_{sem} \tag{5.4}$$

  A change can never semantically depend on itself as that would imply that the change would have to be applied before itself in order to obtain a correctly behaving software application. Expressed by means of the dependency operator, this property is $c_1 \overset{sem}{\not\rightarrow} c_1$.

- D_{sem} *is transitive*:

$$\forall c_1, c_2, c_3 \in C \wedge (c_1, c_2) \in D_{sem} \wedge (c_2, c_3) \in D_{sem} \Rightarrow (c_1, c_3) \in D_{sem} \tag{5.5}$$

  This means that if c_1 depends on c_2 and c_2 depends on c_3, then c_1 depends on c_3 or in other words, if c_1 requires c_2 to be applied and c_2 requires c_3 to be applied, c_1 also requires c_3 to be applied. By means of the dependency operator, this property is expressed as: $c_1 \overset{sem}{\rightarrow} c_2 \wedge c_2 \overset{sem}{\rightarrow} c_3 \Rightarrow c_1 \overset{sem}{\rightarrow} c_3$.

With respect to symmetry, we cannot say the semantical dependency relation is symmetric or asymmetric. While some instances of the semantical dependency relation are symmetric (the dependencies from the *Restore* example explained above), others are not (the dependencies from the *Logging* example explained above). As semantical dependencies only depend on domain knowledge, they cannot automatically be derived by the meta-model. Developers should be allowed, however, to make those dependencies explicit. For that, tool support must be provided by the development environment.

5.2.2 Dependency graphs

In this section, we first establish a model that allows expressing software products as graphs that contain changes as the nodes and structural dependencies as the edges. Afterwards, we explain the mathematical properties of such graphs. We do not consider semantical dependencies in those graphs. At the end of the section, we explain why.

Given a set of change objects C and the relation $D_{str} \subseteq C \times C$ with $(c_1, c_2) \in D_{str}$, we can establish a graph $G = (C, D_{str})$. The nodes of the graph are the elements of the change set C. The edges of the graph (denoted by tuples (c_1, c_2))

are the structural dependency relations between the change objects. Together, they form the set of edges of G, called D_{str}.

We now express a program family P as a graph with the changes as its nodes (C) and the structural dependencies between those changes as its edges (D_{str}) and define this graph as a tuple:

$$P = (C, Dstr) \tag{5.6}$$

An edge $v_1 = (c_1, c_2)$ is considered to be directed from c_1 to c_2; c_1 is called the *origin* and c_2 is called the *destination* of the edge; c_2 is said to be a direct successor of c_1, and c_1 is said to be a direct predecessor of c_2. A path `p` is a sequence of nodes $[c_1, c_2, c_3]$ such that from each of its nodes there is an edge to the next node in the sequence ($(c_1, c_2), (c_2, c_3) \in D_{str}$). We introduce an alternative notation $c_1 \rightarrow c_2 \rightarrow c_3$ as a shortcut for $c_1 \overset{str}{\rightarrow} c_2$ and $c_2 \overset{str}{\rightarrow} c_3$. If a *path* leads from c_1 to c_3, then c_3 is said to be a successor of c_1, and c_1 is said to be a predecessor of c_3. The dependency graph of a program family $G = (C, D_{str})$ is a graph with the following properties:

- *G is directed*

$$\forall c_1, c_2 \in C \wedge (c_1, c_2), (c_2, c_1) \in D_{str} \Rightarrow (c_1, c_2) \neq (c_2, c_1) \tag{5.7}$$

  A graph is directed if it has directed edges only. As $D_{str} \subset C \times C$, the dependency graph G is directed.

- *G is acyclic*

$$\nexists c_1 ... c_n \in C \bullet c_1 \rightarrow ... \rightarrow c_n \rightarrow c_1 \tag{5.8}$$

  An acyclic graph is a graph that does not contain any cycles. Consequently there can never exist a path that starts and ends in the same node of the graph. The transitive and asymmetric properties of the structural dependency relation guarantee that the dependency graphs will never contain cycles.

We can conclude that a dependency graph based on the changes and the structural dependency relation is a directed acyclic graph (a DAG). Each DAG gives rise to at least a partial order R on its edges, where uRv holds when there exists a path from u to v in the DAG. Informally speaking, such graphs *flow* in a single direction and can be used as an input of a topological sorting algorithm. This property is used below to sort the changes of a change set in order of applicability.

The inclusion of semantical dependencies between changes in the graph would result in a graph $G = (C, D_{sem} \cup D_{str})$ which is directed, not oriented and maybe cyclic. Such graphs cannot always be fully topologically sorted. As the reason of the graphs is to ease the sorting of changes (as we will see below), we chose to omit the semantical dependencies from the dependency graphs. Semantical dependencies do not affect the order of applicability of changes anyway, as that order is only enforced by the invariants of the meta-model. Semantical dependencies will be used in Section 5.4.3, however, in order to validate software compositions.

5.3 Advantages

In the previous sections we worked towards a novel approach to FOP, which is based on ChOP. We now explain the four advantages this approach to FOP has with respect to the state-of-the-art approaches to FOP. This approach provides *control for feature modularisation*, does *not require one to adapt his development process*, provides *detailed composition support* and enables a *bottom-up approach to FOP*.

None of the state-of-the-art approaches to FOP allow features to express the *deletion* of software building blocks, although this is often required. The *granularity* the approaches provide rarely reaches the statement level and in case it does, it is limited. AHEAD, Lifting Functions, Mixin-layers, FeatureC++ and most AOP approaches do allow the specification of a feature that adds a statement to an existing method by means of the `super` call. This construct, however, only allows the expression of a statement addition before and/or after the complete old method behavior. With those approaches, it is not possible to specify a feature that adds a statement between the statements of an existing method. With the exception of FeatureC++ and the AOP approaches, none of the above techniques provide means to specify crosscutting feature. They require an alternative implementation of such feature depending on the features present in the composition, hindering reusability. All these symtoms show that the state-of-the-art approaches to FOP provide only a limited control for feature modularisation.

To the best of the author's knowledge, all the state-of-the-art approaches to FOP require a programmer to alter the development process. Most of the approaches introduce new programming concepts such as features (by FeatureC++), refinements (by AHEAD), mixins or mixin layers (by the Mixin approach) that the programmer should use in order to do FOP. Most of the approaches also require the use of a specific Interactive Development Environment (IDE) or IDE plugin, which enables the use of those concepts. Consequently, the programmer is required to adapt his development process. ChOP enables an approach to FOP in which the programmer does not need to alter his development process: It does not introduce new concepts nor requires a specific programming language or IDE. We believe such an approach to FOP increases its usability in an industrial context, where companies typically do not want to alleviate from their development methodologies and exerted programming language.

Our approach to FOP ensures that a lot of fine-grained development information is available in the entire development process. Whenever a new software product variation needs to be produced as a composition of feature modules, this information can be used in order to support the software composition process. Inconsistencies can be presented to the developer on a very detailed level in order to assist the debugging process. This information is based on what changes cannot be included in the composition and the reason why they cannot be included: what dependent change should be included first. Because changes can be topologically sorted in an automatic way (Section 5.2.2), the automatic production of software product variations that offer different combinations of functionality is supported.

Finally, none of the existing approaches to FOP support *bottom-up FOP*: An approach in which the individual software building blocks are first specified in great detail and only afterwards grouped in feature modules. As this is the key contribution of this dissertation, we dedicate a complete section to this advantage.

5.4 Bottom-up approach to FOP

FOP is the study of feature modularity, where features are raised to first-class entities [7]. In FOP, features are basic building blocks, which satisfy intuitive user-formulated requirements on the software system. A software product is built by composing features. In this section, we present a bottom-up approach to FOP which consists of three phases. First, change operations have to be captured into first-class entities. Second, these entities have to be classified in features (= separate change sets that each implement one functionality). Finally, these feature modules can be recomposed in order to form software variations that provide different functionality. The following subsections discuss every step of the approach.

5.4.1 Obtaining the changes

In this first phase of the bottom-up approach to FOP, the modifications to the source code resulting from development or evolution actions are to be captured in first-class objects. We present four different techniques to accomplish this task.

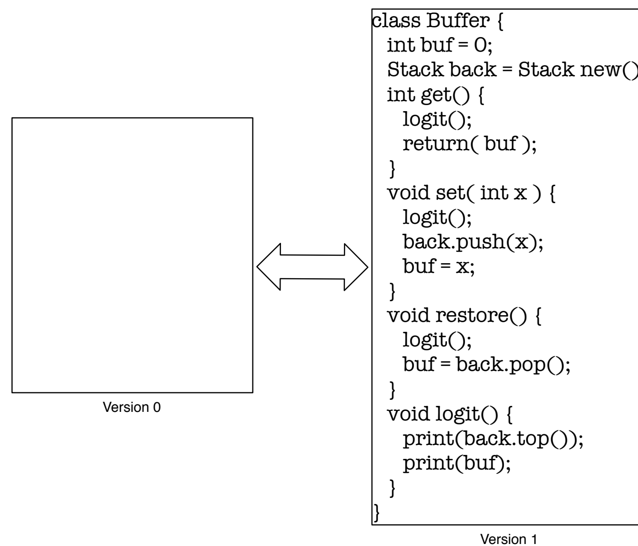

Figure 5.4: Reconstruction of the change sets by means of the differentiation technique

A classic way to obtain changes is by means of the *differentiation* strategy in which for instance the Unix `diff` command is executed on the respective source code files of two finished versions of a software system [113]. This technique reveals the changes at a high level of granularity (the version level) of two snap-shots of the software system. Every snap-shot corresponds to a version that was committed to a code repository. The technique is applicable when development is over and can consequently be used on legacy systems. In [91], Robbes shows that such an approach does not provide a full overview of all the development actions that were taken to take the first version to the second. Consider for instance the statement $buf = back.pop()$; which was actually first added as $buf = back$; in order to implement the restore feature and afterwards modified to $buf = back.pop()$; in order to implement the multiple restore. The application of this strategy on the two versions of Figure 5.4 would, however, result in only one change in which the latter statement is added.

A more subtle alternative is to apply *Chane-oriented Programming*. In ChOP, a developer develops or evolves software systems in a change-driven way by instantiating change objects. These change objects are to be maintained in first-class objects, so that they can be referenced afterwards. We see two drawbacks in this approach. First, it is only applicable to systems that are being developed. Second, we find the development of applications by means of pure ChOP not to be realistic. We do not believe a programmer will actually execute a dialog for adding a statement to a method body for instance. The third strategy overcomes this problem by supporting a more pragmatic ChOP.

The third strategy is based on *logging* the developer when taking development or evolution actions. In this strategy, the IDE is instrumented with a logging mechanism responsible to instantiate change objects that represent all the actions the developers take. The advantage of this approach is that it seems more realistic than pure ChOP, which would require the development process to be adapted. The main drawback of this technique is that it still cannot be used on legacy systems. A secondary drawback might involve privacy issues, as the programmers are required to allow "Big Brother" to watch them.

A final, less important way of obtaining changes is by *receiving them* from a source that already captured the changes. That source can be another developer or a repository that contains the changes that were committed by one or more developers. In order to support this technique, the change objects need to be first-class, persistent and already collected by one of these four strategies.

5.4.2 Classification of Changes

In this subsection, we focus on the second phase in the bottom-up approach to FOP: the classification of changes into modules each representing a feature. Classification has two aspects: the classification *model* and the classification *technique*, which is embodied in the different software classification strategies. We subsequently discuss both.

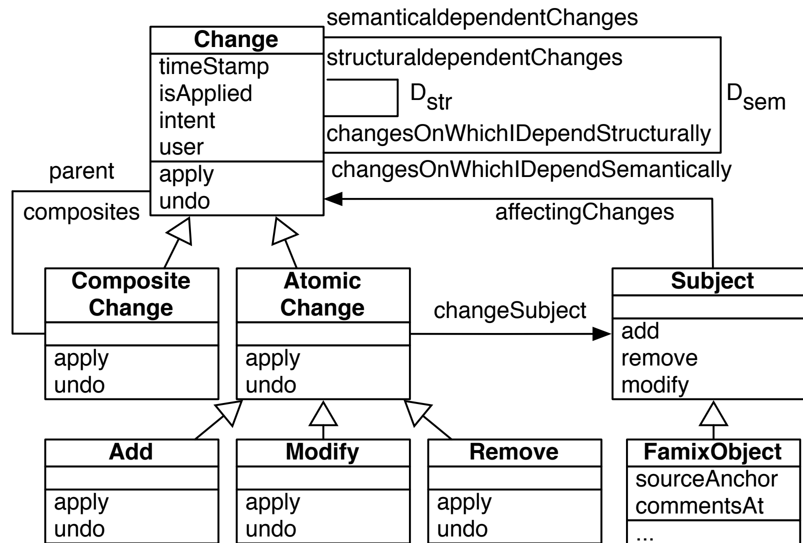

Figure 5.5: Change model

Classification model

The classification model is a meta-model that consists of two parts: the change model and the actual classification model. Each part focuses on another level of granularity. The change model describes how the changes are represented. Figure 5.5 shows that the change model separates between four kinds of changes, which can be composed. Atomic changes have a `subject`: the program building block affected by the change and defined by the Famix model [27].

The actual classification model defines and describes the entities of the superstructure, which is a flexible organisational structure based on feature and change objects. Figure 5.6 shows that the model contains three relations: D_{str}/D_{sem} (the structural and semantical dependencies that hold between the change objects), $F4C$ (which changes are grouped together into which feature) and Sub (which features are contained within another feature). The `cardinality` of the $F4C$ and Sub entities is a boolean attribute that specifies whether or not the son (change and/or feature) has to be included in a composition that includes the parent (a feature). This information can afterwards be used to validate feature compositions (as in Feature Diagrams [53]).

Classification techniques

A classification strategy is a method for setting up classifications. Many classification strategies can be devised ranging from setting up classifications manually to generating classifications automatically. We present three classification strate-

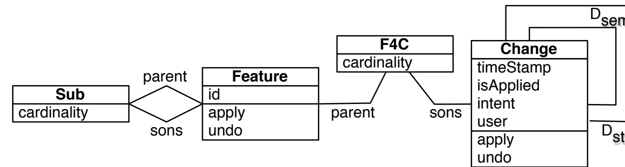

Figure 5.6: Classification model

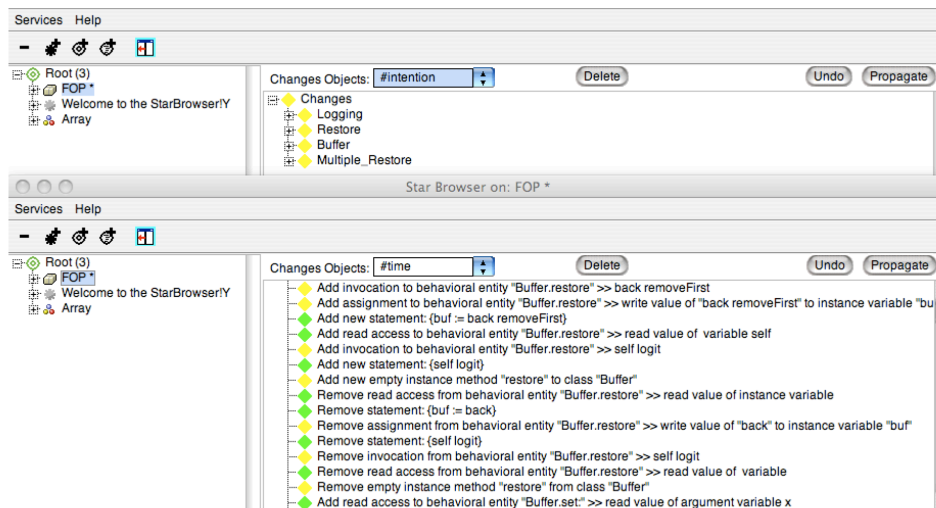

Figure 5.7: Manual classification

gies: *manual* classification, *semi-autmatic* classification through clustering and *automatic* classification through forward tagging.

Manual classification Manual classification is the simplest classification strategy: manually classifying change objects into modules that represent a feature. The strategy can be used by the software engineer to group changes according to his wishes. Since our classification model states that a change can only be classified in one feature, this strategy should be supported by a tool which enforces that rule.

The advantages of this strategy are twofold. First, it is a very straightforward technique which can easily be implemented and included in an IDE. Second, it can be successfully applied to change objects that were obtained with all of the strategies to obtain the changes (differentiation, ChOP, logging or receiving). The main disadvantage is the tedium that comes with the manual effort of this strategy. The lower part of Figure 5.7 presents a list of change objects that were produced in order to create the buffer example. In order to use this technique, the developer

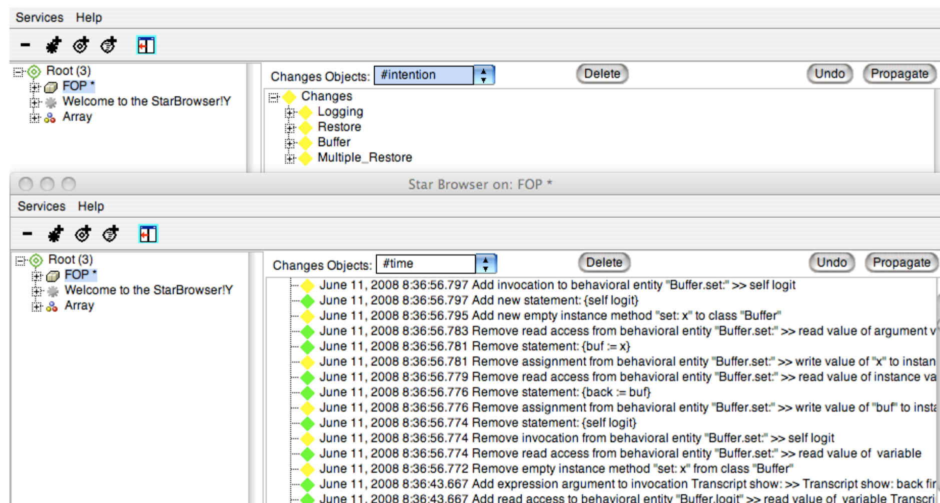

Figure 5.8: Semi-automatic classification

is required to manually go over all the changes in order to classify them by hand in the categories presented in the upper part of Figure 5.7. This manual effort is both error-prone and laborious.

Semi-automatic classification Change objects contain information about *by whom*, *when*, *why* and *where* the operations they represent were carried out. Using clustering techniques [92] based on metrics on these properties, change objects can be grouped in a semi-automatic way. Such classification is always based on an assumption: for instance "the changes that are performed by the same user in a limited time interval are all done for the same purpose". In order for such strategy to work, it should be used in collaboration with a manual classification strategy. Based on the the clusters of changes, the developer has the final responsibility on deciding how the changes must actually be classified.

The main advantage of this strategy is that it can be used to assist the developer in doing manual classification. The disadvantages of this heuristic strategy are threefold. First, it is more difficult to implement as it requires the definition of distance measures, the clustering process and a (graphical) representation of the results. Second, it always depends on an assumption, which does not hold by definition. Different parameters in the metrics might give different clustering results. Extra research is required to find adequate parameters, which might differ depending on the domain. Third, the success of this strategy largely depends on the amount of information available in the change set and within the change objects. It is consequently not recommended to be used in combination with a differentiation strategy – in which changes do not contain detailed information and in which some changes might not be recovered at all.

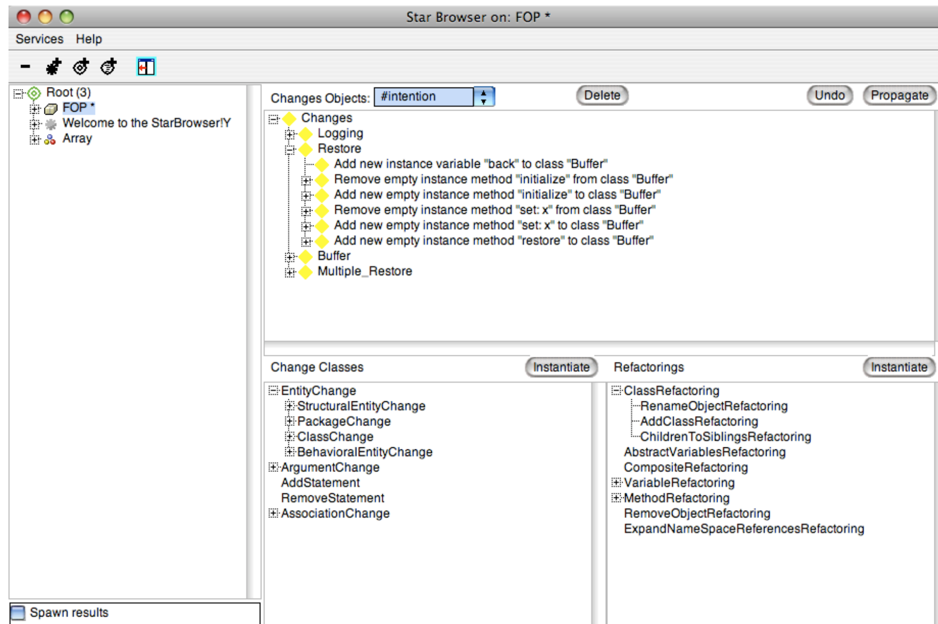

Figure 5.9: Automatic classification

Automatic classification In many cases, a manual classification strategy is not a feasible option. For large software systems it would take a long time to classify all changes by hand. Often classification of a software system is an activity that cannot be done by one software engineer alone since one software engineer seldom knows the entire system. When manual classification is not a valid option for the classification problem at hand, automatic classification may provide a solution.

The automatic strategy is based on the assumption that changes already contain the information on what feature they partially implement. This information is preserved in information tags that are attached to the change objects. These tags can be processed automatically to generate tag-based classifications. Figure 5.9 presents the automatic classification results of the changes that were logged while developing the buffer example.

The advantages of this approach are that it can be applied to the largest of systems as it does not require manual labour and that it is relatively easy to implement. The sole inconvenience is that it can only be used in combination with a change collection strategy that is based on ChOP or logging. The reason for that is it requires every change object to contain the information that denotes in which module it should be classified. That information should be provided beforehand by the developer. We propose to use a *forward tagging* technique in order to obtain this information. The idea behind forward tagging is that when a developer carries out a development operation, for example implementing a new

or changed specification or fixing a bug, he usually knows the context in which the changes are made. When making the changes, he provides that information to the IDE. Moreover, the IDE knows the exact time, and in what part of the software, the change is performed. Instead of keeping this knowledge implicit in the heads of the developers, the IDE *tags* it to the changes.

Since software developers often do not have sufficient time when source code documentation is concerned, it is not realistic to rely on discipline for the forward tagging process. Consequently, the IDE should require that developers provide the information at development time.

Discussion

We introduced a model and three strategies to classify changes and/or features in sets that represent features. The model consists of two parts which respectively model the change objects and the actual classification. The first strategy is straightforward: *manual* classification is a strategy to put together classifications manually; *Semi-automatic* classification is based on clustering changes together based on properties such as *by whom*, *when*, *why* and *where* the changes were applied; *Automatic* classification is based on forward tagging, and automatically groups changes together.

Only the automatic strategy is usable in practice, as it is the only strategy that requires minor manual labour without an increase in error probability. As this strategy requires forward tagging, it is only applicable in combination with pure ChOP or logging as a technique to capture changes. In case a software system was already developed (e.g a legacy system), the system usually does not contain the forward tagged information. Consequently, the changes collected from such systems can only be classified by means of a manual classification strategy, which might be supported with a semi-automatic technique. We conclude that the development environment should support logging or ChOP in order to enforce forward tagging, so that the changes can automatically be classified in recomposable feature modules.

5.4.3 Change composition

In our approach, a software system is specified by all the changes that have been instantiated in order to implement it. The software system can afterwards be (re)generated by (re)applying those changes. Consider the buffer example which we introduced in the beginning of this chapter. Figure 5.10 shows the set of all change objects that were instantiated in order to produce the buffer of which the source code is shown in Figure 5.4. It also contains the dependencies between those changes and classifications of changes into features. Note that we internally split up the `Logging` feature into three subfeatures: `LogMethod`, `PrintInstVars` and `InvokeLog`. We define the diagram included in Figure 5.10 as the *change specification* of the buffer application.

A change specification CS is a graph that consists of *nodes*, *edges* and *groupings*. The nodes are the change objects C that resulted from instantiating the

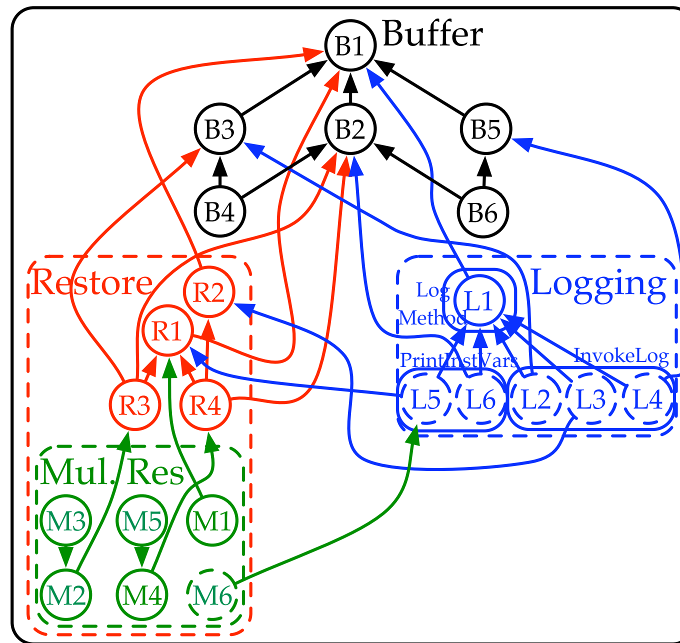

Figure 5.10: Change specification of the buffer example

change classes from the change model (represented by the `Change` class from Section 4.3). The edges denote the structural and semantical dependencies between the changes (modeled by the `D` relation in the change model of Figure 5.5 of Section 4.4). Finally, a grouping form is an entity that denotes the concept of a feature and which groups a set of changes and/or features (modeled by the `F4C` and `Sub` relations respectively in Figure 5.6 of Section 5.4.2). Every change or grouping can be surrounded by a full or dashed line. A dashed line denotes that the change or grouping is *optional* with respect to its parent (the grouping to which it belongs). A full line specifies that the change or grouping is *mandatory* with respect to its parent grouping. This is modeled by the `cardinality` attribute of the `F4C` and `Sub` relations of Figure 5.6 of Section 5.4.2.

Every subset of the set of changes instantiated when developing a software system represents a different *variation* of that system. Since a change specification holds the complete set of changes C, it contains the specification of more than one variation of the software system. Moreover, the change specification can be used to check the validity of such a variation. In our approach, a variation `Var` is valid if all parent changes of the change objects of `Var` are part of `Var`. In FOP, however, the different software variations are to be expressed in terms of required features. A *composition* is a set of features that have to be included in a variation. As every feature consists of a set of changes, the composition is a variation that consists of the union of all changes of those features. An invalid composition is the result of

composing features that contain a change c_1 of which at least one dependency is not satisfied: There is a path $[c_1, ..., c_2]$ in CS for which c_2 is not in the change composition.

The composition of features is the mechanism that allows the "easy" creation of the different software variations that are part of the product's product line. The composition has to list all the features that correspond to the desired functionality. If, for instance, a buffer variation with logging capacities is required, the composition should state: {`Buffer, Logging`}. Since `LogMethod`, `PrintInstVars` and `InvokeLog` are all mandatory with respect to `Logging`, they all need to be included in the composition. Since $B1$, $B2$, $B3$, $B4$, $B5$, $B6$, $L1$ are all mandatory to their parent, they *have to be* included as well. Finally, there are some changes that *can be* included in the composition: $L2$, $L3$, $L4$, $L5$ and $L6$. They are all specified as optional with respect to their parent. This means that a composition containing their parent does not need to contain those changes in order to be valid.

Different composition strategies are conceivable. Algorithm 1 on page 115 produces a maximal change composition, in which all changes that can be included will be included in a composition. This conservative strategy makes sense, since it produces the system with the most complete implementation – in terms of changes – of the corresponding feature set. In this example, this strategy boils down to the inclusion of $L2$, $L4$ and $L6$. The change $L3$ and $L5$ are not included in the composition because they respectively have the parents $R2$ and $R1$, which do not reside in the composition.

Other strategies can also be considered. The *minimal change composition* for a set of features returns the mandatory changes and the optional ones that are required to make the composition valid. The composition of a buffer with logging in this strategy boils down to only including $B1$, $B2$, $B3$, $B4$, $B5$, $B6$, $L1$. Note that applying the minimal composition strategy does not make sense in this setting, as it produces a composition that actually does not include the logging behaviour. The operations that actually add the logging behaviour are omitted from the composition, as they are optional and not required to make the composition valid. Conceptually, however, such a composition strategy might be interesting in the case where code size needs to be minimised, as it returns the most basic implementation of a feature set. Yet another strategy could be a mix of both in which the optional changes that add code are included and the optional changes that remove code are omitted. This, however, remains a topic for future work.

Algorithm 1 presents an algorithm that verifies the validity of a composition and produces the maximal change set in case the composition is valid. It receives as input a list containing the features of the required composition and the change specification (which consists of a set of changes C, a set of structural and semantical dependencies D and a set of groupings that denote the relations between the features). Note that D being a union of semantical and structural dependencies might imply that there are dependency cycles in the change specification. The algorithm takes that into account and when finished, it returns a list that consist of two lists. The first list contains the changes in the order in which they must

be applied to produce the required composition. The second list consists of the changes that had to be omitted from the composition because of unsatisfied dependencies. In case the latter is empty, the validation succeeded. In case the latter is not empty, it can be used by the developer to correct unwanted composition errors.

Input: A feature set F, a change specification CS
Output: A list consisting of 2 change sets

$F_{min} \leftarrow minimal_feature_set(F, CS)$;
$C_{min} \leftarrow minimal_change_set(F_{min}, CS)$;
$C_{unw} \leftarrow unwanted_change_set(F_{min}, C_{min}, CS)$;
$C^{+}_{unw} \leftarrow transitive_closure(C_{unw}, CS)$;
$C_{err} \leftarrow$ all mandatory changes that are in $C^{+}_{unw} \setminus C_{unw}$;
return ($C \setminus C^{+}_{unw}$, C_{err})

Algorithm 1: $validateComposition(F, CS)$ function

Input: A feature set F, a change specification CS
Output: A feature set

$F_{min} \leftarrow F$ **foreach** $f \in F$ **do**
    F_{min} add: f;
    F_{min} add the transitive closure of the parent features of f;
    F_{min} add the transitive closure of the mandatory sub-features of f;
end
return F_{min}

Algorithm 2: $minimal_feature_set(F, CS)$ subroutine

Input: A feature set F_{min}, a change specification CS
Output: A change set

$C_{min} \leftarrow \emptyset$;
foreach $c \in C$ **do**
    if c *is mandatory* $\wedge$ *feature of* $c \in F_{min}$ **then**
        C_{min} add: c
    end
end
return C_{min}

Algorithm 3: $minimal_change_set(F_{min}, CS)$ subroutine

The order of the time complexity of Algorithm 1 is the sum of the time complexities of all its subroutines. Algorithms 2 and 3 have a time complexity of respectively $O(f + Sub)$ and O(f) where f is the number of features and Sub the

Input: A feature set F_{min}, a change specification CS
Output: A change set

```
C_unw ← ∅;
foreach c ∈ C do
    if c ∉ C_min and feature of c ∉ F_min then
        C_unw add: c
    end
end
return C_unw
```

Algorithm 4: *unwanted_change_set*(F_{min}, C_{min}, CS) subroutine

Input: A change set C_{unw}, a change specification CS
Output: A change set C^{+}_{unw}

```
S ← C_unw;
D ← edges in the change specification;
C+_unw ← ∅;
while S ≠ ∅ do
    c ← remove a change c from S;
    C+_unw add: c;
    foreach c2 with d in D from c2 to c do
        D remove: d;
        S add: c2
    end
end
return C+_unw
```

Algorithm 5: *transitive_closure*(C_{unw}, CS) subroutine

number of relations between features. Algorithms 4 and 5 have a time complexity of respectively $O(c)$ and $0(c + d)$ where c is the number of changes and d the number of dependencies in the change specification.

In the best case, this total boils down to $O(n)$ where n is the number of changes in the composition. It occurs when there are no changes with dependencies and when the subroutine in Algorithm 5 can consequently be calculated in n steps. In the worst case the order is $O(n+e)$, which is the result of the depth-first traversal for calculating the transitive closure of unwanted changes in a graph with n change nodes and e dependency edges. This also boils down to a complexity of $O(n)$, since the number of dependencies and changes are usually of the same order of magnitude.

We can conclude that this algorithm calculates the validity of a certain program variation in an amount of time that grows linearly with the amount of changes and dependencies that are present in the change specification. Note that some optimisations can be conceived that reduce the time needed to validate a variation.

These optimisations are not considered in this dissertation and remain topic for future research. As a tradeoff for this performance, the algorithm needs extra working memory to do the necessary bookkeeping.

5.5 Conclusion

In this chapter, we introduced a novel approach to Feature-Oriented Programming (FOP). The approach is based on the view that a feature is a function that has to be applied to a software system in order to add the corresponding functionality to that software system. The building blocks of a feature are instances of the different change kinds (*Add*, *Remove* or $Modify$). In some cases, a change depends on another change in a *structural* or *semantical* way. In our model of ChOP, a change object c_1 is said to depend structurally (resp. semantically) on another change object c_2 if c_1 cannot be applied without c_1 without breaking one of the system invariants imposed by the meta-model (resp. obtaining a incorrectly behaving application).

The structural dependency relation is shown to be a strict partial order (an irreflexive, asymmetric, transitive binary endorelation) over the set of changes. The semantical dependency relation is a transitive binary relation over the set of changes. We introduce dependency graphs as graphs in which the nodes are changes and the edges are structural dependencies between those changes. We show that such a graph is directed and acyclic when only structural dependencies are considered. This is an interesting property as it allows sorting the changes topologically based on the structural dependency relation. The resulting sequence of changes is applicable as the syntactic dependencies are always satisfied whenever a change is applied.

Advantages of describing features as sets of changes that need to be applied in order to add a functionality to a system are threefold. First, this approach enables a *higher control* for feature modularisation, since features can contain the deletion of software building blocks and this on the *fine-grained level of granularity* of code statements. Second, this approach does not require a developer to alter his development process. Features can be constructed while programming in an object-oriented way. Finally, it allows a bottom-up approach to FOP, in which the fine-grained development information can be used to enable an automatic composition of software variations and in which the developer can be supported when debugging software compostions.

All state-of-the-art approaches to FOP are top-down approaches in which an overview of the system is first formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. We present a bottom-up approach to FOP which adheres more to ad-hoc development. With a bottom-up approach, the software system can be developed in a feature-oriented way without having to design it up-front completely.

The approach consists of three phases. First, the change operations have to be captured into first-class entities. We present three techniques to do that which

we name differentiation, Change-oriented programming and logging. Second, the change entities have to be classified in features (= separate change sets that each implement one functionality). We present a classification model and three classification techniques (manual, semi-automatic and automatic classification). Finally, these feature modules can be recomposed in order to form software variations that provide different functionality. We introduce a change specification: a diagram that describes the entire product family of a software system and which can be used to validate different compositions. We end this chapter with a maximal composition algorithm that produces a software variation for a given feature composition and change specification.

Chapter 6

Formalism for feature composition

In this dissertation, we present change-oriented programming (ChOP), a programming paradigm that centralises change and enables a specific approach to FOP. In ChOP, each feature is represented by a set of *changes* applied to a base system [31, 32]. Feature composition, in this case, becomes change classification and composition. One of the challenges in ChOP is to make sure that a composition of several features, viz. changes, will succeed. In this chapter, we present a formal model for ChOP, that allows us to define this kind of property and which may serve as a reference for ChOP implementations. First, we formalise its concepts and properties using basic set theory. Then, we define properties such as *composability* of features that need to be checked before features can be composed.

Software Product Line Engineering is a software engineering paradigm that institutionalises reuse throughout software development. An interesting observation is that our model is quite close to feature diagrams, a software product line engineering notation used to model the variability of an application at an – up to now – relatively coarse-grained level of granularity [53, 9, 95]. The main purpose of feature diagrams it to model which combinations of features are allowed and disallowed in a software product line. By mapping the formal model of ChOP to feature diagrams, we are able to reuse their well-studied semantics as well as existing analysis tools. We prove that *composability* of features in ChOP is equivalent to checking whether the product they form satisfies the feature diagram. We first recall the definition of feature diagrams.

6.1 Feature Diagrams

Feature diagrams were introduced by Kang et al. as part of the FODA method [53], and have become one of the standard modelling languages for variability in Software Product Line Engineering [86]. The purpose of a feature diagram is to define

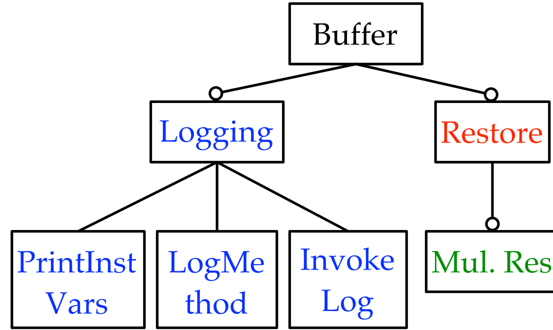

Figure 6.1: Buffer feature diagram

concisely which combinations of features are allowed in a product family. An example feature diagram, based on the buffer example, is shown in Figure 6.1.

Basically, a feature diagram is a hierarchy of features, where the hierarchy relation denotes decomposition. The feature diagram represents the set of allowed feature combinations (called *configurations*), thus a set of sets of features, and several types of decomposition operators determine what is allowed and what not. An *and* decomposition, for instance, means that all child-features need to be included in a configuration if their parent is, while an *or* decomposition requires at least one child-feature to be included. These two decomposition types can be represented with a generic cardinality decomposition $\langle i..j \rangle$ where i indicates the minimum number of children required in a configuration and j the maximum. Table 6.1 lists the classic decomposition types from feature diagrams and shows how they can all be expressed by means of a generic cardinality decomposition $\langle i..j \rangle$.

Decomposition type	Cardinality
An optional child	$\langle 0..1 \rangle$
xor decomposition	$\langle 1..1 \rangle$
or decomposition	$\langle 1..* \rangle$
and decomposition with s children	$\langle s..s \rangle$

Table 6.1: Cardinality as a way to describe feature decomposition

Some authors also consider optional features, generally represented in feature diagrams as a square with a hollow circle above it, which need not be included in a configuration, even if mandated by the decomposition operator. In addition to decomposition operators, a feature diagram can be annotated by constraints in a textual language, such as propositional logic [9], that further restrain the set of allowed configurations.

A number of feature diagram dialects have appeared in the literature since their original proposal [95]. In this chapter, we use the visual syntax of Czarnecki and Eisenecker [21], and the formal semantics of Schobbens et al. [95] which we recall in the following definitions.

Definition 1 (Feature diagram Abstract Syntax)**.** *A feature diagram d is a 6-tuple $(N, P, r, \lambda, DE, \Phi)$ where:*

- *N is the (non empty) set of features (or* nodes*),*
- *$P \subseteq N$ is the set of primitive features (i.e. those considered relevant by the modeller),*
- *$r \in N$ is the root,*
- *$DE \subseteq N \times N$ is the decomposition relation between features. For convenience, we will sometimes write $n \to n'$ instead of $(n, n') \in DE$, where n is the parent of n' and n' is the child of n.*
- *$\lambda : N \to \mathbb{N} \times \mathbb{N}$ indicates the decomposition type of a feature, represented as a cardinality $\langle i..j \rangle$ where i indicates the minimum number of children required in a configuration and j the maximum.*
- *$\Phi \in \mathbb{B}(N)$ is a set of propositional logic formulae on features, expressing additional constraints on the diagram.*

Furthermore, each d must satisfy the following well-formedness rules:

- *Only r has no parent: $\forall n \in N (\nexists n' \in N \bullet n' \to n) \Leftrightarrow n = r$,*
- *DE is acyclic: $\nexists n_1, .., n_k \in N \bullet n_1 \to .. \to n_k \to n_1$,*
- *Every minimum of a decomposition is smaller than the corresponding maximum: $\forall \langle i..j \rangle \in \lambda : i < j$*
- *Terminal nodes are $\langle 0..0 \rangle$ decomposed.*

Definition 2 (Feature diagram Semantics)**.** *Given a feature diagram d, its semantics $[\![d]\!]$ is the set of all* valid feature combinations *$FC \in \mathcal{PP}N$ (= the powerset of powersets of N) restricted to primitive features: $[\![d]\!] = \{c \cap P | c \in FC\}$, where the valid feature combinations FC of d are those $c \in \mathcal{P}N$ (= the powerset of N) that:*

- *contain the root: $r \in c$;*
- *satisfy the decomposition type:*
  $f \in c \wedge \lambda(f) = \langle m..n \rangle \Rightarrow m \leq |\{g | g \in c \ \wedge \ f \to g\}| \leq n$;
- *include each selected feature's parent: $g \in c \wedge f \to g \Rightarrow f \in c$;*
- *satisfy the additional constraints: $\forall p \in \Phi \bullet c \models p$.*

The semantics of the diagram in Figure 6.1 is the set

$$\begin{aligned}\{&\{Buffer\},\\ &\{Buffer, Logging, PrintInstVars, LogMethod, InvokeLog\},\\ &\{Buffer, Restore\},\\ &\{Buffer, Restore, Mul.Res\},\\ &\{Buffer, Logging, PrintInstVars, LogMethod, InvokeLog,\\ &\quad Restore\},\\ &\{Buffer, Logging, PrintInstVars, LogMethod, InvokeLog,\\ &\quad Restore, Mul.Res\}\}\end{aligned}$$

For the remainder of this chapter, unless otherwise stated, we always assume d to denote a feature diagram, and $(N, r, \lambda, DE, \Phi)$ to denote the elements of its abstract syntax.

6.2 A formal model for ChOP

In this section, we first provide a formal model of the intuitive notions of ChOP presented in Chapter 4 and define some basic properties such as *composability*.

6.2.1 Fundamental concepts

The principal concept in ChOP is the *change object*, which encapsulates a development step. A change can be applied to a software system in order to execute the development operation it encapsulates. Let C be the set of all change objects that make up the system. Another important concept is that of a *feature*, so let F denote the set of all features f_i in the system. As seen in Chapter 4, features consist of changes and can have sub-features.

Dependencies between features

Consider the sub-feature relation. A feature f_i can consist of sub-features, which can each be mandatory (man) or optional (opt), as captured by the relation Sub

$$Sub \subseteq F \times F \times \{man, opt\}, \tag{6.1}$$

where the first element denotes the parent feature and the second the child. For convenience, we will note

$$\begin{aligned} f_1 \overset{man}{\rightarrow} f_2 \quad & \text{if } (f_1, f_2, man) \in Sub,\\ f_1 \overset{opt}{\rightarrow} f_2 \quad & \text{if } (f_1, f_2, opt) \in Sub, and\\ f_1 \overset{?}{\rightarrow} f_2 \quad & \text{if } (f_1, f_2, man) \in Sub \vee (f_1, f_2, opt) \in Sub. \end{aligned}$$

The relation Sub needs to satisfy two well-formedness constraints.

- A sub-feature is either mandatory or optional.

$$\forall f_1 \overset{?}{\rightarrow} f_2 \Rightarrow \neg(f_1 \overset{man}{\rightarrow} f_2 \wedge f_1 \overset{opt}{\rightarrow} f_2) \tag{6.2}$$

- The relation contains no cycles, and each feature has no more than one parent.

$$\{(f_1, f_2)|f_1 \xrightarrow{?} f_2\} \text{ forms a forest}^1. \tag{6.3}$$

We define $roots(F)$ as a function that returns the set of all features of F that do not have a parent feature:

$$roots(F) = \{f \in F | \nexists f' \in F \bullet f' \xrightarrow{?} f\}$$

Dependencies between features and changes

A feature generally consists of changes $c \in C$ which can also be mandatory (man) or optional (opt). This is formalised with the function $F4C$

$$F4C : C \to F \times \{man, opt\}, \tag{6.4}$$

which each change associates with its parent feature. For convenience, we will note

$$\begin{array}{ll} f \xrightarrow{man} c & \text{if } F4C(c) = (f, man), \\ f \xrightarrow{opt} c & \text{if } F4C(c) = (f, opt), and \\ f \xrightarrow{?} c & \text{if } F4C(c) = (f, man) \vee F4C(c) = (f, opt). \end{array}$$

Dependencies between changes

In Chapter 4 we already elaborated on the properties of the structural and semantical dependencies that are respectively imposed by the FAMIX meta-model and by the knowledge of the problem domain. We define those relations and recapitulate those properties with respect to the formal concepts and properties that we defined in this chapter.

Given changes c_1, c_2 which belong to the set of change objects C, change c_1 is said to structurally depend on change c_2, if the application of c_1 without the application of c_2 would violate the rules specified by the meta-model. The structural dependencies between changes are denoted by the relation D_{str},

$$D_{str} \subseteq C \times C, \tag{6.5}$$

which is required to be:

- D_{str} *is irreflexive*:

$$\forall c_1 \in C \bullet (c_1, c_1) \notin D_{str} \tag{6.6}$$

- D_{str} *is asymmetric*:

$$\forall c_1, c_2 \in C \wedge (c_1, c_2) \in D_{str} \Rightarrow (c_2, c_1) \notin D_{str} \tag{6.7}$$

[1] A forest is a disjoint union of trees. Trees are graphs in which any two vertices are connected by exactly one path.

- *D_{str} is transitive*:

$$\forall c_1, c_2, c_3 \in C \wedge (c_1, c_2) \in D_{str} \wedge (c_2, c_3) \in D \Rightarrow (c_1, c_3) \in D_{str} \qquad (6.8)$$

In other words, D_{str} is a strict partial order over C. The semantical dependencies between changes, are denoted by the relation D_{sem},

$$D_{sem} \subseteq C \times C, \qquad (6.9)$$

which is required to be:

- *D_{sem} is irreflexive*:

$$\forall c_1 \in C \bullet (c_1, c_1) \notin D_{sem} \qquad (6.10)$$

- *D_{sem} is transitive*:

$$\forall c_1, c_2, c_3 \in C \wedge (c_1, c_2) \in D_{sem} \wedge (c_2, c_3) \in D_{sem} \Rightarrow (c_1, c_3) \in D_{sem} \qquad (6.11)$$

In other words, D_{sem} is a irreflexive and transitive endorelation over C. For the sake of simplicity, we also define D as the union of the D_{str} and D_{sem} relations. D actually groups all the dependency relations between change objects,

$$D = D_{str} \cup D_{sem}, \qquad (6.12)$$

which is only required to be transitive. For convenience, we write that c_1 depends on c_2 as $c_1 \rightarrow c_2$ if $(c_1, c_2) \in D$. For the remainder of this chapter, we do not differentiate between D_{str} and D_{sem} anymore, but consistently refer to D if it comes to dependencies between changes. Reasons for this are that (a) this simplifies the formalism and (b) change specifications are only used to validate software compositions and the fact wether a dependency is semantical or structural does not matter to that regard. We refer to Section 5.2.2 on page 103 for more information on the differences between semantical and structural dependencies.

Change specification

In addition to the well-formedness constraints on Sub, we require that each feature must have sub-features, changes, or both – as in our approach to FOP, a feature is always specified by a set of change objects or features.

$$\forall f \in F \;\bullet\; (\exists f' \in F \bullet f \stackrel{?}{\rightarrow} f') \vee (\exists c \in C \bullet f \stackrel{?}{\rightarrow} c) \qquad (6.13)$$

Considered together, all these concepts make up a *change specification* as the following definition records.

Definition 3 (Change specification)**.** *A* change specification *Cs is a 5-tuple $Cs = (C, F, Sub, F4C, D)$, where $C, F, Sub, F4C$ and D are as defined above.*

As an example of a change specification, consider the software application from Figure 6.2 on page 130. With respect to Definition 3, the change specification of the buffer application is a 5-tuple $(C, F, Sub, F4C, D)$ where,

$$
\begin{aligned}
C &= \{B1, B2, B3, B4, B5, B6, R1, R2, R3, R4, M1, M2, \\
&\quad M3, M4, M5, M6, L1, L2, L3, L4, L5, L6\}, \\
F &= \{Buffer, Restore, Mul.Res, Logging, LogMethod, \\
&\quad PrintInstVar, InvokeLog\}, \\
Sub &= \{Buffer \overset{opt}{\to} Restore, Restore \overset{opt}{\to} Mul.Res, Buffer \overset{opt}{\to} Logging, \\
&\quad Logging \overset{man}{\to} LogMethod, Logging \overset{man}{\to} PrintInstVar, \\
&\quad Logging \overset{man}{\to} InvokeLog\}, \\
F4C &= \{Buffer \overset{man}{\to} B1, Buffer \overset{man}{\to} B2, Buffer \overset{man}{\to} B3, \\
&\quad Buffer \overset{man}{\to} B4, Buffer \overset{man}{\to} B5, Buffer \overset{man}{\to} B6, \\
&\quad Restore \overset{man}{\to} R1, Restore \overset{man}{\to} R2, Restore \overset{man}{\to} R3, \\
&\quad Restore \overset{man}{\to} R4, Mul.Res \overset{man}{\to} M1, Mul.Res \overset{man}{\to} M2, \\
&\quad Mul.Res \overset{man}{\to} M3, Mul.Res \overset{man}{\to} M4, Mul.Res \overset{man}{\to} M5, \\
&\quad Mul.Res \overset{opt}{\to} M6, LogMethod \overset{man}{\to} L1, PrintInstVars \overset{opt}{\to} L5, \\
&\quad PrintInstVars \overset{opt}{\to} L6, InvokeLog \overset{opt}{\to} L2, InvokeLog \overset{opt}{\to} L3, \\
&\quad InvokeLog \overset{opt}{\to} L4\}, and \\
D &= \{B2 \to B1, R1 \to B1, L1 \to B1, M1 \to R1, B3 \to B1, R2 \to B1, \\
&\quad L2 \to L1, M2 \to R3, B5 \to B1, R3 \to R1, L2 \to B3, M3 \to M2, \\
&\quad B4 \to B3, R3 \to B3, L3 \to L1, M4 \to R4, B4 \to B2, R3 \to B2, \\
&\quad L3 \to R2, M5 \to M4, B6 \to B2, R4 \to R1, L4 \to L1, M6 \to L5, \\
&\quad B6 \to B5, R4 \to R2, L4 \to B5, R4 \to B2, L5 \to L1, L5 \to R1, \\
&\quad L6 \to L1, L6 \to B2\}.
\end{aligned}
$$

6.2.2 Properties

From the fundamental concepts, we can now define properties of change specifications, such as the property of *composability* mentioned earlier. Let us first define what *change compositions* and *legal change compositions* are.

Definition 4 (Change composition)**.** *A* change composition *is a set of changes $H \subseteq C$ (with $H \neq \emptyset$) that can be applied together to construct a software system.*

For the remainder of this text, we use H to refer to a change composition, unless explicitly mentioned otherwise.

Definition 5 (Legal change composition, legal feature set)**.** *A* legal change composition H *is a change composition such that there exists a* legal feature set $G \subseteq F$, *which satisfies the following constraints*

- *If a feature is selected, its parent feature must be selected, too:*

$$\forall f \in G \bullet r \overset{?}{\to} f \implies r \in G \tag{6.14}$$

- *If a feature with mandatory sub-features is selected, the latter need to be selected, too:*
$$\forall r \in G \bullet r \overset{man}{\rightarrow} f \implies f \in G \tag{6.15}$$
- *Let* $M = \{c | f \in G \wedge f \overset{man}{\rightarrow} c\}$*, the set of mandatory changes and* $O = \{c | f \in G \wedge f \overset{opt}{\rightarrow} c\}$*, the set of optional changes. From Definition 6.4, we know that* $O \cup M = F$ *and that* $O \cap M = \emptyset$*. We need that:*
  - *all changes that are mandatory with respect to the selected features are in:*
  $$M \subseteq H \tag{6.16}$$
  - *all changes stem from selected features:*
  $$H \setminus M \subseteq O \tag{6.17}$$
- *All dependencies are satisfied*
$$\forall c \in H \bullet \exists (c, c') \in D \implies c' \in H \tag{6.18}$$

By extension, we will say that such a G is a *legal feature set* for H with respect to Cs; or that the changes H are *composable*. H is the set of features in which the changes of G are contained (through $F4C$). For the remainder of this text, we always assume G to be a feature set, unless otherwise stated. In order to prove that there is at least one legal change composition and feature set for a given change specification (Theorem 7), we first define the semantics of a change specification.

Definition 6 (Semantics of a change specification)**.** *The* semantics of a change specification Cs*, noted* $[\![Cs]\!]$*, is defined as the set of pairs* (H, G) *such that* H *is a legal change composition of* Cs *and* G *is a legal feature set for* H *with respect to* Cs *according to the above definition.*

Note that this definition does not imply that there is only one legal change composition for a single legal feature set and vice versa. The following theorem establishes that every change specification has at least one legal change composition, i.e. $[\![Cs]\!] \neq \emptyset$ for every change specification Cs.

Theorem 7 (Legal composition existence)**.** *For each change specification* Cs*, there is at least one legal change composition and feature set:*

$$[\![Cs]\!] \neq \emptyset$$

Proof. We prove this theorem by constructing a legal composition: (C, F) is a valid change composition ($(C, F) \in [\![Cs]\!]$) because it complies to the constraints of Definition 5. Constraints 6.14, 6.15 and 6.18 are satisfied trivially by the equations 6.1 and 6.12. Given M and O as defined in Definition 4, we need to show that $M \subseteq H$ and that $H \setminus M \subseteq O$. Because a change is either optional or mandatory to a feature, we know that $F = O \cup M$ and that $O \cap M = \emptyset$. From that we know that both properties hold. □

Given a legal feature set G, there might be several legal change compositions. And similarly, given a legal change composition H, there can be several legal feature sets such that $(H, G) \in [\![Cs]\!]$. For proving this, simply consider the following example. Let us assume we have a Cs with only $f \stackrel{opt}{\rightarrow} f'$, $f \stackrel{man}{\rightarrow} c$ and $f' \stackrel{opt}{\rightarrow} c'$. In this case, we have $[\![Cs]\!] = \{(\{c\}, \{f\}), (\{c\}, \{f, f'\}), (\{c, c'\}, \{f, f'\})\}$.

Consequently, we will define the notions of minimal and maximal change compositions and prove their unicity in Theorem 10.

Definition 8 (Minimal change composition)**.** *Given H, G and Cs such that $(H, G) \in [\![Cs]\!]$, H is said to be* minimal *if $\nexists H' \cdot H' \subset H \wedge (H', G) \in [\![Cs]\!]$.*

Definition 9 (Maximal change composition)**.** *Given H, G and Cs such that $(H, G) \in [\![Cs]\!]$, H is said to be* maximal *if $\nexists H' \cdot H' \supset H \wedge (H', G) \in [\![Cs]\!]$.*

In the small example we gave above, $\{c\}$ is both the minimal and maximal change composition with respect to $\{f\}$, whereas with respect to $\{f, f'\}$, we have a minimal change composition $\{c\}$ and a maximal change composition $\{c, c'\}$. The following theorem proves the unicity of the minimal and maximal change composition in the general case.

Theorem 10 (Uniqueness of minimal and maximal change compositions)**.** *A minimal change composition is unique. And so is a maximal change composition.*

Proof. Let us consider building a change composition as follows. We include in H all mandatory changes with respect to G and only those optional changes required to satisfy the dependencies steming from mandatory changes. With respect to Definition 5, this means we build:

$$H = M \cup (O \cap \{c \mid c' \in M \wedge (c', c) \in D^+\})$$

where D^+ is the transitive closure of D. Such an H is legal, unique and minimal since we cannot remove any optional change from it without violating a dependency. The unicity of the maximal change composition is proved by considering $H = M \cup O$ which makes H legal, unique and maximal. □

Similarly, we can define the notions of minimal and maximal feature set and prove their unicity in Theorems 13 and 14.

Definition 11 (Minimal feature set)**.** *Given H, G and Cs such that $(H, G) \in [\![Cs]\!]$, G is said to be* minimal *if $\nexists G' \cdot G' \subset G \wedge (H, G') \in [\![Cs]\!]$.*

Definition 12 (Maximal feature set)**.** *Given H, G and Cs such that $(H, G) \in [\![Cs]\!]$, G is said to be* maximal *if $\nexists G' \cdot G' \supset G \wedge (H, G') \in [\![Cs]\!]$.*

Theorem 13 (Uniqueness of minimal feature sets)**.** *A minimal feature set is unique.*

Proof. Let us assume we have a legal change composition H. All legal G associated to it satisfies $\forall c \in H \cdot G \subseteq \{f \mid f \stackrel{?}{\rightarrow} c\}$ since all changes need to be justified by a

feature. Since H is fixed, one can only add features to G that do not require more changes being added to H.

Minimally, to make G legal, only those features that help satisfy Sub should be added. This means that for each $c \in H$ with $f \stackrel{?}{\rightarrow} c$, we need to include in G (1) the feature f, (2) the set Anc_f of all its ancestors, (3) the mandatory descendants of f, noted $Desc_f^{man}$, and (4) the mandatory descendants of the features in Anc_f. Because the resulting feature set should be legal with respect to H, those features do not require more changes. We now define this formally.

In what follows, we will use the notation $f \stackrel{man+}{\rightarrow} f'$ to mean:

$$\exists f_1, f_2 \ldots f_n \cdot f \stackrel{man}{\rightarrow} f_1 \wedge f_1 \stackrel{man}{\rightarrow} f_2 \wedge \ldots \wedge f_{n-1} \stackrel{man}{\rightarrow} f_n \wedge f_n \stackrel{man}{\rightarrow} f'$$

and the notation $f \stackrel{?+}{\rightarrow} f'$ to mean:

$$\exists f_1, f_2 \ldots f_n \cdot f \stackrel{?}{\rightarrow} f_1 \wedge f_1 \stackrel{?}{\rightarrow} f_2 \wedge \ldots \wedge f_{n-1} \stackrel{?}{\rightarrow} f_n \wedge f_n \stackrel{?}{\rightarrow} f'.$$

If we define $Anc_f = \{f' \mid f' \stackrel{?+}{\rightarrow} f\}$ and $Desc_f^{man} = \{f' \mid f \stackrel{man+}{\rightarrow} f'\}$ then the unique, legal and minimal feature set G_2 can be constructed as follows:

- $G_0 = \{f \mid f \stackrel{?}{\rightarrow} c \wedge c \in H\}$
- $G_1 = G_0 \cup \bigcup_{f \in G_0} Anc_f$
- $G_2 = G_1 \cup \bigcup_{f \in G_1} Desc_f^{man}$

□

Theorem 14 (Uniqueness of maximal feature sets)**.** *A maximal feature set is unique.*

Proof. Let us again prove this theorem by means of construction. Starting from G_2 built according to the previous proof, we can now possibly add (1) root features that were not already in G_2 plus some of their descendants, and (2) optional descendants of features already in G_2. If we want to keep a change composition that is legal with respect to H, the selected additional features cannot require to add any change that is not already in H. Those requirements are fulfilled by the algorithms 6 and 7 which provide a unique legal and maximal set G_3. □

Input: A minimal feature set G_2
Output: A maximal feature set G_3

$candidates \leftarrow roots(F) \setminus G_2 \cup \{f \mid f' \stackrel{opt}{\rightarrow} f \wedge f' \in G_2\}$;
$added \leftarrow addCandidates(candidates)$;
$G_3 \leftarrow G_2$;
while $added \neq \emptyset$ **do**
  $G_3 \leftarrow G_3 \cup added$;
  $candidates \leftarrow \{f \mid f' \stackrel{opt}{\rightarrow} f \wedge f' \in added\}$;
  $added \leftarrow addCandidates(candidates)$;
end

Algorithm 6: Constructing a maximal feature set

$added \leftarrow \emptyset$;
foreach $f \in candidates$ **do**
  if $\forall f' \in Desc^{man}(f) \bullet \exists c \in C \bullet f' \stackrel{man}{\rightarrow} c$ **then**
    $added \leftarrow added \cup \{f\} \cup Desc^{man}(f)$
  end
  return $added$;
end

Algorithm 7: *addCandidates* subroutine

6.3 From change specification to feature diagram

The goal of this section is to define a way to systematically translate a change specification into a feature diagram (like the one in Figure 6.1), so that the resulting feature diagram has the *same meaning* as the change specification. Such a procedure has two main benefits. First, generating and then visualising a feature diagram can provide an alternative representation of the change specification, which is in many cases more readable. Second, and more importantly, having a formal feature diagram opens the way for automated queries and reasoning about the change specification through the use of a feature diagram tool such as FAMA [11] or the one described in [74]. In particular, since ChEOPS is a valid implementation of our formal ChOP model, this allows a safe and efficient integration of ChEOPS with those tools. Let us first give an intuition of what such a translation might look like and what 'the same meaning' means.

Figure 6.1 shows a feature diagram inspired by the buffer example. Even though this diagram is based on the *description* of the buffer example rather than the corresponding change specification (Figure 6.2), it illustrates part of the translation. Indeed, the change specification is made up of features and changes, as well as the relations between them. The feature hierarchy of ChOP translates almost immediately into a feature diagram (like Figure 6.1).

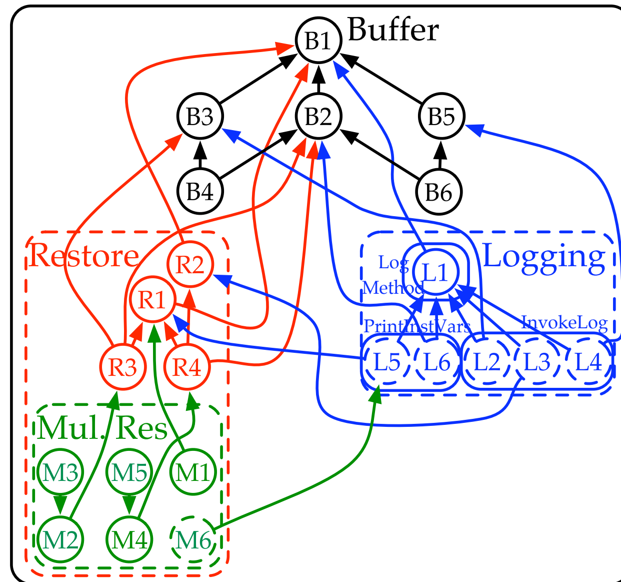

Figure 6.2: Change specification of the buffer example

The decomposition type of each feature in the feature diagram will be an *and*. This means that, in the resulting feature diagram, all children are mandatory with respect to their parent. In order to represent the optionality of a child with respect to its parent in the feature diagram, a dummy feature node is inserted between the parent and the optional child. That dummy node is mandatory to its parent and has itself a decomposition type of $\langle 0..1 \rangle$ for its son: the optional child from the change specification. Figure 6.3 presents the change specification of the `Mul.Res.` feature (on the left) and shows how a dummy node is inserted between the optional $M6$ node and its parent feature `Mul.Res.` (in the middle) in order to represent the feature diagram of the `Mul.Res.` (on the right). Note that other decompositions types like *xor* and *or* (Table 6.1 on page 120) are not used in change specifications.

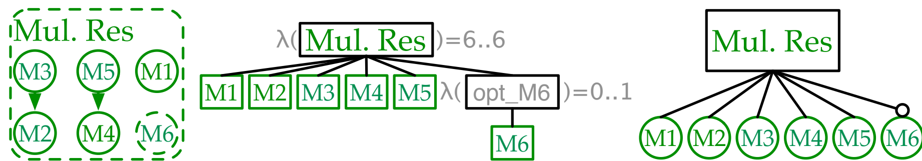

Figure 6.3: Mapping optional changes to optional features

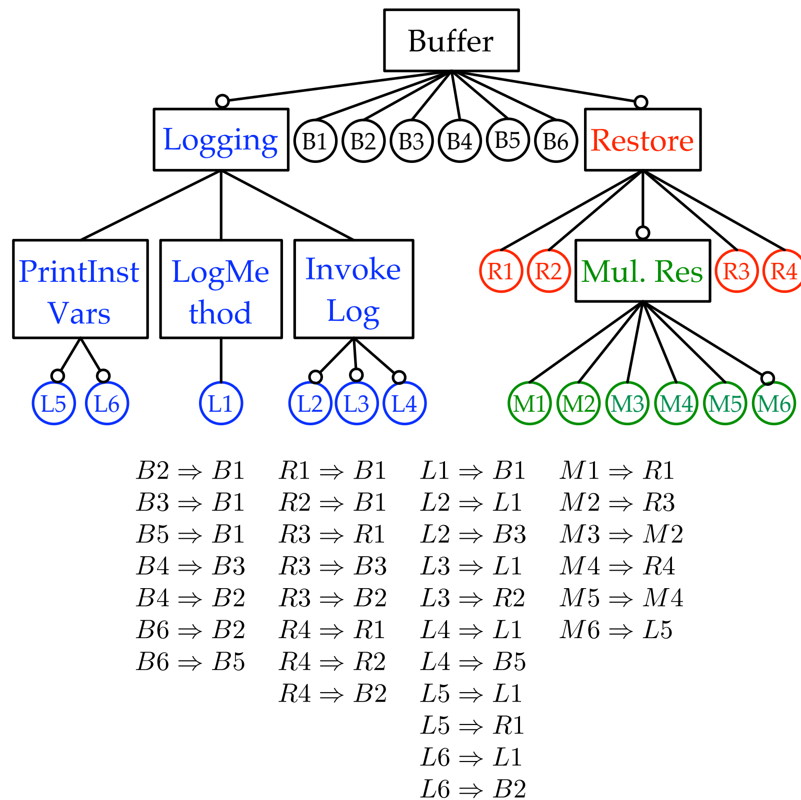

Figure 6.4: Tentative feature diagram representing the buffer change specification.

The changes with their dependencies, however, also need to be represented as part of the feature diagram. The most obvious way to do that is to see every change object as a feature without children, and therefore to consider them as the leaf features of the feature diagram. The dependencies between changes, however, crosscut the hierarchy and can therefore not be represented this way. Instead, we capture them by additional feature diagram constraints (the logic formulae of Φ). Finally, features and changes can be optional with respect to their parent. This immediately translates into optional features.

The result of this translation is the diagram of Figure 6.4. Intuitively, its 'meaning' is the same as that of the change specification in Figure 6.2 because it preserves all of its constraints, meaning that if a set of features (consisting of normal ones and those representing changes) satisfies the feature diagram, it is also a legal change composition/feature set.

This property, however, needs to be established formally, not only for the one example here, but for the algorithm that does the translation. Similarly, we need to formally specify the properties which we want to capture when analysing the feature diagram, and make clear how they translate back to ChOP. This is the

goal of the next two sections.

The particularity of a feature diagram obtained from a change specification is that it contains, unlike most feature diagrams obtained from analysts, implementation details that were recorded as the code was written. The level of granularity is the statement, which is very fine. In realistic cases, the resulting feature diagram will be enormous, but given the industrial-strength *satisfiability solvers* on which most feature diagram implementations are based, this should not be a problem [36]. On the contrary, it will be an opportunity and allow for a number of interesting analysis properties.

6.3.1 Translating the formalism

A general procedure for translating a change specification into a feature diagram is given by Algorithm 8. As can be seen in the `Mapping root features` part, one thing that the previous example did not account for is the fact that a change specification does not necessarily have a root. A feature diagram, however, needs to have one, which is why the algorithm starts by creating an artificial root r, and making each of the top level features an optional child of that root.

The result from applying this algorithm to the change specification of Figure 6.2 is presented in Figure 6.5. It is more complex but semantically equivalent to the feature diagram of Figure 6.4. Actually, the algorithm makes abundant use of dummy features, not only for the root, but also to express optionality. These dummy features are, however, not primitive, and will not appear in the semantics of the resulting feature diagram. Note that dummy features only appear in the translation; they are thus 'hidden' from the user.

The following theorem formalises an important property of this algorithm, which we intuitively referred to as the resulting feature diagram having the *same meaning* as the original change specification. More formally, the algorithm *preserves the semantics* of the change specification.

Theorem 15 (Correctness of Algorithm 8)**.** *Let* cs2fd *denote the translation function described by algorithm 8. Then for each change specification Cs,*

$$flatten([\![Cs]\!]) = [\![cs2fd(Cs)]\!]$$

where

$$flatten(Set) = \{a \cup b | (a, b) \in Set\}.$$

The set returned by $[\![Cs]\!]$ consists of couples (H, G) such that H is a legal change composition for G and that G is a legal feature set for H with respect to Cs. Consider for instance calling $flatten$ on $((H_1, G_1), (H_2, G_2), (H_3, G_3))$. This returns the set $((H_1 \cup G_1), (H_2 \cup G_2), (H_3 \cup G_3))$. Each element of this set consists of features and the corresponding changes, which is the same as the semantics of the result of the feature diagram returned by Algorithm 8. The proof is straightforward and was omitted from this text.

Input: A change specification $Cs = (C, F, Sub, F4C, D)$
Output: a feature diagram $d = (N,\ P,\ r,\ \lambda,\ DE,\ \Phi)$

```
% Initialisations
```

$r \leftarrow$ a new fresh node;
$P \leftarrow C \cup F$;
$N \leftarrow P \cup \{r\}$;
$(\lambda, DE, \Phi) \leftarrow (\emptyset, \emptyset, \emptyset)$;

```
% Mapping root features
```

Let $roots \leftarrow \{f \mid f \in F \wedge \neg\exists f' \cdot f' \xrightarrow{?} f\}$;
Let $i \leftarrow 0$;
foreach $n \in roots$ **do**
  $f \leftarrow$ a new fresh node;
  $N \leftarrow N \cup \{f\}$;
  $\lambda \leftarrow \lambda \cup \{f \mapsto card_1[0..1]\}$;
  $DE \leftarrow DE \cup \{(r, f), (f, n)\}$;
  $i \leftarrow i + 1$;
end
$\lambda \leftarrow \lambda \cup \{r \mapsto card_i[i..i]\}$;

```
%Mapping non-root features & changes
```

foreach $f \in F$ **do**
  $i \leftarrow 0$;
  foreach $n \in \{f' \mid (f, f', x) \in Sub\} \cup \{c \mid F4C(c) = (f, x)\}$ **do**
    if x=man **then**
      $DE \leftarrow DE \cup \{(f, n)\}$;
    end
    else
      Let $z \leftarrow$ a new fresh node;
      $N \leftarrow N \cup \{z\}$;
      $\lambda \leftarrow \lambda \cup \{z \mapsto card_1[0..1]\}$;
      $DE \leftarrow DE \cup \{(f, z), (z, n)\}$;
    end
    $\lambda \leftarrow \lambda \cup \{f \mapsto card_i[i..i]\}$;
    $i \leftarrow i + 1$;
  end
end

```
% Mapping change dependencies
```

foreach $(c, c') \in D$ **do**
  $\Phi \leftarrow \Phi \cup \{\text{“}c \Rightarrow c'\text{”}\}$;
end

Algorithm 8: Transforming a Cs to a feature diagram

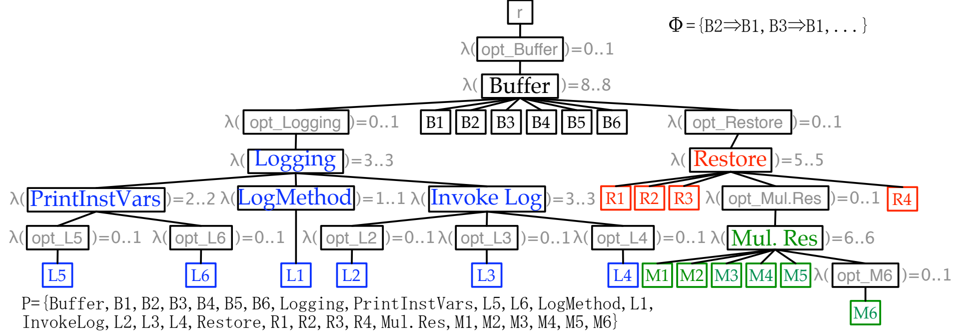

Figure 6.5: Buffer feature diagram resulting from the translation algorithm

6.3.2 Applications

Given Algorithm 8 and Theorem 15, it is possible to translate a change specification Cs into a feature diagram d whose legal products are exactly the legal change composition/feature set pairs of the change specification. This means that we can analyse d instead of Cs but interpret the results in terms of Cs. Here we present several analysis methods for feature diagrams and show how they can be useful in the context of ChOP.

A first application of feature diagrams was already hinted at in the previous sections. Indeed, given a change composition/feature set pair, it is legal iff it is a product of the feature diagram. Which means that we can use feature diagram tools to check the validity of change composition/feature set pairs. Formally, given a change specification Cs, $H \subseteq C$ and $G \subseteq F$, then

$$(H, G) \text{ is legal iff } (H \cup G) \in [\![cs2fd(Cs)]\!].$$

Furthermore, we can use the feature diagram to determine the feature compositions that are legal with respect to a change composition, i.e.

$$fcomp(H) = \{P \cap F | P \in [\![cs2fd(Cs)]\!] \wedge H = P \cap C\},$$

where $H \subseteq C$, or the other way around, with $G \subseteq F$

$$ccomp(G) = \{P \cap C | P \in [\![cs2fd(Cs)]\!] \wedge G = P \cap F\}.$$

If $fcomp(H)$ or $ccomp(G)$ return an empty set, we know that H is an illegal change composition, respectively G an illegal feature set. Otherwise, the results of $fcomp(H)$ (resp. $ccomp(G)$) can easily be used to determine the minimal/maximal feature set (resp. change composition), it suffices to take the element of $fcomp(H)$ (resp. $ccomp(G)$) with minimal/maximal cardinality.

A common analysis means for feature diagrams are metrics defined on the feature diagram [11]. For instance, as feature diagrams are generally used to express

the variability of a software product line, the number of valid feature combinations of a feature diagram $|[\![d]\!]|$ is a measure for the variability of the software product line. If the feature diagram was obtained from a change specification, it measures the variability of the change specification. This kind of metric is already implemented in feature diagram tools such as FAMA [11]. Obtaining the same information based on only the data in Figure 6.2 would require a new algorithm and would consequently imply more work.

A similar metric would be to determine in how many ways a feature set $G \subseteq F$ can be implemented, and how many changes are needed (at most/least) to implement it. This can be done by calculating $fcomp(H)$ and determining the minimal/maximal cardinality of its elements. A configuration tool, i.e. a tool that lets a user choose which features to include, could then show this kind of statistics while performing the choices.

If additional information about changes is available, such as the lines of code added or additional memory consumption, it can be added to the feature diagram in the form of feature attributes. The configuration tool could then show more comprehensive statistics about the user's current feature selection. Instead of seeing merely how many changes it will require, the user will be able to see to what extend the resulting application will grow in code size or memory consumption. Instead of showing metrics to the user, the configuration tool could also choose the change composition itself, for instance by selecting the one that is optimal with respect to an objective function (e.g. minimise memory consumption) [11]. The advantage here is that the attribute values would be automatically derived from the actual changes in the code without the need for human intervention.

Another application of feature diagrams is determining which features are always present (called *commonality*),

$$comm(d) = \bigcap_{G \in [\![d]\!]} G,$$

and which are never present (called *dead features*),

$$dead(d) = P \setminus \bigcup_{G \in [\![d]\!]} G.$$

If we project these results to the sets of changes $comm(cs2fd(Cs)) \cap C$ and $dead(cs2fd(Cs)) \cap C$, we obtain the set of changes which are always/never present in the system. With the current model of ChOP, however, these indicators are not very useful. As shown in the proof of Theorem 7, the change composition/feature set consisting of all changes and features is always legal, hence $dead(cs2fd(Cs)) = \emptyset$. Furthermore, through Algorithm 8, each top-level feature of the change composition becomes optional, hence $comm(cs2fd(Cs)) = \emptyset$. A more subtle and relevant approach would be to consider the number of occurrences of a change or feature among the set of legal compositions. A change with a high frequency ('almost common', one could say) could suggest a refactoring to make the corresponding code efficient, whereas no effort should be put in code that is

'almost dead'. Formally, a tool should indicate, given a change $c \in C$, whether

$$\frac{||\{p \in [\![cs2fd(Cs)]\!] \mid c \in p\}||}{||[\![cs2fd(Cs)]\!]||} \quad > \quad k_1 \quad (resp. < k_2)$$

where k_1 and k_2 are some fixed thresholds.

6.4 Conclusion

In Chapter 5 we propose an approach to feature-oriented programming (FOP). In this approach, changes that encapsulate any developer action (including the removing of code) are first instantiated. Afterwards, changes can be combined to form a product. Before that is done, it has to be ensured that the selected changes actually *can* be composed.

In this chapter, we make an effort to verify the validity of change compositions. In order to do that, we first propose a formal model of change-oriented programming and afterwards map it to the well-understood notion of feature diagrams (feature diagrams), which has become one of the standard modelling languages for variability in Software Product Line Engineering. In order to take those steps, we first provide a formal description of feature diagrams.

We propose a formal model of change-oriented programming which is based on set theory. Properties including the legality of a change and feature composition are explained subsequently. A composition is valid if the following five properties hold: if a feature is selected, its parent feature must be selected too, if a feature with mandatory sub-features is selected the latter need to be selected too, all changes that are mandatory with respect to the selected features are in the composition, all changes stem from selected features and all dependencies are satisfied. Afterwards, we prove that at least one legal change composition exists and that there is always one maximal and one minimal legal change composition for a given change specification and feature set. Correspondingly, we prove that at least one legal feature composition exists and that there is always one maximal and one minimal legal feature composition for a given change specification and change set.

We then map the formal model of ChOP to the well-understood notion of feature diagrams. The mapping, provided in form of a translation algorithm, allows us to reuse a number of results in feature diagram research and apply them to ChOP. The first application consists of the validation of compositions. A composition is legal if it is a product of the feature diagram. The second application consists of the calculation of all valid compositions for one change specification. The final application contains some metrics, which we can easily measure on feature diagrams and which provide some information about so called common (resp. dead) features, which must always (resp. never) included in legal compositions.

The particularity of a feature diagram obtained from a change specification is that it contains, unlike most feature diagrams obtained from analysts, implementation details that were recorded as the code was written. The level of granularity is the statement, which is very fine. This huge amount of information allows for

the calculation of a number of interesting analysis properties, such as the amount of changes required to construct a certain software variation. Instead of just presenting this metric to the developer, we envision a tool that returns the optimal change composition, for instance by selecting the one that is optimal with respect to an objective function (e.g. minimise memory consumption). The advantage of a feature diagram obtained from a change specification is that the attribute values would be automatically derivable from the actual changes in the code without the need for human intervention.

Chapter 7

Expressing crosscutting concerns

Some functionality of system implementation, such as logging or error handling are notoriously difficult to implement in a modular way. The result is that code implementing such functionality is scattered over and tangled across a system. This leads to quality, productivity and maintenance problems. A functionality is said to be *crosscutting* if it cannot be cleanly separated into a separate module because it is affecting several other modules [37]. According to our model, a crosscutting functionality is represented by a feature that includes changes which modify software building blocks scattered over the system.

In this chapter, we first explain how crosscutting functionality is modularised in feature-oriented programming (FOP) and show our change-based approach to FOP already supports such modularisation. We introduce the concept of *flexible* and *monolithic* features, their semantics and how they can be composed to form different program variations. Afterwards, we expose a weakness of our approach which is a consequence from features holding *extensional* sets of changes. We elaborate on a solution for that weakness which includes the extension of the change model with a new change kind: An *intensional* change is a descriptive change that can evaluate to an extension of changes.

7.1 Crosscutting functionality in feature-oriented programming

In our model, crosscutting functionality is represented by a feature that includes changes which modify software building blocks scattered over the system. The application of such a feature affects (creates, modifies or removes) building blocks scattered over the software system. Take, for instance, the logging functionality of the buffer application that was introduced in Chapter 5. That functionality is implemented by the `Logging` feature. It contains changes which introduce code

statements scattered around every method in the `Buffer` class. Consequently, the `Logging` feature depends on more than one feature (Figure 5.3).

A positive property of our approach is that the implementation of crosscutting functionality is actually not scattered over the software application. All the changes of a feature that implements a crosscutting functionality are grouped in one change set. It is the application of the changes that actually produces a software system containing the scattered building blocks of the crosscutting functionality.

A property of features that implement a crosscutting functionality is that their changes usually depend on more than one feature. As we have seen in the previous chapter, a change-based software composition is only valid if all the changes on which the changes of the composition depend are also included in the composition. Consequently, the inclusion of a crosscutting functionality in a composition requires all the features that contain changes on which the changes of the crosscutting functionality depend to be included in the composition. As that is not always desired, this boils down to a composition conflict. A quick and dirty solution would be to provide a feature variation for each combination of the features on which it depends. This kind of solution would suffer from a combinatorial explosion, increase the coupling between features and decrease reusability.

7.1.1 Flexible features

We advocate another solution which allows features to be partially deployed in a composition. A feature that implements a crosscutting functionality can be implemented as a set of changes that does not have to be applied as a whole for a composition to be valid. In contrast to a feature that has to be applied as a whole (*monolithic*) we call features that can be partially applied *flexible* features. In order to be flexible, a feature is required to have at least one optional sub-entity (which can be a feature or a change). Figure 5.10 shows that according this definition, `PrintInstVars`, `InvokeLog` and `MultipleRestore` are all flexible features. When such a flexible feature is included in a composition, the composition algorithm (like the one presented in Algorithm 1 on page 115) decides which of the optional changes are included in, and which are omitted from, the composition.

In order to be flexible, a feature has to include at least one optional subentity. Such a decision depends on domain knowledge and must be taken by the developer. If a developer classifies a change or feature as optional with respect to its parent, the parent automatically becomes flexible. Note that composing a feature that was erroneously specified as flexible can yield a program that actually does not contain the functionality implemented by that flexible feature. For example, consider a flexible feature that contains optional changes only. If all these changes are omitted from the composition, the feature is actually not included anymore in the composition. Consequently, programmers should understand the responsibility that comes with the declaration of flexible features.

In Figure 7.1, one can see that `Logging` actually consists of three features, which all have to be included in a composition that contains `Logging`. The monolithic `LogMethod` feature consists of a mandatory change that adds a `logit()`

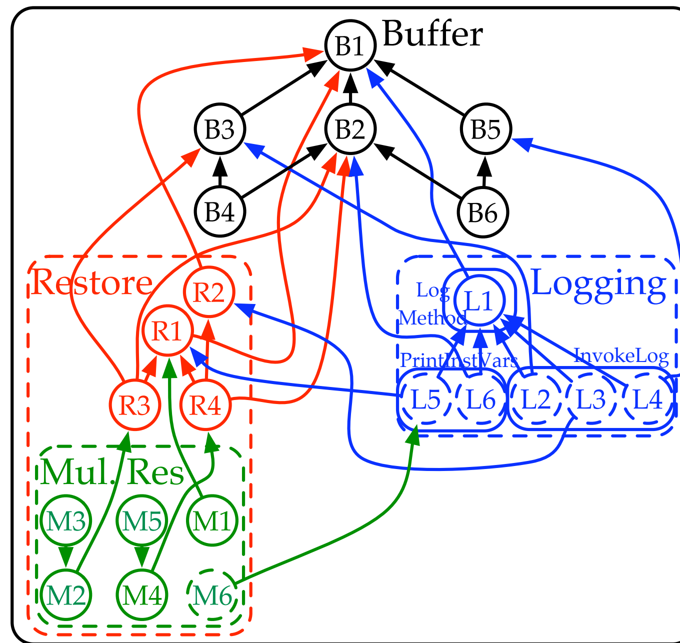

Figure 7.1: Change specification of the buffer example (copy of Figure 5.10)

method to the `Buffer` class. The flexible `PrintInstVars` feature consists of two optional changes, which each add a statement printing the value of an instance variable of `Buffer`. Finally, the flexible `InvokeLog` feature contains three optional changes – which each add a statement that invokes the `logit()` method – to a method of `Buffer`. We argue that in case a flexible feature like `InvokeLog` is included in a composition, its `AddInvocation` changes that have to be added to methods that do not exist because their creational changes are not part of the composition, should be omitted from the composition in order to make it valid. Let us now demonstrate the flexibility brought to the composition of features by flexible features.

7.1.2 Composing flexible features

A composition is specified by listing all the features that must be included. No distinction is made between flexible and monolithic features. It is in fact the composition algorithm that decides which parts of the flexible features are included and which are not. Different composition strategies are conceivable. In Chapter 6, we present the definition of a minimal and maximal change composition and provide an algorithm for obtaining the latter.

The strategy for obtaining the maximal change composition makes sense, since it produces the system with the most complete implementation of the correspond-

ing feature set. On the other hand, a strategy for obtaining the minimal change composition can also be interesting in the case where code size needs to be minimised, as it returns the most basic implementation of a feature set. Yet another strategy could be a mix of both in which the optional changes that add code are included and the optional changes that remove code are omitted.

The structural dependencies are enforced by the meta-model of the asserted programming language and are used by the composition strategy to ensure composition validity. The fact whether a feature is monolithic or flexible does not affect the structural dependencies between the change objects. This means that, if a change of a flexible feature is omitted from a composition, all of the changes that depend on it are also omitted from that composition. As long as these changes are optional, the composition will remain valid. In the case one of these changes is mandatory the composition is invalid. For more details on this composition strategy, we refer the reader back to Algorithm 1 on page 115.

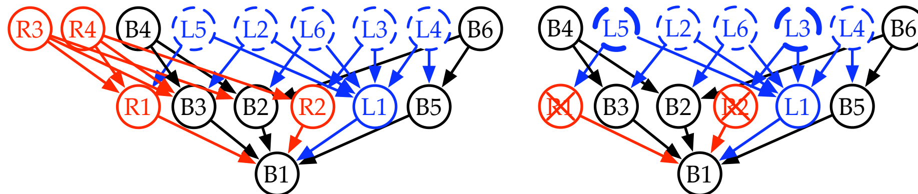

Figure 7.2: Compositions based on first-class changes

Figure 7.2 shows an example of a valid maximal composition of *Base*, `Restore` and `Logging` (on the left) and *Base* and `Logging` (on the right). Since in the right composition the `Restore` feature is not present, the changes $L5$ and $L3$ (which respectively belong to the flexible `InvokeLog` and `PrintInstVars` features) can be safely omitted from the composition as they are optional. This process produces a legal composition consisting of $B1$, $B2$, $B3$, $B4$, $B5$, $B6$, $L1$, $L2$, $L4$ and $L6$.

This example shows that the flexible `InvokeLog` and `PrintInstVars` can be included completely or partially, depending on which other features are included in the composition. The decision to include an optional change is specified by the composition strategy and does not require an additional manual effort. While monolithic features can only be included in a composition together with all the features they depend on, flexible features can be *automatically customised* by means of a composition strategy in order to include them in a valid composition that does not necessarily contains all the features they depend on. This brings the flexibility to cope with features that implement a crosscutting functionality.

7.1.3 Other uses for flexible features

The applicability of *flexible* features is not only confined to describing crosscutting functionality. For instance, a feature that implements a *facade* pattern [42],

would add a class and a method for each complex service. It can be conveniently described by a *flexible* feature, allowing for it to be composed with a set of features which not necessarily include all the services that the *facade* class references. In a composition, the *facade* class will only provide the methods for which functionality is indeed present in the composition.

While flexible features already help in modeling crosscutting functionality, they still suffer from some inconveniences. As we have already stated, it is up to the developer to specify which changes or features are optional and which are not. A second drawback of flexible features is that they only contain an extensional list of changes which leads to issues maintaining such features. In the following section, we elaborate on this issue.

7.2 Extensional changes

In case a feature, which represents a crosscutting functionality, needs to be adapted, changes representing that adaptation have to be added to the change set. This process is actually quite cumbersome, as the programmer is required to instantiate the changes that specify the adaptation. In order to clarify this problem, we first extend the `Buffer` example with an extra feature, which is responsible for maintaining statistics on how many times the buffer is used. Figure 7.3 contains the code of the new buffer software system (on the left hand side) and the change objects that were instantiated in order to obtain that code (on the right hand side).

The inclusion of this feature in the buffer application introduces two new instance variables `setc` and `getc` which are used to maintain the amount of times the `set()` and `get()` methods have been invoked. It also adds a new method `statistics()` to the `Buffer` class that prints the values of these instance variables. The introduction of the `Statistics` feature requires an adaptation of the logging feature, which is required to print the values of *all* instance variables whenever *any* method of the `Buffer` class is executed. Consequently, the `Logging` feature should be extended with changes that add the printing of the new instance variables in the `logit()` method and with a change that adds an invocation of the `logit()` method in the `statistics()` method. Figure 7.4 presents the source code of the adapted `Buffer` case.

The modification of the `Logging` feature involves changes that are scattered over the application. In case a logging or differentiating technique is used to capture the changes (see Chapter 5), the developer has to manually recover all the locations in the source code where changes need to be made. In case change-oriented programming is used to capture changes, the developer has to filter the increment in the change set in order to recover what changes need to be added. In this example, for every `AddMethod` change, an invocation to the `logit()` method should be added and for every `AddAttribute` change, a print statement that prints that attribute should be added to the `logit()` method. The changes that correspond to the modification are presented in Figure 7.5.

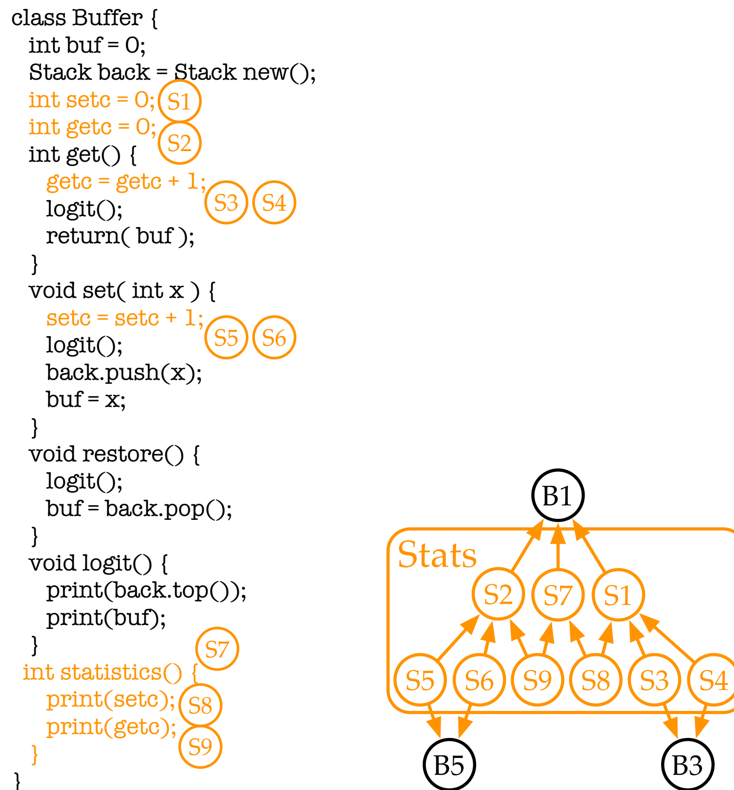

Figure 7.3: A buffer with a functionality for maintaining statistics

In order to overcome this tedious task, we propose a new kind of changes – *intensional changes*. The following section elaborates on this novel kind of change.

7.3 Intensional changes

Our model defines a feature as a set of changes. There are two ways of specifying such a set: *extensionally* or *intensionally*. A set can be defined extensionally by explicitly enumerating all elements of the set. For example, we can define the set E of all even numbers extensionally as follows:

$$E = \{0, 2, 4, 6, 8, ...\}$$

However, the same set can also be specified intensionally by means of a description:

$$E = \{2x \mid x \in N\}$$

The same applies for sets of changes. The `Logging` feature can be specified by an extensional set of changes $F_{Logging}$ which is a union of all changes

```
class Buffer {
  int buf = 0;
  Stack back = Stack new();
  int setc = 0;   (S1)
  int getc = 0;   (S2)
  int get() {
    getc = getc + 1;   (S3) (S4)
    logit();
    return( buf );
  }
  void set( int x ) {
    setc = setc + 1;   (S5) (S6)
    logit();
    back.push(x);
    buf = x;
  }
  void restore() {
    logit();
    buf = back.pop();
  }
  void logit() {   (L7)
    print(setc);
    print(getc);   (L8)
    print(back.top());
    print(buf);
  }   (S7)
  int statistics() {
(L9)  logit();
    print(setc);   (S8)
    print(getc);   (S9)
  }
}
```

Figure 7.4: Extended buffer code after adding the statistics feature

of $LogMethod$, `PrintInstVars` and `InvokeLog` and which can be applied on any variation of `Buffer` in order to add the logging functionality:

$$F_{Logging} = \{L1, L2, L3, L4, L5, L6, L7, L8, L9\}$$

An intensional description of the same `Logging` feature that is added to a program p is:

$$F_{Logging} = \{\text{all changes needed to add a logging functionality to p}\}$$

This definition can be refined as:

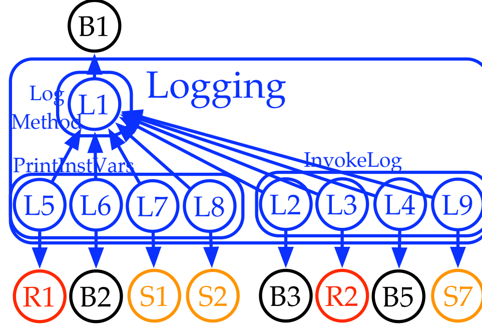

Figure 7.5: Extended logging changes after adding the statistic code

$$F_{Logging} = \left(\begin{array}{l} \{\text{changes of} LogMethod\} \cup \\ \{\text{changes of} PrintInstVars\} \cup \\ \{\text{changes of} InvokeLog\} \end{array} \right)$$

$$= \left(\begin{array}{l} \{\textbf{AddMethod}((Buffer, false, logit()), 1235465, \text{“}Peter\text{”}, \text{“}Logging\text{”})\} \cup \\ \{\text{changes of} PrintInstVars\} \cup \\ \{\text{changes of} InvokeLog\} \end{array} \right)$$

$$= \left(\begin{array}{l} \{\textbf{AddMethod}((Buffer, false, logit()), 1235465, \text{“}Peter\text{”}, \text{“}Logging\text{”})\} \cup \\ \{\text{add statement print}(var) \text{ in } logit \text{ for every instance variable } var \text{ in p}\} \cup \\ \{\text{changes of} InvokeLog\} \end{array} \right)$$

$$= \left(\begin{array}{l} \{\textbf{AddMethod}((Buffer, false, logit()), 1235465, \text{“}Peter\text{”}, \text{“}Logging\text{”})\} \cup \\ \{\text{add statement print}(var) \text{ in } logit \text{ for every instance variable } var \text{ in p}\} \cup \\ \{\text{add an invocation to } logit \text{ in every method of } \texttt{Buffer}\} \end{array} \right)$$

In order to be able to express a change set C in an intensional way, two things are required: a *change-cut language* and a *change-cut model*. A *change-cut language* is a language that can be used to specify change-cuts: a quantification or query that evaluates to an extensive set of changes. A *change-cut model* is a model that defines a change-cut language and a means of specifying new changes that have to be instantiated for a change cut. The following two subsections subsequently describe the language and the model.

7.3.1 Language for specifying intensional changes

When an intensional change c needs to be applied to a program p in order to add to p the functionality c implements, the specification of c has to be evaluated to find out the exact changes that are required in the specific situation (the enumeration of changes described by c). In the above examples, we used a natural language to describe the intensional specification of a change. It is clear that, in order to automate the evaluation process, a more formal language is needed. In this section we establish such a language based on a tuple calculus. We start by explaining that the building blocks of our approach consist of tuples of changes.

Building blocks

The population of a change set is represented by means of n-tuples. Each $n-tuple$ consists of n artifacts which belong together. For instance, a number of examples of 2-tuples are:

$$(\text{“}B1\text{”}, 112)$$

$$(\text{“}B2\text{”}, 123)$$

$$(\text{“}B3\text{”}, 145)$$

These tuples represent for example an association of the id of a change with the time stamp of that change. In classic set-theory, the order in which the artifacts appear in a tuple is important. For instance, the tuple $(\text{“}B1\text{”}, 112)$ is different from the tuple (112, “$B1$”). In order to improve readability of the tuples, we opt to use *named* n-tuples which contain tags mapping a certain attribute of a tuple to a value, similar to the way the population of databases are often described. In this notation, we can express our example 2-tuples as follows:

$$(\textbf{id:}\text{“}B1'', \textbf{timestamp:}112)$$

$$(\textbf{id:}\text{“}B2'', \textbf{timestamp:}123)$$

$$(\textbf{id:}\text{“}B3'', \textbf{timestamp:}145)$$

where *id* and *timestamp* are the *attributes* of the tuples and $B1$, $B2$, $B3$, 112, 123 and 145 are the *values* of those tuples. Using this notation, the order of the artifacts in the tuples is no longer important. For instance, the tuples $(\textbf{id:}B1, \textbf{timestamp:}112)$ and $(\textbf{timestamp:}112, \textbf{id:}B1)$ are considered to be identical.

We define C_p to be the set of all changes that produce a program p whenever they are applied. The changes in this set are 6-tuples that follow the following pattern:

$$(id, type, parameterList, timestamp, user, intent)$$

where *id* is the attribute specifying the unique identifier of the change object, *type* is the attribute denoting the kind of the change, *parameterList* is the attribute containing a list of parameters that can be used to apply the change, *timestamp* is the attribute that specifies the time at which this change is instantiated, *user* is the attribute that contains the information of which user instantiated this change and *intent* is the attribute that contains a description of the intent of this change.

Atoms

We assume a finite set C_p of tuples and an infinite set of tuple attributes $attrib(C_p)$, for the construction of formulas on the set of tuples. $Changetypes$ is the set of all different types of changes (as described in Section 4.3). We then define the set of atomic formulas $A[C_p]$with the following rules:

1. if c_1 and c_2 in C_p, a and b in $attrib(C_p)$ then the formula "$c_1.a = c_2.b$" is in $A[C_p]$,
2. if c in C_p, a in $attrib(C_p)$ and k denotes a value then the formula "$c.a = k$" is in $A[C_p]$, and
3. if c in C_p and r in $Changetypes$ then the formula "$r(c)$" is in $A[C_p]$.

Examples of atoms include:

$$(c_1.user = c_2.user)$$

$$(c_1.user = \text{``}Peter'')$$

$$AddMethod(c)$$

The first example means that tuple c_1 has a "user" attribute and c_2 has a "user" attribute with the same value. The second example means that tuple c_1 has a "user" attribute and its value is "Peter". The last example means that tuple c is of the $AddMethod$ type. The formal semantics of such atoms is defined given a change set C_p and a tuple attribute binding $val : C_p \times attrib(C_p) \rightarrow Object$ that maps tuple attributes to tuple values over the domain in C_p:

1. "$c_1.a = c_2.b$" is true if and only if $val(c_1, a) = val(c_2, b)$
2. "$c.a = k$" is true if and only if $val(c, a) = k$
3. "$r(c)$" is true if and only if the type atrribute of c is r.

The atomic formulas denote a condition for a change object. The next step in defining the change cut language is the construction of more complex formulas which consist of *composed* and *quantified* atomic formulas as we explain below.

Formulas

In our change cut language, the atomic formulas defined above can be combined into formulas, with the logical operators $\wedge$ (and), $\vee$ (or) and $\neg$ (not), and we can use the existential quantifier ($\exists$) and the universal quantifier ($\forall$) to quantify variables or relations. We define the complete set of formulas $F[C_p]$ inductively with the following rules:

1. every atom in $A[C_p]$ is also in $F[C_p]$,
2. if f_1 and f_2 are in $F[C_p]$ then the formula "$f_1 \wedge f_2$" is also in $F[C_p]$,
3. if f_1 and f_2 are in $F[C_p]$ then the formula "$f_1 \vee f_2$" is also in $F[C_p]$,
4. if f is in $F[C_p]$ then the formula "$\neg f$" is also in $F[C_p]$,
5. if c in C_p and f a formula in $F[C_p]$ then the formula "$\exists c : f$" is also in $F[C_p]$,
6. if c in C_p and f a formula in $F[C_p]$ then the formula "$\exists! c : f$" is also in $F[C_p]$,

7. if c in C_p and f a formula in $F[C_p]$ then the formula "$\nexists c : f$" is also in $F[C_p]$, and
8. if c in C_p and f a formula in $F[C_p]$ then the formula "$\forall c : f$" is also in $F[C_p]$.

Examples of formulas are:

$$(c.user = \text{"}Peter\text{"} \lor c.user = \text{"}Sabine\text{"})$$

$$(AddMethod(c) \land c.user = \text{"}Peter\text{"}$$

$$\forall c : (AddMethod(c) \land c.user = \text{"}Peter\text{"} \land p.intent = \text{"}Buffer\text{"})$$

The first example binds c to all changes of C_p instantiated by Peter or Sabine. The second example binds c to all the changes of C_p that are of the $AddMethod$ type that are instantiated by Peter. The final example returns true if all changes of C_p are of the $AddMethod$ type, instantiated by Peter in an intent to create a buffer. Note that we omit brackets if this does not cause ambiguity about the semantics of the formula. We assume that the quantifiers quantify over the set of all tuples of the change set C_p. This leads to the following formal semantics for formulas:

1. see the formal semantics of the atomic formulas,
2. "$f_1 \land f_2$ " is true if and only if "f_1" is true and "f_2" is true,
3. "$f_1 \lor f_2$" is true if and only if "f_1" is true or "f_2" is true or both are true,
4. "$\neg f$" is true if and only if "f" is not true,
5. "$\exists c : f$" is true if and only if there is a tuple c in C_p for which the formula "f" is true,
6. "$\exists! c : f$" is true if and only if there is exactly one tuple c in C_p for which the formula "f" is true,
7. "$\nexists c : f$" is true if and only if there is no tuple c in C_p for which the formula "f" is true, and
8. "$\forall c : f$" is true if and only if for all tuples c in C_p the formula "f" is true.

Formulas denote the condition for a change cut and consist of compositions and/or quantifications of formulas. Let us now explain how a change cut can actually be defined by means of formulas.

Change cuts

Finally, we define a change cut expression (in short a change cut) for a given change set C_p as:

$$\{c : f(c)\}$$

where c is a tuple in C_p and f(c) a formula in $F[C_p]$. The result of such a query for a given change set C_p that specifies a program p is the set of all tuples c of C_p such that f is true.

Examples of change cut expressions include:

$$\{c : \forall c : (AddMethod(c) \wedge AddClass(d) \wedge c\ parameterList\ class = d\ parameterList\ class)\}$$

$$\{c : \forall c : (AddAttribute(c) \wedge AddClass(d) \wedge c\ parameterList\ class = d\ parameterList\ class)\}$$

which returns all the changes that add a method to a class and all changes that add an attribute to a class respectively. Now that we are capable of expressing change cuts, all that remains in order to enable expressing intensional changes are the actions that need to be taken for the tuples of a change cut. This is the subject of the following paragraph.

Actions

The purpose of intensional changes is to allow a developer to *describe* which changes have to be added (in an intensional way) instead of *enumerating* them (in an extensional way). As an example, we consider again the `Logging` feature which we want to express as:

$\{$**AddMethod**$((Buffer, false, logit()), 1235465,$ "*Peter*", "*Logging*"$)\}\cup$
$\{$add statement print(var) in *logit* for every instance variable var in $p\}\cup$
$\{$add an invocation to *logit* in every method of `Buffer`$\}$

Now that we have established a formal language for expressing change cuts like "for every instance variable var in p" or "in every method of `Buffer`". We only lack a way of specifying the actions that need to be taken for all the tuples of a change cut (e.g. add statement print(var) in *logit* or add an invocation to *logit*). These actions consist of the addition of a change (or change set) to the change set.

We define an action as the specification of the addition of a new tuple (representing a change instance) to the tuple set C_p. We differentiate between actions that add that tuple just *before* and just *after* a change:

1. c_{new} before c
2. c_{new} after c

where c_{new} is a 6-tuple $(id, type, parameterList, c.timestamp, user, intent)$ in which all the attributes except for the *id* and the *timestamp* must be bound to values. Those values can be the value of another attribute of the action, or a constant. The *id* is assigned to the change tuple when the action is evaluated. It is ensured to be a unique identifier. The *timestamp* of the new change tuples is determined by the keyword (*before* or *after*) and by the *timestamp* of the change c. In case the action specifies the new change to come after c, the new *timestamp* is set to a *timestamp* which is just after the one of c. In case the action specifies the new change to come before c, the new *timestamp* is set to a *timestamp* which is just before the one of c. We elaborate on how the uniqueness of the *id* is ensured and how the timestamps are calculated in Section 7.3.3. As an example of an action, we present

$(?id2, AddStatement, (Buffer, false, get, \text{"}logit\text{"}), ?timestamp2, \text{"}Peter\text{"}, \text{"}InvokeLog\text{"})$ `after`
$(B3, AddMethod, (Buffer, false, get), 235935, \text{"}Peter\text{"}, \text{"}InvokeLog\text{"})$

which adds a statement to the *get* method of the `Buffer` class just after the method *get* is added to the `Buffer` class. We conclude this subsection with the definition of an intensional change.

Intensional change definition

An intensional change consists of two parts: a set of actions A and a change cut:

$$\{c : A \mid f(c)\}$$

where the values of all tuples that are specified in the change cut can be used as values for the attributes of the tuples in the actions of A. In order to give an example of intensional changes, we use intensional changes in order to specify the complete `Logging` feature by means of intensional changes:

$$
\begin{array}{l}
\{\mathbf{AddMethod}((Buffer, false, logit()), 1235465, \text{“}Peter\text{”}, \text{“}LogMethod\text{”})\}\cup \\
\{c : \{(?id, AddStatement, (Buffer, false, logit, \text{“}print(c.parameterList.attribute)\text{”}), \\
\quad ?timestamp, \text{“}Peter\text{”}, \text{“}PrintInstVars\text{”}) \text{ after } c\} \mid \\
\quad \{\forall c : (AddAttribute(c) \wedge c.parameterList.class = Buffer)\}\}\cup \\
\{c : \{(?id, AddStatement, (Buffer, false, ?m, \text{“}logit()\text{”}), ?timestamp, \text{“}Peter\text{”}, \text{“}InvokeLog\text{”}) \\
\quad \text{after } c\} \mid \\
\quad \{\forall c : (AddMetod(c) \wedge c.method = ?m \wedge c.parameterList.class = Buffer)\}\}
\end{array}
$$

The two intensional changes included in the definition of the `Logging` feature need to be evaluated with respect to a change set in order to produce the extension they describe. The following subsection elaborates on the evaluation of intensional changes.

7.3.2 Intensional change evaluation

An intensional change is a kind of change that can be applied to a software program p in order to apply the changes described by that intensional change to p. This application consists of two phases. First, the intensional change needs to be evaluated with respect to the change set C_p that specifies p. This step produces an enumeration of changes which are all applied on p in the second step.

As an example, consider a software composition

$$\{Buffer, Restore, MultipleRestore, Statistics, Logging\}$$

which contains the `Buffer`, `Restore`, `Multiple Restore`, `Statistics` and `Logging` features of the buffer example. The application of the `Logging` feature as it is specified by means of intensional changes evaluates to the extension of Listing 7.1 while the application on the same `Logging` feature in a composition

$$\{Buffer, Statistics, Logging\}$$

evaluates to the extension of Listing 7.2. The order in which the features are specified in a composition determines the outcome of the evaluation of intensional changes. This is due to the growing change set on which the intensional changes are evaluated.

Listings 7.1 and 7.2 present the outcome of the first step of the evaluation process. The second step consists of applying the changes that correspond to the tuples from those listings. In order to do that, a change object is instantiated for every tuple. The first tuple, for instance, results in a change object $AddMethod((Buffer, false, logit), 235465, \text{"}Peter\text{"}, \text{"}LogMethod\text{"})$. Note that

```
(L1,AddMethod,(Buffer,false,logit),235465,"Peter","LogMethod")
(L7,AddStatement,(Buffer,false,logit,"print(setc)"),247537.1,
   "Peter","PrintInstVars")
(L8,AddStatement,(Buffer,false,logit,"print(getc)"),247737.1,
   "Peter","PrintInstVars")
(L5,AddStatement,(Buffer,false,logit,"print(back)"),236537.1,
   "Peter","PrintInstVars")
(L6,AddStatement,(Buffer,false,logit,"print(buf)"),235789.1,
   "Peter","PrintInstVars")
(L2,AddStatement,(Buffer,false,get,"logit()"),235935.1,
   "Peter","InvokeLog")
(L3,AddStatement,(Buffer,false,set,"logit()"),237537.1,
   "Peter","InvokeLog")
(L4,AddStatement,(Buffer,false,restore,"logit()"),236137.1,
   "Peter","InvokeLog")
(L9,AddStatement,(Buffer,false,statistics,"logit()"),247937.1,
   "Peter","InvokeLog")
```

Listing 7.1: Extension of `Logging` on $\{Buffer, Restore, Statistics\}$

```
(L1,AddMethod,(Buffer,false,logit),235465,"Peter","LogMethod")
(L5,AddStatement,(Buffer,false,logit,"print(back)"),236537.1,
   "Peter","PrintInstVars")
(L6,AddStatement,(Buffer,false,logit,"print(buf)"),235789.1,
   "Peter","PrintInstVars")
(L2,AddStatement,(Buffer,false,get,"logit()"),235935.1,
   "Peter","InvokeLog")
(L3,AddStatement,(Buffer,false,set,"logit()"),237537.1,
   "Peter","InvokeLog")
(L4,AddStatement,(Buffer,false,restore,"logit()"),236137.1,
   "Peter","InvokeLog")
```

Listing 7.2: Extension of `Logging` on $\{Buffer, Restore\}$

all change objects that result from this process are optional with respect to the feature they belong to.

For the sake of clarity, we specified the `Logging` feature in a very naive way as the specification does not consider the possibility that attributes or methods were modified or removed by other change objects in the change set. Such naive specification could produce change objects with subjects that no longer exist in the software product at the time the change is applied. As the composition algorithm takes into account the structural dependencies between the change objects, this never produces products with invalid structure. The semantics, however, can indeed be influenced by this naive specification. It is up to the developer to alter the intensional change in such a situation. We come back to this issue in Section 7.3.5 but first discuss the implementation of the change cut language and model of intensional changes.

7.3.3 Implementation

First, we extend the change model of Chapter 4 with the notion of intensional changes. Figure 7.6 shows the change model which is extended with the `Intensional Change` class, which is a subclass of `Change` that models the new change kind. This extension shows that the change model we presented in Chapter 4 is easy to extend when new kinds of changes are concerned.

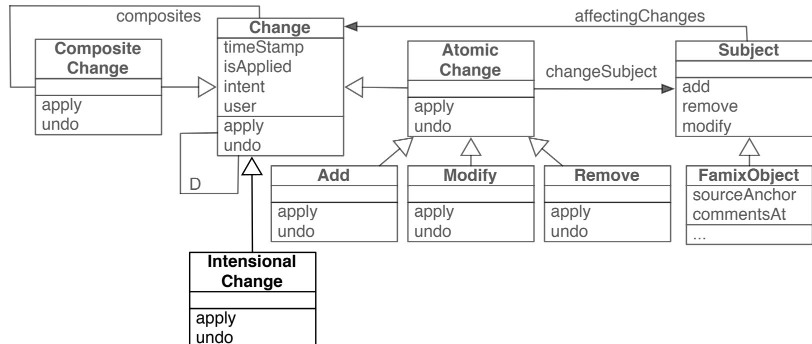

Figure 7.6: Extended change model

Second, we implement the language we defined for specifying intensional changes. For this implementation we use SOUL, an implementation of a Prolog-like declarative language on top of Smalltalk. The strengths of the declarative programming paradigm have already been demonstrated in [111]. Declarative programming allows a developer to minimize or eliminate side effects by describing what the program should accomplish, rather than describing how to go about accomplishing. Another key feature of SOUL is that it can be used to reason about software programs specified in different programming languages. A third key feature of SOUL is that it supports symbiosis with the underlying Smalltalk

or Java environment. The symbiosis allows one to write Smalltalk or Java code within the declarative language which gets executed whenever that declaration is asserted. These features make SOUL an ideal candidate for reasoning about and creating change objects, and thus for specifying intensional changes.

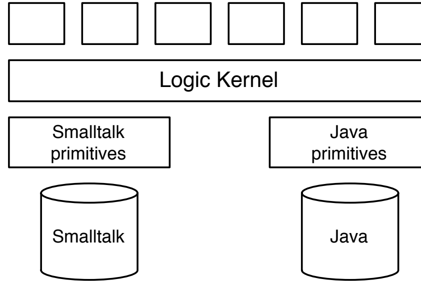

Figure 7.7: SOUL core

The SOUL kernel is depicted in Figure 7.7. It consists of a logic kernel together with an underlying library of basic logic primitives for every programming language it can reason about. Two such libraries exist, namely the Library for Code Reasoning (LiCoR) [72] for reasoning about Smalltalk code and Irish [38] for reasoning about Java code. The logic kernel itself consists of a library of predicates that serve for basic reasoning. They can, for instance, be used to do parse tree traversal. The predicates in that library can be used by the developers for creating new predicates in SOUL. Predicates can be grouped in layers, for the purpose of classification. We create a basic layer, which contains predicates for every change kind.

$$change(?c) \text{ if } member(?c, [ChangeLogger \text{currentComposition changes}])$$

This predicate states that something is a change if it is a member of the collection of changes. Note that this predicate uses the symbiotic properties of SOUL to obtain the collection of changes. We come back to this in Chapter 8 when we discuss the integration of intensional changes into a proof-of-concept-implmentation of our approach.

The evaluation of the query $change(?c)$ makes sure that every change instance is bound once to $?c$. For every kind of attribute of a change, we include a predicate of the shape:

$$user(?u, ?c) \text{ if } change(?c), equals(?u, [\texttt{?c user}])$$

This predicate binds to $?u$ the value of the Smalltalk expression that requests a change $?c$ for its `user` attribute. Now we have explained the building blocks of the language (tuples of change objects), we elaborate on the atoms. An atom is a predicate such as $equals(?u, ?n)$ where $?n$ and ?u can be constants or variables

that must have the same value whenever the predicate is evaluated. Formulas are implemented by chaining atoms. A comma represents the $\wedge$, an alternative definition of a predicate represents the $\vee$, the *not*() represents the $\neg$. As for the $\forall$ and $\exists$ quantifiers, they are implemented by means of standard SOUL predicates $forall(?query, ?test)$ and $exists(?query, ?test)$.

Third, we implement the actions. An action is specified as a SOUL predicate that includes symbiotic Smalltalk code that is executed to instantiate the concerned changes. An example of an action for adding a method before a change $?c$ is presented here:

$$\begin{aligned}&AddMethodBefore(?ParameterList, ?user, ?intent, ?c, ?c_{new}) \text{ if}\\&\quad equals(?c_{new}, [\text{AddChange newMethod: } ?ParameterList\\&\qquad \text{before:}?c \text{ by:}?user \text{ for:}?intent])\end{aligned}$$

A predicate like the one above is defined for each combination of change kind (add, remove or modify), the kind of subject (e.g. class, method, etc) and the time keyword (before or after). All predicates of this kind are grouped in a separate SOUL layer called "actions". The uniqueness of the id of the generated change is ensured by the Smalltalk virtual machine, which assigns a unique object id to every object that is instantiated. As we use the object ID as the change ID, it is also unique. The timestamp is assigned in the same way that we explained in the section on composite changes of Chapter 4.

Finally, we present the implementation of an intensional change and how its evaluation is implemented. An intensional change is specified by a SOUL query that includes a set of predicates (representing the actions) and another set of predicates (representing the change cut). Take for instance the following SOUL query:

$$\begin{aligned}&AddStatementAfter((Buffer, false, ?m, \text{“}logit()\text{”}),\\&\quad \text{“}Peter\text{”}, \text{“}InvokeLog\text{”}, ?c, ?c_{new}),\\&AddMethod(?c),\\&method(?m, ?c),\\&class(\texttt{Buffer}, ?c)\end{aligned}$$

the first predicate of which is an action and the three final predicates of which represent the change cut. This query represents the intensional change of the `InvokeLog` feature. It creates an AddStatement change with a timestamp later than the one of $?c$, for each change $?c$ that adds a method to the `Buffer` class.

Whenever an intensional change instance needs to be applied, its *apply* method needs to be called. That method is implemented by a call to the SOUL engine that executes the query of that change. This causes an enumeration of the described change set to be produced and added to the change set.

7.3.4 Formalising intensional changes

The formalisms from Chapter 6 need to be extended in order to include intensional changes. The fundamental concepts – that consist of the relations Sub, $F4C$, D

between changes and features – remain the same as intensional changes are changes that can depend on multiple other changes (D) and that are contained within one feature ($F4C$). Also the change specification does not require altering in order to include intensional changes.

With respect to the properties of our formal model, we need to modify the definition of a legal change and feature composition. As intensional changes evaluate to an enumeration of changes with respect to a change set, the *actual* change set has to be considered whenever an intensional change is evaluated. Because of that, feature compositions must no longer be specified by a set, but rather by an ordered set – a sequence. On that sequence, we define the

$$before : F \times F \times \mathcal{P}F \rightarrow boolean$$

function, which returns *true* for (f_1, f_2, F) if f_1 is specified on the left side of f_2 in the sequence F. In order to express the evaluation of intensional change, we first introduce a function *eval*:

$$eval : C \times \mathcal{P}C \rightarrow \mathcal{P}C' \tag{7.1}$$

which returns the set of changes that corresponds to the enumeration of changes described by an intensional change c with respect to the change set containing all the changes that stem from features in the feature composition before the feature of c. In case *eval* is called for a change that is not intensional, it is returned unchanged in a singleton set.

Now that we have a definition of the *eval* operator, we can redefine the legal change and feature compositions as:

Definition 16 (Legal change composition, legal feature sequence)**.** *A* legal change composition H *is a change composition such that there exists a* legal feature sequence $G \subseteq F$*, which satisfies the following constraints:*

- *If a feature is selected, all its ancestor features must be selected before that feature in G:*

$$\forall f \in G \bullet g \stackrel{?+}{\rightarrow} f \Rightarrow g \in G \wedge \ before(g, f, G) \tag{7.2}$$

  in which $g \stackrel{?+}{\rightarrow} f$ *is the transitive closure of the parent features of* f*.*

- *If a feature with mandatory sub-features is selected, these need to be selected, as well:*

$$\forall f \in G \bullet f \stackrel{man}{\rightarrow} g \implies g \in G \tag{7.3}$$

- *Let* $M = \{c | f \in G \wedge f \stackrel{man}{\rightarrow} c\}$*, the set of mandatory changes and* $O = \{c | f \in G \wedge f \stackrel{opt}{\rightarrow} c\}$*, the set of optional changes. We need that*
  - *all changes that are mandatory with respect to the selected features are included in the change set:*

$$M \subseteq H \tag{7.4}$$

– *all changes in the change set stem from selected features:*

$$H \setminus M \subseteq O \tag{7.5}$$

- *All dependencies are satisfied*

$$\forall c \in H \bullet \exists (c, c') \in D \implies c' \in H \tag{7.6}$$

- *Let* $I = \{c_i | c_i \in H \wedge c_i$ *is intensional*$\}$ *be the collection of intensional changes of the composition. Let* k_i *be the parent feature of* c_i *that is included in* G*:* $k_i \in G \bullet k_i \xrightarrow{?} c_i$*. Let* $J_{k_i} = \{j | j \in H \wedge before(j, k_i, F)\}$ *be the set of features that are included in* G *before* k_i*. Let* $C_j = \{c | c \in H \wedge \exists j \in J_{k_i} \bullet j \xrightarrow{?} c\}$ *be the set of changes that stem from features in* J_{k_i} *that are included in* H*. We require all dependencies of intensional changes to be satisfied:*

$$\forall c_i \in I \bullet \forall c_e \in eval(c_i, C_j) \bullet \exists (c_e, c') \in D \implies c' \in H \tag{7.7}$$

In comparison to the previous definition of legal change and feature compositions (Definition 5 on page 125) there are two differences: Constraint 7.2 is more strict than 6.14 as it incorporates the order and an extra constraint 7.7 was added to make sure that dependencies of intensional changes are also satisfied.

The other properties of the formal model are not affected by the introduction of intensional changes and are not elaborated on. Only the algorithms that are capable of obtaining the maximal or minimal change composition have to be adapted in order to evaluate the intensional changes when including them in a composition. We present an adapted version of the maximal composition algorithm in Algorithm 9 that uses the subroutines of Algorithms 10, 11, 12, 13 and 14.

Input: A feature set F, a change specification CS
Output: A list consisting of 2 change sets

$F_{min} \leftarrow minimal_feature_set(F, CS)$;
$C_{min} \leftarrow minimal_change_set(F_{min}, CS)$;
$C_{unw} \leftarrow unwanted_change_set(F_{min}, C_{min}, CS)$;
$C^{+}_{unw} \leftarrow transitive_closure(C_{unw}, CS)$;
$C_{err} \leftarrow \{c \in C^{+}_{unw} \setminus C_{unw} \wedge c$ is mandatory$\}$;
return ($C \setminus C^{+}_{unw}$, C_{err})

Algorithm 9: $validateComposition(F, CS)$ function

The only difference between this algorithm and the one presented in Algorithm 1 on page 115 is that the changes that depend on changes resulting from the evaluation of intensional changes need to be retrieved in order to calculate the transitive closure of unwanted changes (Algorithm 14). This is done by Algorithm 12, which first evaluates all changes of the change specification and afterwards computes the complete set of dependencies among that complete set of changes.

```
Input: A feature set F, a change specification CS
Output: A feature set

F_min ← F foreach f ∈ F do
    F_min add: f;
    F_min addall: {g|g ?+→ f};
    F_min addall: {g|f man+→ g};
end
return F_min
```

Algorithm 10: *minimal_feature_set*(F, CS) subroutine

```
Input: A feature set F_min, a change specification CS
Output: A change set

C_min ← ∅;
foreach f ∈ F_min do
    foreach c ∈ f do
        if c is mandatory then
            C_min add: c;
        end
    end
end
return C_min
```

Algorithm 11: *minimal_change_set*(F_{min}, CS) subroutine

```
Input: A change specification CS
Output: A set of dependencies

C_all ← ∅;
D_all ← ∅;
foreach c ∈ C do
    C_all addall: eval(c,C_all)
end
foreach c ∈ C_all do
    D_all addall: c dependencies
end
return D_all
```

Algorithm 12: *find_intensional_dependencies*(CS) subroutine

```
Input: A feature set F_min, a change specification CS
Output: A change set

C_unw ← ∅;
foreach f ∈ F_min do
    foreach c ∈ C do
        if c ∉ C_min and feature of c ∉ F_min then
            C_unw add: c;
        end
    end
end
return C_unw
```

Algorithm 13: *unwanted_change_set*(F_{min}, C_{min}, CS) subroutine

```
Input: A change set C_unw, a change specification CS
Output: A change set C+_unw

S ← C_unw;
D ← find_intensional_dependencies(CS) ;
C+_unw ← ∅;
while S ≠ ∅ do
    c ← remove a change c from S;
    C+_unw add: c;
    foreach c2 with d in D from c2 to c do
        D remove: d;
        S add: c2
    end
end
return C+_unw
```

Algorithm 14: *transitive_closure*(C_{unw}, CS) subroutine

Note that the actual construction of a software variation requires that intensional changes are evaluated before they are carried out. As these algorithms are only used to validate compositions, they do not evaluate intensional changes for constructing variations, but rather only output change sets that can be used to validate and fix erroneous compositions.

7.3.5 Advantages and drawbacks

Now that we have elaborated on how our change model is extended with intensional changes, we evaluate this extension with respect to our approach to feature-oriented programming.

Intensional changes allow for developers to *describe* sets of changes in stead of enumerating them. This is what one wants when implementing a crosscutting functionality. The description of an intensional change is evaluated with respect to a change set in order to produce the corresponding extension of the intensional change. Consequently, an intensional change can be *reused* in different compositions, as it will basically evaluate to the right extension anyway. This makes an intensional change *more flexible* than an ordinary change collection. Consequently, the intensional changes allow our approach to FOP to be more robust against changes in the feature composition.

Drawbacks of the intensional changes are fourfold. First they require an additional change cut language and evaluation step, making them somewhat *more complex* than ordinary changes. A second drawback is that an intensional change cannot be obtained by logging a developer or differentiating between source code files, but must *always be specified by a developer.* Thirdly, intensional changes *complicate debugging*, as they evaluate differently in different compositions. Finally, while the *order* of the features within a composition was not important before the intensional changes were included in the model, it now has become important as it influences the way the intensional change is evaluated. In Chapter 9, we hint at how the second issue could be overcome.

We have implemented both the change cut language and the change cut model by means of SOUL. We integrated both in ChEOPS – our proof-of-concept implementation. Some small experiments were conducted to validate the usability of intensional changes in the context of feature-oriented programming. The following chapter elaborates on both the integration and the experiments.

7.4 Conclusion

We started this chapter by explaining that some functionality, such as logging or error handling, are notoriously difficult to implement in a modular way and that we name such functionality *crosscutting*, because they include changes which modify software building blocks scattered over the system. We show that in our change-based approach to FOP, the implementation of crosscutting functionality is actually not scattered over the software application because the changes are grouped in one change set. An inconvenience of our model, however, is that

features that implement a crosscutting functionality usually depend on more than one feature and that a composition that contains such a feature must consequently include all the features it depends on.

We advocate a solution based on the concept of *flexible* features. A feature is considered flexible if it contains at least one optional change object. That is a change that does not have to be included in a composition in order to make the composition valid. This principle allows a flexible feature to be partially included in a composition that excludes some features on which the flexible feature depends. We show that this brings about more flexibility with respect to feature composition and that flexible features can for instance also be used to implement a facade design pattern.

While flexible features already help in modeling crosscutting functionality, they still suffer from some conveniences with respect to maintainability as they can only be specified as extensional lists of changes. In order to overcome these problems, we introduce an extension to our change model. An intensional change is a kind of change that can be applied on a software program p in order to apply the changes described by that intensional change to p. This application consists of two phases. First, the intensional change needs to be evaluated with respect to the change set C_p that specifies p. This step produces an enumeration of changes which are all applied on p in the second step.

We present a formal language that can be used to specify intensional changes. It is based on a tuple calculus and implemented by means of SOUL, an implementation of a Prolog-like declarative language on top of Smalltalk. Finally, we evaluate the extension of our model with intensional changes and present the advantages and drawbacks. Benefits include that intensional changes make features more reusable and feature composition more flexible. Drawbacks include an increase in complexity of the change specification and debugging process and the fact that the order in which the features are specified within a composition became important after the introduction of intensional changes, as those changes are evaluated with respect to a change set.

Chapter 8

Validation

The main goal of this dissertation is to validate that software can indeed be automatically modularised in feature modules if it is developed in a development environment that records fine-grained modularisation information resulting from development actions. In the previous chapters, we elaborated on a novel development approach which enables bottom-up feature-oriented programming (FOP). That approach consists of three phases: First, the software system is completely developed in a standard object-oriented way while fine-grained modularisation information is recorded (change collection). Second, that information is used to decompose the software system into feature modules (change classification). Finally, those modules are recomposed in order to construct variations of the software system (change composition).

In this chapter, we validate this approach in two steps. We first elaborate on the Change- & Evolution Oriented Programming Support (ChEOPS). It is a proof-of-concept implementation of a development environment that is capable of capturing fine-grained modularisation information resulting from development actions. It is written as a plugin for the Smalltalk VisualWorks development environment and supports all three phases (change collection, change classification and change composition) of our approach to FOP. Afterwards, we present a case study, which we implemented in ChEOPS and show that the resulting software system can indeed be automatically restructured in feature modules.

8.1 Proof-of-concept implementation

The high-level design of ChEOPS was already sketched in Chapter 3. Figure 3.7 recalls this design. In this section, we subsequently discuss all the parts of this design and how they were implemented. As Section 3.5 already extensively discussed about the *Change Browser*, we do not discuss it in further detail in this section. The *Logic Kernel* is elaborated on in Section 8.1.7, as we use it for integrating intensional changes in ChEOPS.

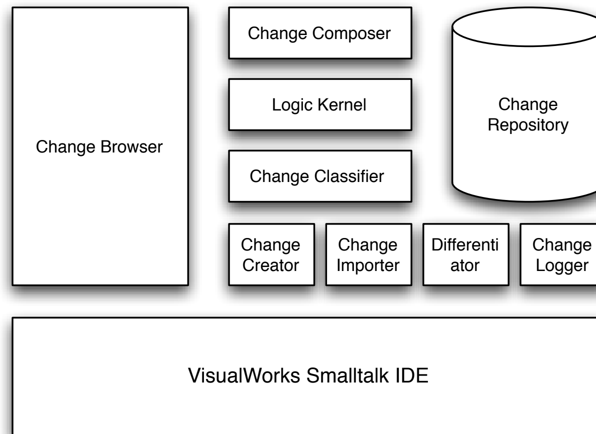

Figure 8.1: ChEOPS Architecture

ChEOPS is developed as a plugin for the VisualWorks interactive development environment (IDE), which we created as a proof-of-concept implementation of our approach to FOP. ChEOPS is a research vehicle that implements the model of changes described in Chapter 4 and does not fall back on ChangeList [107] – a change management tool included in most Smalltalk IDEs. Reasons for this are elaborated on in Chapter 3.

The goal of this section is to show that ChEOPS is a development environment that records fine-grained modularisation information resulting from development actions. In order to do that, we first explain why we chose Smalltalk and VisualWorks for Smalltalk as the programming language and IDE which we implemented ChEOPS in. Afterwards, we show how ChEOPS provides support for all three phases of our change-based approach to FOP (which are described in Chapter 5). We conclude this section by showing that the implementation supports the formal model that was introduced in Chapter 6 and by an explanation of how the implementation of intensional changes (from Chapter 7) is linked to ChEOPS.

8.1.1 VisualWorks for Smalltalk

We intend ChEOPS to be a research vehicle that proves the concepts of this work. Consequently, development speed is more important than execution speed. That premise is the main driver for choosing Smalltalk as the development language for ChEOPS. Concretely, Smalltalk provides three advantages. First, it is a *dynamically-typed* programming language, which speeds up the development of prototype implementations [106]. Second, Smalltalk provides powerful reflective capabilities to its own meta-model. This enables a programmer to adapt the Smalltalk language itself from within the language, again benefitting rapid proto-

typing. Finally, Smalltalk is a class-based object-oriented programming language that adheres to the FAMIX model. As our change model is based on the FAMIX model, it can be used to express the development and evolution of programs written in Smalltalk. Implementing ChEOPS in that same language, allows for a meta-circular implementation of ChEOPS by means of ChEOPS [1].

The interactive development environment which we implemented ChEOPS in is VisualWorks for Smalltalk. It is one of the leading commercial implementations of the Smalltalk programming language and environment. A benefit of VisualWorks, is that the non-commercial version has all the power and functionality of the commercial version. In both versions for instance, the user can inspect all the source code (including the one of the IDE itself). A strength of VisualWorks for Smalltalk is that it is itself implemented in Smalltalk and that it provides extension hooks for external plugins. These hooks are used to speed up the integration of ChEOPS into the VisualWorks IDE.

8.1.2 Model of first-class change objects

ChEOPS focusses on class-based object-oriented development and consequently implements the model of changes as it was presented in the class diagram of Figure 7.6 on page 153. The implementation contains a part that implements all the building blocks of the FAMIX model. Every FAMIX building block (Figure 4.2 on page 74) is implemented as a separate class that denotes that particular FAMIX building block. All these classes form the set of subclasses of the `Subject` class, which is the subject of `MyChange` class [2].

The `MyChange` class is sub-classed by the `AtomicChange`, the `CompositeChange` and the `IntensionalChange` class which respectively implement the component and composite roles of a compiste design pattern and the intensional change kind, that we introduced in Chapter 7. While `AtomicChange` consists of a single change object, a `CompositeChange` consists of a set of changes that need to be applied together in order to apply the composite change.

Dependencies between changes are implemented by two by two ordered collections that are maintained within the change object. A change object maintains the references to changes which it depends on semantically and structurally. In order to speed up the computation of transitive closures of dependencies, a change object does not only maintain the list of all change objects that it depends on, but it also maintains a reference to all change objects that in turn depend on it.

In ChEOPS, it would be impossible to represent the change objects in a non-first class way, as they are continuously manipulated by the tool. In ChEOPS, changes form the main development entity. As such, change objects must be continuously referenced, dragged and dropped, combined, queried, instantiated, etc. In order to do so, we need to maintain a reference to them in the *Change Repository*. The repository itself is implemented by the `ChangeLogger` class, which maintains an ordered collection of all the first-class change objects.

[1] In the end we did not develop ChEOPS by means of ChEOPS due to time constraints.

[2] Note that VisualWorks already contains a class named `Change`, which is used by the ChangeList tool. So as not to interfere with that tool, we name our `Change` class `MyChange`

8.1.3 Obtaining changes

In ChEOPS, a first-class change object is created by instantiating an `AddChange`, a `ModifyChange`, a `RemoveChange` or an `IntensionalChange`. The instantiation of a change requires a subject and some additional information that annotates the change with some extra information (the intent of the change, the user who instantiates the change, the time when the change is instantiated and the unique identifier of the change) that can later be used to modularise the software. After a change is instantiated, it becomes a first-class object, which can be referenced and passed along.

In ChEOPS, a reference to every change instance is maintained by the change logger – a single instance of the `ChangeLogger` class – that can be queried for changes. The change logger maintains a reference to all change objects and stores them ordered according to time. We now describe three different ways of instantiating change objects in ChEOPS. In this dissertation, we presented four techniques of capturing change: differentiation, change-oriented programming, logging or passing. Each technique is provided by a dedicated ChEOPS entity. Differentiation is done by the *Differentiator*, change-oriented programming is supported by the *Change creator*, the logging functionality is provided by the *Change Logger* and passing changes around can be done by the *Change Importer*. The current version of ChEOPS does not include the latter, but does include the former three change capturing techniques. We now elaborate on those.

Differentiation

The first technique to capture changes requires two versions of a (part of a) software system and computes the changes between both by executing a command similar to the Linux `diff` command on both versions. This technique is supported in ChEOPS by means of an already-existing IDE plugin called *StORE*. StORE is the version control system used by the VisualWorks for Smalltalk environment. It is based on a client-server architecture and uses a centralized server with a database acting as the central repository. Developers have the possibility to publish (commit) packages which will be versioned by StORE. Instead of versioning files, StORE works on a granularity-level of program entities (e.g. a class or method) which facilitates for example the merging of source code of different developers.

StORE also provides the functionality of comparing two versions of the same software system and is capable of calculating and presenting the differences between those versions detailed to the method level. This is presented in Figure 8.2. In the upper left pane of the figure, a list of classes is presented. The upper right pane contains the methods that were added, removed (crossed out) and /or modified (crossed out and not crossed out). The lower left and right panes show the class definition of the class respectively in both versions.

We extended the functionality of StORE in two ways. First we developed a parser that is capable of parsing method bodies and that can detect the differences between two method bodies. Second, we implemented the functionality that

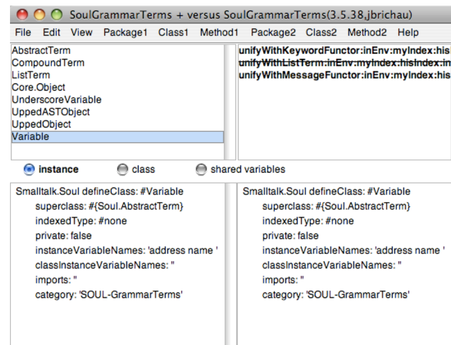

Figure 8.2: Differentiation to obtain change objects

instantiates the corresponding change objects for all the differences between two versions.

The current implementation of ChEOPS calculates the differences between two method bodies in a very naive way as it sequentially compares all the statements and produces a *RemoveStatement* for every statement of the old version and an *AddStatement* for every statement in the new version. A more intelligent comparison within method bodies (such as Envy [110] provides) might be better suited, but remains a topic of future work as it is not essencial to validate our work.

Change-oriented programming

Just like in many other IDE's (such as Squeak or Eclipse), change-oriented programming is already partially supported in VisualWorks: A class can be created though interactive dialogs, or the code can be modified by means of an automated refactoring. ChOP goes further than that, however, as it requires all building blocks to be created, modified and deleted in a change-oriented way (e.g. adding a method to a class, removing a statement from a method, etc).

ChEOPS provides an interactive dialog for every change kind, so that the developer can instantiate that change to engage in pure change-oriented programming. An interactive dialog queries the developer for all the information that is required to instantiate the concerned change object. Some of those dialogs can be triggered from within the standard IDE. The dialog for the addition of a class, for instance, can be triggered by right-clicking the package as illustrated in Figure 8.3.

Other dialogs can only be triggered from within the change pane of the ChEOPS plugin. Figure 8.4 shows different change types and the instantiate button that can be clicked in order to trigger the dialog.

Another functionality of ChEOPS allows developers to specify their own kinds of changes, by for instance composing atomic changes into a higher order composite change. ChEOPS only supports this principle in a textual way. A developer who wants to declare a new change kind, must write the definition of the new change

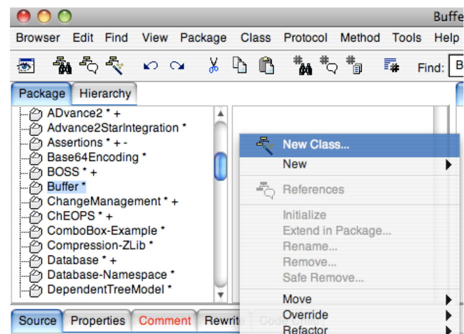

Figure 8.3: Change-oriented programming to obtain change objects

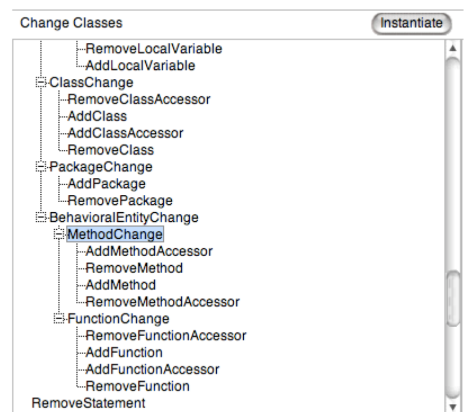

Figure 8.4: Change-oriented programming to obtain change objects (view 2)

kind by subclassing the composite change and implement the correct *new* : and *instantiate* : methods for the new change kind.

Logging

ChEOPS has the capability of logging developers producing code in the standard object-oriented way. To that regard, ChEOPS instruments the VisualWorks IDE with hooks and uses them to instantiate first-class change objects that represent the software development or evolution actions taken by the developer. In order to explain how this is done, we first explain how software is developed in a standard object-oriented way in VisualWorks for Smalltalk.

In VisualWorks for Smalltalk, there only exist three entities that can be defined: packages, classes and methods. All other building blocks are considered a part of one of those three. An instance variable for instance, is a part of its class description. A developer can add or modify one of these entities by typing the new definition of that entity in the corresponding IDE pane and by afterwards

clicking the *accept* button. Clicking this button triggers a method of the IDE that performs the necessary actions for (re)defining that class or method. ChEOPS overrides that method and adds the code for instantiating a change object for every change that is detected. Note that here, in fact we are also applying a differentiation at the level of method bodies and class definitions. In order to perform that differentiation, ChEOPS uses the same parser that we introduced earlier.

8.1.4 Change classification

In this section, we focus on the classification of changes into change sets that represent features. This functionality is implemented in the *Change Classifier*. Classification has two aspects: the classification model and the classification technique, which is embodied in the different software classification strategies. We subsequently discuss the implementation of the classification model and three classification techniques.

Classification model

The classification model is a metamodel that consists of two parts: the change model and the actual classification model. Each part focuses on another level of granularity. The change model describes how the changes are modeled. We refer the reader to Section 8.1.2 for an explanation on how that model was implemented.

The actual classification model defines and describes the entities of the superstructure which is a flexible organisational structure based on feature and change objects (represented in Figure 5.6 on page 109). A class named `Feature` was implemented. Instances of that class represent the actual features of a software system. A feature has a unique *id* (it's object id) and a *name*. While the *name* has to be provided by the one instantiating the feature, the *id* is provided automatically by the IDE.

The implementation of the D relation was already elaborated on in Section 8.1.2. $C4F$ is implemented by means of the `intent` attribute of the `Change` instances. Every change instance has an intent, which contains a reference to the feature the change is related to in terms of a $C4F$ relation. The cardinality of the $C4F$ relation is maintained within the change instance. How the intent of a change instance is specified depends on the classification technique that is used and is discussed below.

The *Sub* relation is implemented by a dedicated class `Sub`. An instance of the `Sub` class maintains a reference to the *parent* and *son* feature that are in a subfeature relationship. The *cardinality* of the instance, specifies whether the son feature has to be included in a composition that includes the parent feature. How the `Sub` class is instantiated depends on the on the classification strategy that is used and will be discussed below.

Classification techniques

The implementation of the classification of changes and features boils down to creating the right instances of the *Feature* and *Sub* classes, setting the *intent* of change instances to the correct feature instances and setting the cardinality of the change. There exist different techniques for classifying changes and features, which each have their own implementation. As in this dissertation we presented three classification techniques, we now present how ChEOPS supports these three techniques and how we implemented them.

Manual classification In this technique, a developer must manually create the features in which the changes have to be classified. In ChEOPS, this can be done by instantiating the `Feature` class and by providing the instance with a name that denotes the functionality of that feature. This classification of change instances in feature instances by means of manual classification implies that the developer manually sets the *intent* of a change to the *feature* that change should be classified in. Figure 5.7 on page 109 presents a screenshot of how ChEOPS supports this process. The developer can drag and drop a change instance from one pane (that contains the changes) to another pane (that contains the features).

Setting the cardinality of the change with respect to its feature must be done in a manual way as well. ChEOPS supports this by allowing a developer to select multiple changes and by giving him the opportunity of right clicking these changes and setting their cardinality together to 0 (denoting that it is *optional*) or 1 (denoting it is *mandatory*). Also the cardinality of the relations between features must be manually set by the developer. ChEOPS supports this process by allowing a developer to inspect the instances of the *Sub* relation and by setting their cardinality to 0 (denoting that it is *optional*) or 1 (denoting it is *mandatory*).

Semi-automatic classification A semi-automatic classification technique uses a technique such as clustering in order to propose a classification of change instances to the developer, who has the final word on classifying those instances in feature instances. The current version of ChEOPS contains one simple clustering strategy, which uses the timestamp of the change instances as a distance measure. Algorithm 15 presents the clustering algorithm that we used in our validation. It is a linear clustering algorithm that groups changes together if they were applied shortly after one another. Other clustering algorithms are concevable but were not included in the implementation, as that is beyond the focus of our work.

The clustering algorithm first calculates the average time between two changes in the change set and afterwards loops over all the members of the ordered collection of changes. If the timestamp of a change indicates that it was applied a *long* time (the average interval times three – a constant which shows to provide a good distribution for our example case) after the previous change, a new cluster is created. In the same loop, all changes are classified in the cluster they belong to. Although this is a very naive algorithm, it has shown to produces useful results.

The result of the clustering algorithm depends on the actual change set that it is given as an input and on the constant used to compare change intervals. As this

Input: *changes* $\leftarrow$ a set of change instances (ordered by timestamp)
Output: A set of features, that consist of changes

```
clusters ← ∅;
δt ← (changes last timestamp - changes first timestamp) / (changes size);
c_prev ← changes first;
cluster ← ∅;
clusters add: cluster;
foreach c ∈ changes do
    if c timestamp - c_prev timestamp > 3.δt then
        cluster ← ∅;
        clusters add: cluster;
        c_prev ← c
    end
    cluster add: c;
end
return clusters;
```

Algorithm 15: Clustering change instances based on their timestamp

section is about a proof-of-concept validation of our approach, we do not elaborate on the qualitative results of this clustering algorithm[3] but rather focus on how it is used and implemented in ChEOPS. Its implementation is done in Smalltalk and derives directly from the pseudo-code of the algorithm in Algorithm 15. In case it is required, a developer can modify this clustering algorithm or even create a new one. This is done by adapting the `cluster` method of the `ChangeLogger` class.

After the clustering algorithm has been finished and the clustering results have been presented, the developer has to manually classify the changes in features. Figure 5.8 on page 110 contains a screenshot of how ChEOPS supports this process. The developer can drag and drop the change instances – which are clustered based on their timestamp – from one pane (that contains the changes) to another pane (that contains the features). Behind the scenes, ChEOPS sets the *intent* of every change to the *feature* it is classified in.

In order to have features in which to classify change instances, a developer must – just as in the manual classification strategy – first create the features. This can be done by instantiating the `Feature` class and by providing the instance with a name that denotes the functionality of that feature. In order to speed-up the classification process, a developer can declare a default *cardinality* of a feature. All changes that are classified in a feature with *cardinality* 0 are by default set to *optional*, while changes that are classified in a feature with *cardinality* 1 are by default set to *mandatory*. After classifying a change into a feature with a default *mandatory* cardinality, the developer can reset the cardinality of the change to *optional* if that is required. This is done in the same way as in the manual classification strategy.

[3]Some qualitative results are presented in Section 8.2.

With respect to the *Sub* relation, this technique is not different from the manual classification technique. The developer must instantiate the *Sub* class with the right relations between parent and son features and set the cardinality.

Automatic classification The automatic classification technique relies heavily on the information provided at development time. In ChEOPS, we use a forward tagging technique to capture that information. Before a developer starts the implementation of a new feature, he is required to provide ChEOPS with his name and the name of that feature. ChEOPS requests the latter by means of an interactive dialog, which is triggered by the developer. The dialog presents a list of all features of the current project. The developer can select one of those features (in order to extend or adapt a previously created feature) or type a new name (for initiating a new feature). In case a new feature is specified, ChEOPS creates a new instance of the `Feature` class and provides it a name that equals the new name given by the developer. The relations between the features need to be declared manually by the developer. This is done and implemented in exactly the same way as for the two previously explained classification strategies.

Just as in the semi-automatic strategy, a feature has a default cardinality for its changes, which is specified by the developer, who can override the cardinality of every change afterwards. This might still be a tedious process, considering the high numbers of change objects even for small software projects. A conservative – but less tedious – strategy can be to declare all changes as mandatory with respect to their parent feature. Such a conservative strategy will obviously more often result in composition conflicts. Upon the detection of a conflict, the developer can of course still decide to modify the cardinality of a change in order to resolve that conflict.

Whenever a change is instantiated, ChEOPS automatically forward tags it with the information concerning the functionality it implements and who instantiated it. While in the current ChEOPS implementation, the latter is done by inserting a string in the *user* field of the change instance, the former is done by setting the *intent* of the change instance to the $feature$ instance, that corresponds to the functionality which is being developed. At any point in time, a user can query ChEOPS for an overview of all changes. Figure 5.9 on page 111 presents a screenshot of the result of the automatic classification in ChEOPS.

8.1.5 Feature composition

Our model allows the composition of a program variation by specifying the features that variation should include. Some compositions, however, are not possible due to unsatisfied dependencies. Others have multiple possibilities (due to the optionality of some changes). Thanks to the fine-grained level of feature specification, ChEOPS can check wether a composition is valid or not. Moreover, in case it is not, it can assist in resolving the conflict by providing detailed information about the conflict. The following elaborates on the *Change Composer*: the module of ChEOPS that deals with composition strategies, composition views and composition correction.

Change composition strategies

As already mentioned before, different change composition strategies might be conceivable – depending on what the developer desires. A maximal change composition, for instance, represents the most complete implementation of a feature composition. A minimal change composition stands for the most concise implementation of a feature composition. Yet another strategy could be a mix of both in which the optional changes that add code are included and the optional changes that remove code are omitted. This and other composition strategies, however, remain a topic for future work.

ChEOPS contains only one change composition strategy: the maximal one. It is implemented in the `generateCompositionFor:` method of the `ChangeLogger` class. The implementations maps to the algorithm for that strategy presented in Algorithm 1. In case another change composition strategy is required, another definition of this method should be provided. In case multiple strategies are required, this implementation should be refactored in such a way that it includes a *strategy* design pattern [42]. This design pattern would make ChEOPS more easily extensible with new change composition strategies.

The `generateCompositionFor:` method returns a list that contains two lists. The first one denotes all the changes of the composition that must be included in the composition in order to make it valid. The second one contains all the changes that caused the composition to be invalid (= all the changes with at least one unsatisfied dependency). Consequently, the composition is only valid in case the second list is empty. In case the composition is not valid, however, this second list can be used to point the developer to the problem(s) that cause the invalidity of that composition. This is further discussed below.

Composition views

ChEOPS includes a tool that can produce a graphical representation of a composition. In that graphical view, the changes and the dependencies among them are depicted as graphs. All the nodes that represent changes of the same feature are depicted in the same color. The nodes of the graphs can be inspected in order to obtain all the information of the change they represent. For the sake of easing debugging, the change objects that cause a composition to fail are colored in black.

As a base for the graphical framework, ChEOPS uses Mondrian [75]. Mondrian is a plugin for VisualWorks for Smalltalk that provides basic functionality for drawing and coloring graphical objects but also contains more powerful features, such as a clean distribution of graphical objects over the screen. ChEOPS uses that functionality for representing feature compositions. The `analyzeCompositionFor:` method of the `ChangeLogger` class first checks the validity of a composition by means of the `generateCompositionFor:` and afterwards produces the visualisation of the changes. Figures 8.8 to 8.11 contain screenshots of the graphical representation ChEOPS produced for different compositions. Note that the cardinality of a change with respect to its parent feature is also expressed in the

graphical representation (a circle is a mandatory change and a rectangle is an optional change).

Composition correction

In some cases, the developer requires a software variation that contains incompatible features. In such case, ChEOPS can support the developer in taking the right corrective actions. In order to do that, the developer should provide the `analyzeCompositionFor:` method with the desired composition. In case the composition is invalid because of some unsatisfied dependencies among the change objects, the resulting graphical representation will contain black nodes. The black nodes denote the spots that provoke the composition conflict. The developer can inspect these nodes in order to obtain extra information that can assist in completing the composition. Examples of such information are which changes should be included in order to make this composition valid or which changes would have to be excluded from the composition in order to make it valid. Such information can afterwards serve as an input for a reclassification of changes in features.

While not included in the actual version of ChEOPS, another functionality with respect to composition correction, may consist of the production of a feature diagram from the changes, the features and the relations among them. As was elaborated on in Chapter 6, a change specification can automatically be transformed into a feature diagram. This feature diagram represents the product family of the actual implementation of the software product. This might differ from the product family the designers of the software product had in mind. The comparison of the generated feature diagram and the originally designed feature diagram might reveal discrepancies, that might have to be corrected.

The final support ChEOPS provides with respect to composition correction, lies in the intent of the software building blocks of the software system. Every building block of a software system that was developed with ChEOPS maintains a reference to the changes that affect it. An inspection of this list reveals why it was created (and adapted). The list might also reveal which developer(s) "touched" this building block. Such information can assist a developer in the process of understanding a software system [43].

8.1.6 ChEOPS supports the formal model

Let us now show how the current version of our proof-of-concept implementation – ChEOPS – is capable of producing change specifications that adhere to Definition 3 on page 124 of Chapter 6.

The principal functionality of ChEOPS is capturing the development actions performed within VisualWorks and to represent them into instances of the `Change` class. The set of all these instances corresponds to the set C of the formal model. Once changes are collected, ChEOPS takes changes that belong together and puts them into an instance of the `Feature` class. The set of all these instances maps to the set F of the formal model, and the act of grouping of changes into features to the $F4C$ function (Equation 6.4 on page 123). The grouping relation is a function,

since ChEOPS ensures that every change belongs to exactly one parent feature. As $F4C$ is a function it does not matter that we implement the cardinality of a change with respect to its feature in the change itself.

ChEOPS has an interface that allows a developer to group features into higher-level features. The grouping relation actually maps to Sub (Equation 6.1 on page 122) of the formal model. For technical reasons, however, ChEOPS does not implement optionality as a property of the relation between two features, but rather as a property of the parent feature; i.e. the sub-features of a parent are either *all* optional or *all* mandatory. Consequently, ChEOPS allows only a subset of Sub as defined in the formal model, namely Sub with the property $\forall (f_a, f_b, x), (f_a, f_c, y) \in Sub \Rightarrow x = y$. The relation implemented by ChEOPS hence satisfies the formal model, but is more strict. Moreover, it satisfies properties 6.2 and 6.3 on page 122. Property 6.2 holds since the particular implementation of Sub implies that every feature will either be optional or mandatory with respect to its parent. Property 6.3 holds because ChEOPS ensures that the relation Sub over F only contains trees. In order to do that, it imposes two restrictions on the grouping of features. First, a feature can never be part of more than one other feature. Second, a feature can never be included in a feature that it already consists of.

The structural dependencies between changes are imposed by the meta-object protocol of the programming language used in the IDE. ChEOPS is capable of identifying all kinds of dependencies for dynamically typed programming languages that adhere to the FAMIX model (Section 4.4.1 on page 91), and records them while changes are applied. The set of all recorded structural dependencies corresponds to D_{str} (Equation 6.5) of the formal model. It satisfies the properties required for D_{str}, since it is: *irreflexive* (a change can never structurally depend on itself as that would mean that it would never be applicable in the IDE), *asymmetric* (if a change c_1 structurally depends on c_2, c_2 never structurally depends on c_1 as that would mean that both c_1 and c_2 would never be applicable from within the IDE) and *transitive* (if a change c_1 structurally depends on c_2 and c_2 structurally depends on c_3, c_1 always structurally depends on c_3. If c_1 can only be applied if c_2 is applied and c_2 can only be applied if c_3 is applied, we can indeed say that c_1 can only be applied of c_3 is applied).

The semantical dependencies between changes have to be manually instantiated by the developer as they depend on domain knowledge. ChEOPS supports this process by allowing the developer to inspect and modify attributes of the change objects. One of those attributes is `Dsem`: a collection of changes on which the owner of the attribute semantically depends. A developer can add or remove semantical dependencies from a change c by editing the `Dsem` of c. The union of the `Dsem` collections of all changes in C represents D_{sem} (Equation 6.9) of the formal model. It satisfies the properties required for D_{sem}, since it is: *irreflexive* (a change never semantically depends on itself as ChEOPS prohibits this) and *transitive* (if a change c_1 semantically depends on c_2 and c_2 semantically depends on c_3, c_1 always semantically depends on c_3. If c_1 can only be applied if c_2 is applied and c_2 can only be applied if c_3 is applied, we can indeed say that c_1 can

only be applied of c_3 is applied).

Finally, ChEOPS adheres to Equation 6.13 on page 124 as it only allows creating a feature by grouping changes and/or features, thus every feature in ChEOPS necessarily consists of sub-features, changes or both. From this, we can conclude that ChEOPS completely adheres to the formalisms introduced in Section 6.2.1 and that we can safely say that each change specification created with ChEOPS adheres to Definition 3 of page 124.

As the current implementation of ChEOPS only includes one composition strategy (the one of Algorithm 1), it always produces a maximal change composition. Using this as a default strategy makes sense, since it produces the system with the most complete implementation of the corresponding feature set. Other strategies – such as a strategy that produces the minimal composition (as defined in Definition 8 on page 127) – can be added to ChEOPS. From this, we conclude that ChEOPS provides an implementation of the formal model that we proposed in Chapter 6. Let us now take a look at how we integrate the intensional changes into ChEOPS.

8.1.7 Intensional changes in ChEOPS

This section focusses on the integration of intensional changes – that was already reported on in Chapter 7 – with ChEOPS. In order to support intensional changes, a *Logic Kernel* was integrated in ChEOPS. As a logic kernel, we chose the SOUL engine. The implementation of this integration is eased by the fact that SOUL – the language used to specify change cuts – is itself a plugin for VisualWorks for Smalltalk and that it has a powerful symbiosis with Smalltalk. We subsequently discuss how the change cut language uses the change objects of ChEOPS as n-tuples to reason about, how the actions of intensional changes produce ChEOPS change objects, how ChEOPS allows the specification of intensional changes and how this extension affects the formal model behind ChEOPS.

We first recall how a change is defined by the change cut language. The definition

$$change(?c)\ if\ member(?c, \texttt{[ChangeLogger currentComposition allChanges]})$$

states that something is a change if it is a member of the the result of the Smalltalk block [*ChangeLogger* currentComposition changes]. Upon execution of the block, the message *allChanges* is sent to the result of sending the message *currentComposition* to *ChangeLogger* class. This returns set of all the changes that are already included in the feature current composition. The specification of this Smalltalk block within SOUL query is already one example where the symbiosis between SOUL and Smalltalk proves useful. Another example can be found in the predicate definitions for retrieving the values of the attributes of the changes within SOUL. The predicate

$$user(?u, ?c)\ if\ change(?c), equals(?u, \texttt{[?c user]})$$

for instance, returns all the parts where $?c$ is bound to a change of the ChEOPS change set and $?u$ is bound to the value of the smalltalk block [`?c user`], which evaluates to the value of the *user* message that is sent to the object bound to $?c$.

Just as the formulas of the change cut language do, the actions of the intensional changes also use the symbiotic capabilities of SOUL. An action needs to instantiate the ChEOPS change objects denoted by an n-tuple. Remember the example of an action that has to produce a change object that adds a method just before another change:

$AddMethodBefore(?ParameterList, ?user, ?intent, ?c, ?c_{new})$ if
$equals(?c_{new},$ `[AddChange newMethod:?ParameterList`
```
                          before:?c
                          by:?user
                          for:?intent])
```

in which a new change object is bound to $?c_{new}$ after it is created by the Smalltalk block [AddChange newMethod: $?ParameterList$ before:$?c$ by:$?user$ for:$?intent$]. This block, when evaluated, produces a new change instance and registers it in the *ChangeLogger* class.

Now we showed how the symbiosis from SOUL to Smalltalk is used for obtaining and creating ChEOPS change objects from within intensional change specifications, we elaborate on how the symbiosis from Smalltalk to SOUL is used in order to specify intensional changes from within ChEOPS. Just like all ChEOPS change kinds, intensional change objects have to be created by instantiating a concrete subclass of the *Change* class. In case of an intensional change, the *IntensionalChange* class has to be instantiated. This can be done by invoking the `new:` method on that class. The parameter of the method has to be a sequence that denotes the actions and change cut of the intensional change. An example of such sequence is this one:

$$\begin{array}{l} AddStatementAfter((Buffer, false, ?m, \text{“}logit()\text{”}), \\ \quad \text{“}Peter\text{”}, \text{“}InvokeLog\text{”}, ?c, ?c_{new}), \\ AddMetod(?c), \\ method(?m, ?c), \\ class(\text{”}Buffer\text{”}, ?c) \end{array}$$

The query is maintained in a dedicated attribute of the intensional change instance. The `apply` method of the intensional change consists of two steps. It first calls the `eval` method on the intentional change and stores the result – which consists of the enumeration of changes of this intension – in a local variable. Afterwards, it calls the `apply` method of all those changes, in order to apply all of them. The `eval` method of an intensional change contains Smalltalk code that causes the SOUL evaluator to evaluate the query. The set of all new changes (= the values of $?c_{new}$) is returned by the `eval` method. Consequently, by sending the `eval` message to an intensional change, an enumeration corresponding to that intensional change can be obtained with respect to all the changes that reside in the current feature composition.

Thanks to splitting up the `eval` and `apply` methods of the intensional changes, one can evaluate a intensional change without applying it. This comes in handy for the implementation of a composition algorithm that produces the maximal change composition of a feature composition containing intensional changes – as presented in Algorithm 9. That algorithm requires an `eval` function that evaluates an intensional change without applying it.

8.2 Validation: FOText

The goal of this section is to demonstrate that an application – developed in an IDE that captures modularisation information resulting from development actions (like ChEOPS) – can indeed be automatically restructured in feature modules. We start out by explaining `FOText`: a feature-oriented implementation of a *word processor* that has been used in related work on feature-oriented programming [67]. We extend `FOText` with some crosscutting functionality to demonstrate the power of flexible features and intensional changes and elaborate on its object-oriented implementation in ChEOPS. Finally, we show how `FOText` can then automatically be modularised in feature modules that can be recomposed to form `FOText` variations that provide different combinations of functionality.

By implementing FOText, we also validate our approach to FOP. Concretely, we expect our implementation to allow for the expression of features that consist of additions, modifications and deletions of building blocks down to the level of a single code statement (*control over feature modularity*). We expect our model to support the customised deployment of *crosscutting functionality* without breaking the validity of a composition. Finally, we want to confirm that our approach to FOP indeed does not require an upfront feature-oriented design and as such enables *bottom-up FOP* without deviating from the standard object-oriented development process.

8.2.1 FOText design

FOText is an application that provides a graphic user interface in which users may type and edit texts. It also provides a menu – launched by the right mouse button – that allows the execution of certain functions to edit the text. FOText adheres to the *Model-View-Controller* design pattern [42].

Figure 8.5 presents the FODA diagram of the `FOText` application. In its original form, the `FOText` application from [67] contains all features but two; the `Compress` and the `Logging` feature, which we explain below. Features such as: `New`, `Quit`, `Open`, `Save`, `SaveAs`, `Print`, `Copy-Cut-Paste`, `Find`, `SelectAll` and `Help` are self explanatory. The `File` feature is a mandatory feature with respect to `FOText`, and must consequently be included in every product variation. The `New` and `Quit` features are also mandatory with respect to their parent feature and will have to be included in every product variation as well. All other features are optional with respect to their parent and – if required – can be safely omitted from a composition that includes the parent.

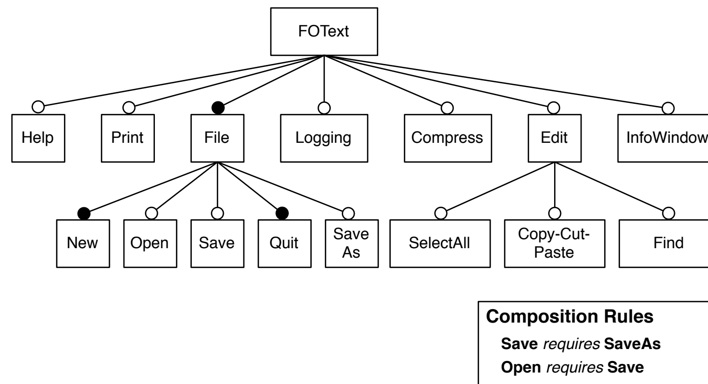

Figure 8.5: FODA diagram of FOText

The FODA diagram of Figure 8.5 also includes two composition rules. The first one imposes that every text editor variation that includes the `Save` feature also includes the `SaveAs` feature. The second one states that the `Open` feature can only be included in a composition that contains the `Save` feature. Note, that due to transitivity, these two rules impose that the functionality to `Open` a text file, can only be included if the functionality to `SaveAs` is included in the product variation.

We extend `FOText` with two more features (the `Compress` and the `Logging` feature), in order to demonstrate the power of flexible features and intensional changes. The `Compress` feature provides the ability to compress text files before they are saved, and decompresses them before they are opened. The `Status Title` feature displays the name of the opened file and the name of the file that is being saved in the title bar of the FOText window. It also clears the window title bar when the user starts a `new` file. We specify the latter two as flexible while the former nine are specified as monolithic.

The `Logging` feature is a feature that adds logging behaviour to the text editor. It is a crosscutting functionality and involves changes that depend on many `FOText` features. The composition rules state two other dependencies. The `Open` feature cannot be applied without the `Save` feature, which in its turn can not be applied without the `SaveAs` feature. The rationale of the diagram states that the `Compress` feature is useful for producing a variation for an environment that has few resources.

The UML class diagram of the complete `FOText` application is presented in Figure 8.6. The main class `Editor` has a method `execute` that produces an instance of the class `ApplicationWindow`. It provides the window to display and edit text. The `execute` method also creates an instance of the class `TextEditorView` which is linked to an instance of the `EditorController` and `KeyboardProcessor` classes. The `EditorController` class inherits from the `TextEditorController` class included in VisualWorks for Smalltalk and which adds some functionality such as

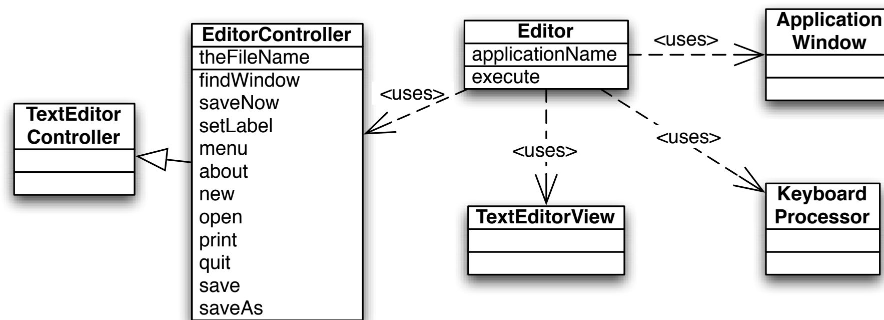

Figure 8.6: Class diagram of FOText

a method `menu` which is used to launch the `FOText` methods that implement the different features. The `KeyboardProcessor` captures the events originating from the keyboard and is linked to an `ApplicationWindow` to embed the text area into the window.

8.2.2 FOText implementation

We implemented `FOText` in a standard object-oriented way in Smalltalk and used the VisualWorks for Smalltalk IDE that was instrumented with the ChEOPS tool [31] to capture our development operations as first-class change entities by means of a *logging* technique. At the beginning of the development of a new feature, we inform the IDE of its name and type (*flexible* or *monolithic*). By doing that, our tool is capable of *automatically classifying* changes in features and by keeping track of whether changes are optional or mandatory with respect to their parent feature.

From the moment the changes are captured in first-class objects, they can be used to validate and compose different variations of the software program. Table 8.1 shows some statistics about the number of changes and dependencies ChEOPS captured when we developed `FOText`. Note that the numbers of changes and dependencies are about the same.

For the sake of simplicity, we introduce an artificial feature, `Base`, which is the basic feature that needs to be included in every variation of `FOText`. `Base` consists of the changes that should always be included in a composition: `FOText`, `File`, `New` and `Quit`. It provides the main functionality: a basic word processor that provides a window to type text and a menu with two choices: `new` and `quit` – which are respectively introduced by the `New` and `Quit` features. `Base` consists of the mandatory features `File`, `New` and `Quit`.

We incrementally add the implementation of the other features to this base program. Most of those features *add* a new method to the menu of the `FOText`

Feature	# changes	# dependencies
Base	130	158
SaveAs	88	106
Save	65	74
Open	101	121
Copy_Cut_Paste	72	82
Find	86	98
SelectAll	89	102
Print	182	226
Help	137	154
Status_Title	159	193
Compress	151	147
Total	1260	1362

Table 8.1: Statistics of the size of FOText

application. Some of them, however, also introduce *modifications* (e.g. the `Open` feature modifies the menu method introduced by the `Base` feature) and *removals* (e.g. the `Compress` feature deletes several *statements* and introduces new ones within existing methods).

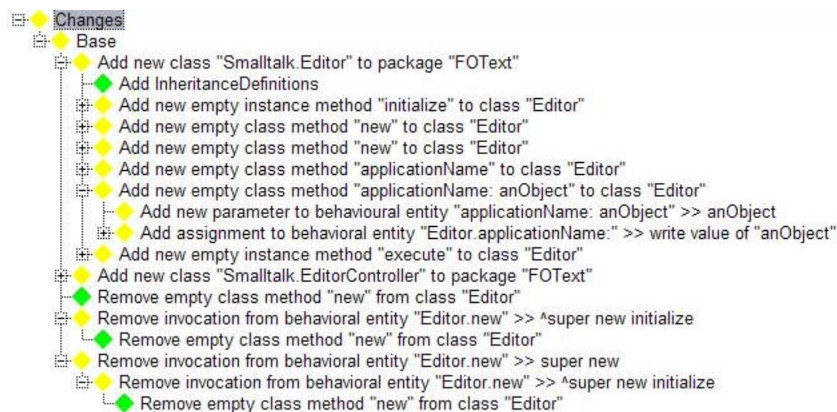

Figure 8.7: FOText: List of logged changes

Figure 8.7 presents a ChEOPS view which hierarchically presents the change objects captured for implementing the `Base feature`. It contains *additions* of classes, methods, instance variable or statements and and *removals* of methods and statements. The figure shows that (a) our model is capable of expressing features that include deletions of program building blocks, (b) that our model allows features to specify changes up to the level of statements. Finally, we showed that we were able to do feature-oriented programming in a bottom-up way by programming in a standard object-oriented way (the Smalltalk way) in a stan-

dard interactive development environment (in VisualWorks for Smalltalk). We now show how different product variations can be constructed by composing the modularised feature modules.

8.2.3 Feature composition

Our model allows the composition of a program variation by specifying which functionality that variation should include. As in our approach every functionality is linked to a set of changes – a feature, the specification of a set of functionalities can automatically be translated by a composition algorithm (such as Algorithm 1) into a change composition that represents the variation. In order to obtain the variation, the corresponding change composition has to be applied. This is done by applying all the change objects of the composition. Some compositions, however, are not possible due to unsatisfied dependencies. Thanks to the fine-grained level of feature specification, ChEOPS can check whether or not a composition is valid. In case it is not, it can assist in resolving the conflict by presenting the developer with fine-grained information about the conflict. In this section we present six compositions which validate the usability of these ChEOPS capabilities.

A valid composition

In this first scenario, we want a variation of `FOText` that includes all the features with the exception of the logging functionality, which we will add later in this chapter. Our tool informs that this composition is valid (there were no unsatisfied dependencies) and depicts the change composition graph of Figure 8.8.

This composition involves all features except `Logging` and is specified by 1260 changes.

A graphical overview of the changes of a valid composition, such as the one shown in Figure 8.8 hints graphically at the size of all participating features. Other uses of such a diagram might exist, but do not reside in the scope of this research. As we "only" want to validate and produce program variations, the layout of the changes of a valid composition does not seem useful and can be avoided in order to significantly speed up the variation validity checking and composition process.

An invalid composition

The second composition involves the `Base` and `Save` features. Figure 8.9 depicts the result of this composition: The changes belonging to the `Base` and `Save` features are respectively depicted as red and green circles. The black nodes represent the change objects that belong to the features in the composition which have at least one unsatisfied dependency.

The graphical view of an invalid composition (such as the one shown in Figure 8.9) is already more useful than a view on the changes of a valid composition, as it can be used to assist programmers in debugging their compositions. An inspection of the black nodes of the diagram of Figure 8.9, illustrates that there are

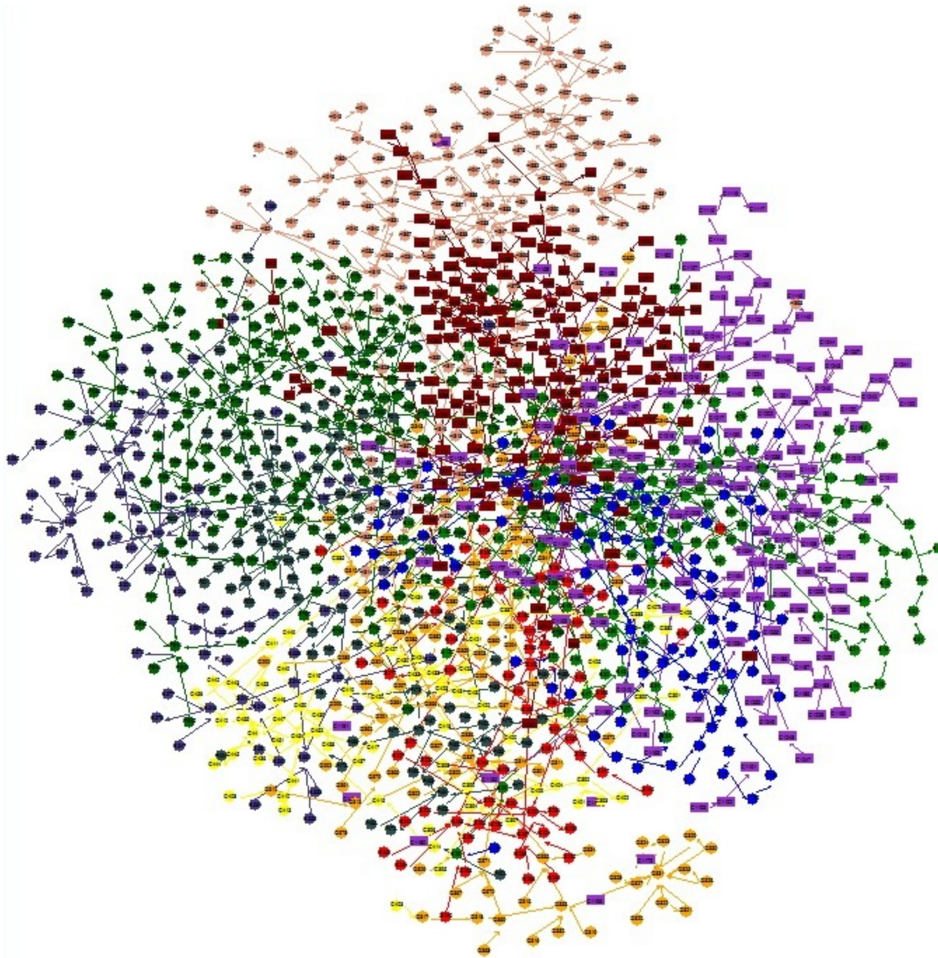

Figure 8.8: Composition of all features except for `Logging`

four changes from the `Save` feature that depend on changes of the `SavesAs` feature. Consequently, in this implementation of `FOText`, the `Save` feature cannot be included in a composition without including at least the `SavesAs` feature. In case this dependency is not desired, the developer can use the fine-grained information of the inspected black first-class change objects to adapt the implementation of the relevant features.

Note that the fact that `Save` depends on `SaveAs` is also included in the second composition rule of the FODA diagram of Figure 8.5. The reason why these features are dependent (the four changes of `Save` that depend on changes of `SaveAs`), however, is not included in the FODA diagram and only becomes apparent in the view of Figure 8.9. This shows that the problem and solution spaces are connected

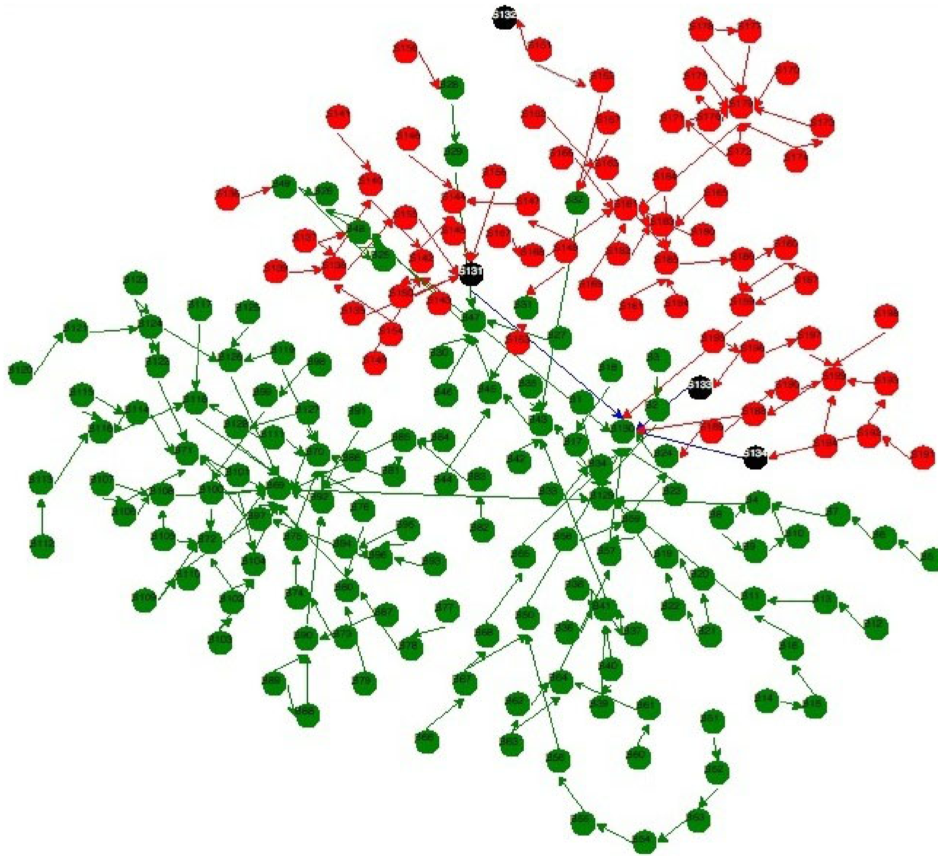

Figure 8.9: Composition of `Base` and `Save`

in our approach, and that this connection can be used for assisting the developer in correcting non-valid compositions.

Valid compositions by means of flexible features

ChEOPS allows a developer to declare a feature as *flexible*. The children of a flexible feature (the changes and/or features the parent consist of) are optional and do not have to be included for a composition that contains the parent to be valid. Consequently, a *flexible* feature provides a customized functionality depending on the features that are present in a composition. In the third and fourth scenario, we demonstrate this by composing the flexible `Compress` feature with different features.

We first compose the flexible `Compress` feature with the monolithic `Base` and `SaveAs` features. Figure 8.10 shows this composition: Changes belonging to `Base`,

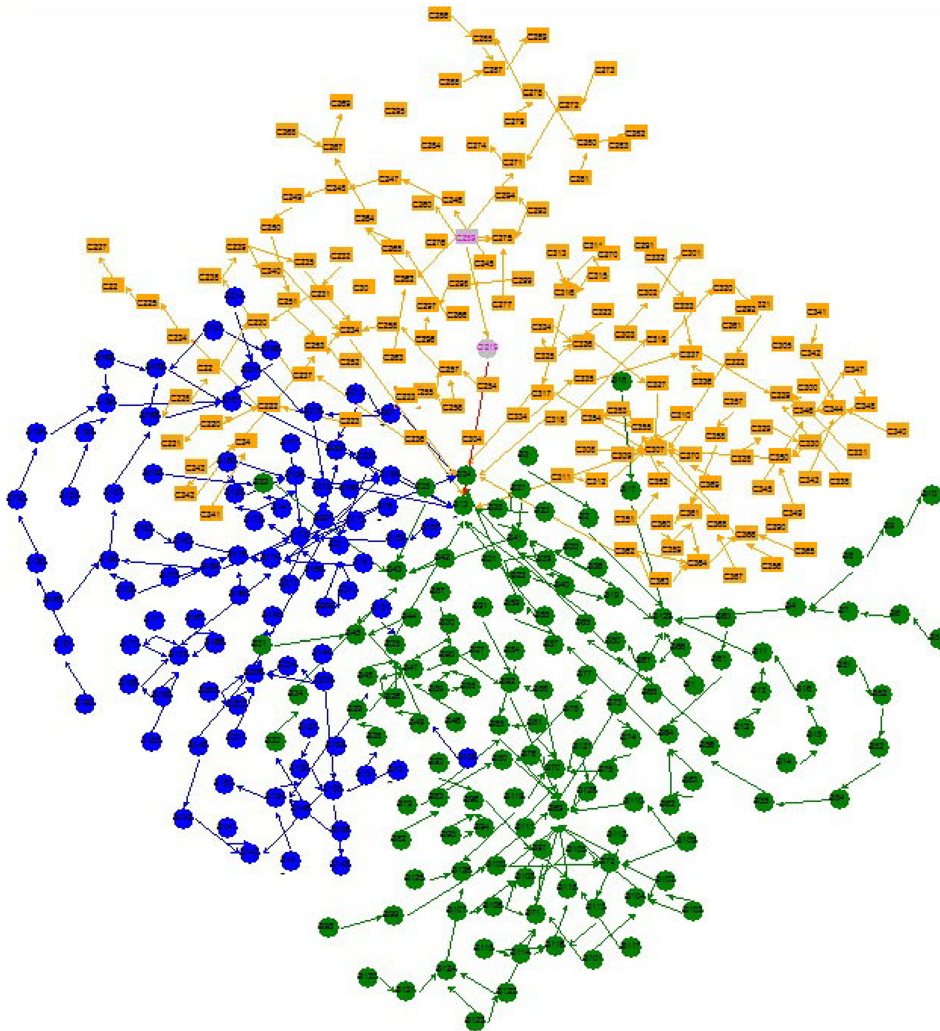

Figure 8.10: Composition of `Base`, `SaveAs` and `Compress`

`SaveAs` and `Compress` are respectively depicted as green circles, blue circles and yellow boxes. The composition is valid, but one of the changes of the `Compress` feature had to be omitted from the composition as it had an unsatisfied dependency. Since that change was optional with respect to its parent, it was omitted from the composition in order to make the composition valid.

Note that the composition diagram in Figure 8.10 contains some gray nodes that belongs to the `Compress` feature. The gray nodes denote the change objects that were omitted from the composition due to at least one unsatisfied dependency

they have. The nodes can be inspected by the developer in order to verify why they were not included. An inspection of the change shows that it depends on a change of the `Open` feature.

In the fourth composition, we add the flexible `Compress` feature to a viewer version of `FOText` which is composed of the monolithic `Base` and `Open` features. The result of this composition is depicted by the diagram of Figure 8.11. Changes of `Base`, `Open` and `Compress` are respectively depicted as green circles, blue circles and yellow boxes. In this composition, several changes of the flexible `Compress` feature are grayed out and omitted from the composition (because they depend on changes that belong to the `SaveAs` feature).

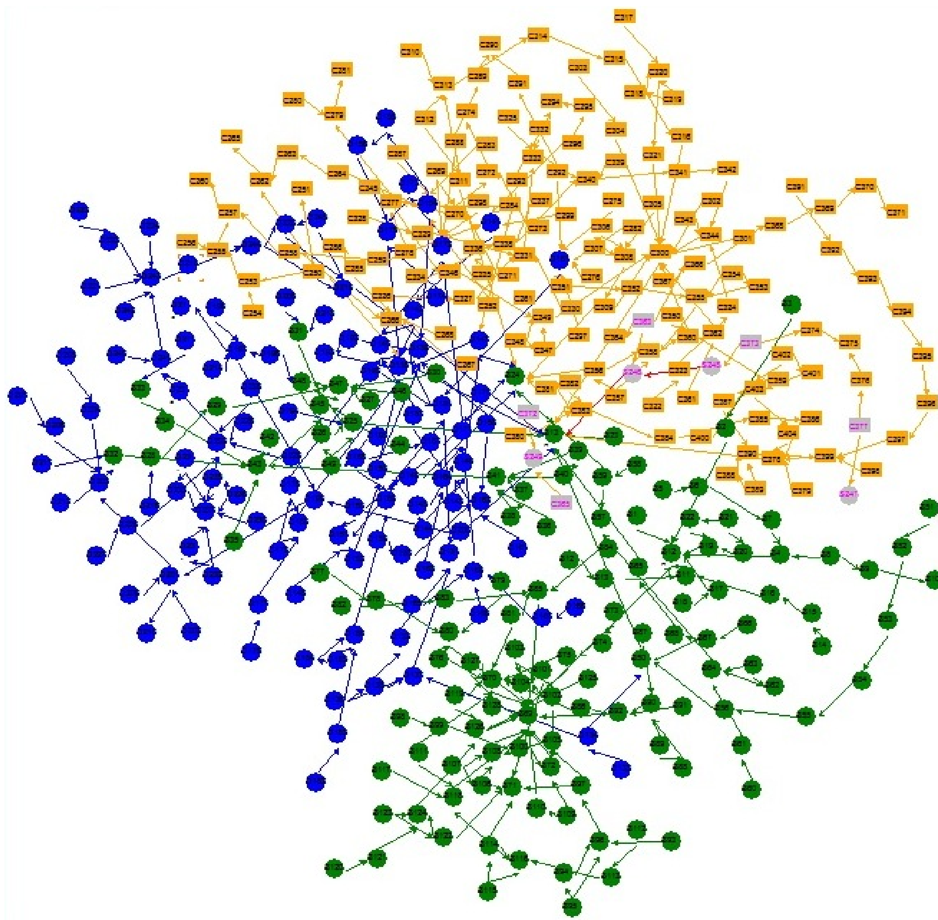

Figure 8.11: Composition of `Base`, `Open` and `Compress`

A closer inspection of the gray entities of both figures 8.10 and 8.11 learns that different change objects of the `Compress` feature are omitted from the com-

position depending on the composition the Compress feature belongs to: C259 in Figure 8.10 and C363, C365, C372, C373 and C377 in Figure 8.11. Those change objects are omitted from the composition because they depend on at least one change object that does not reside within the composition. C259 consists of the addition of a statement to the open method, which is originally added by O219 in the Open feature. It is omitted as the Open feature and its changes are not included in the composition of Figure 8.10.

Similarly, C363, C365, C372, C373 and C377 are changes that respectively depend on S246, S249, S249, S248 and S247, and which are automatically omitted from the composition as the SaveAs feature is not included in the composition of Figure 8.11. This demonstrates how our approach and tools automatically customize the deployment of *flexible* features in order to make compositions valid. It shows how this approach supports the construction of compositions that would not be permitted by other FOP approaches, but which do make sense. Consequently, this validates that our model supports the customized feature deployment, which improves the reusability of feature modules.

Valid compositions including crosscutting functionality

Finally, we demonstrate how we can deal with crosscutting functionality: which implementation affects many feature modules. Concretely, we present two compositions in which we include Logging; a feature that modularises the functionality of printing the value of each instance variables whenever a method in the FOText application is invoked. The implementation of this feature includes two intensional changes, which ensure that the feature is applied correctly in any composition.

The first composition consists of a variation of the viewer version of FOText (which is composed of the monolithic Base and Open features) and which has logging capabilities. Our tool informs us that this composition is valid (there were no unsatisfied dependencies) and depicts the change composition graph of Figure 8.12. The changes of the Logging feature are represented in purple. Note that the Logging feature contains 181 change objects.

Now, let us produce a composition that contains all the features of FOText including Logging. Figure 8.13 presents the change composition produced by ChEOPS. The change objects of the Logging feature are again represented in purple. While it is impossible to count all the changes in the figure, please note the high amount of purple changes. In fact, there is a total of 541 changes representing the Logging feature. The reason for the higher number of Logging changes is that it requires more changes to add the Logging functionality to a composition that contains more instance variables and methods as the Logging feature is supposed to log the values of *all* instance variables whenever *each* method is invoked.

These final two compostions show how features that contain intensional changes evaluate to different change sets depending on the context without having to adapt their specification. This principle shows how a combination of flexible features and intensional changes can be used to modularise crosscutting functionality without giving up on flexibility when it comes to software composition.

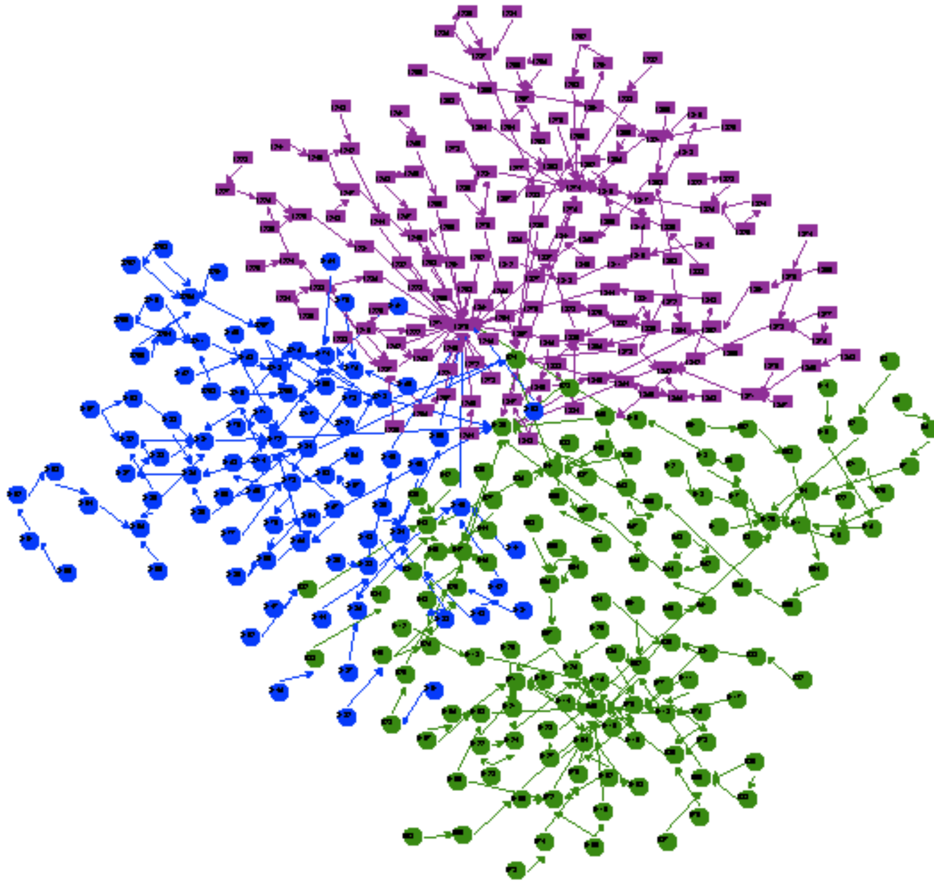

Figure 8.12: Composition of `Base`, `Open` and `Logging`

8.3 Lessons learned

As we developed the text editor by means of our tools and method, we gained some experience on how our bottom-up approach to FOP works in practice. This section elaborates on those findings, and reveals some weaknesses and strengths of the practical applicability of the approach:

- **Development**: It turned out that forward tagging the changes in the right way was not always that easy from the start. At some point, while developing the `Save` feature, we realised that some functionality was missing in the `SaveAs` feature. Consequently, we had to "modify" the `SaveAs` feature by adding some changes to it. In practice, we first had to tell ChEOPS that we were developing the `SaveAs` feature again, so that it could forward tag the upcoming changes in the right way. Afterwards, we could make the changes

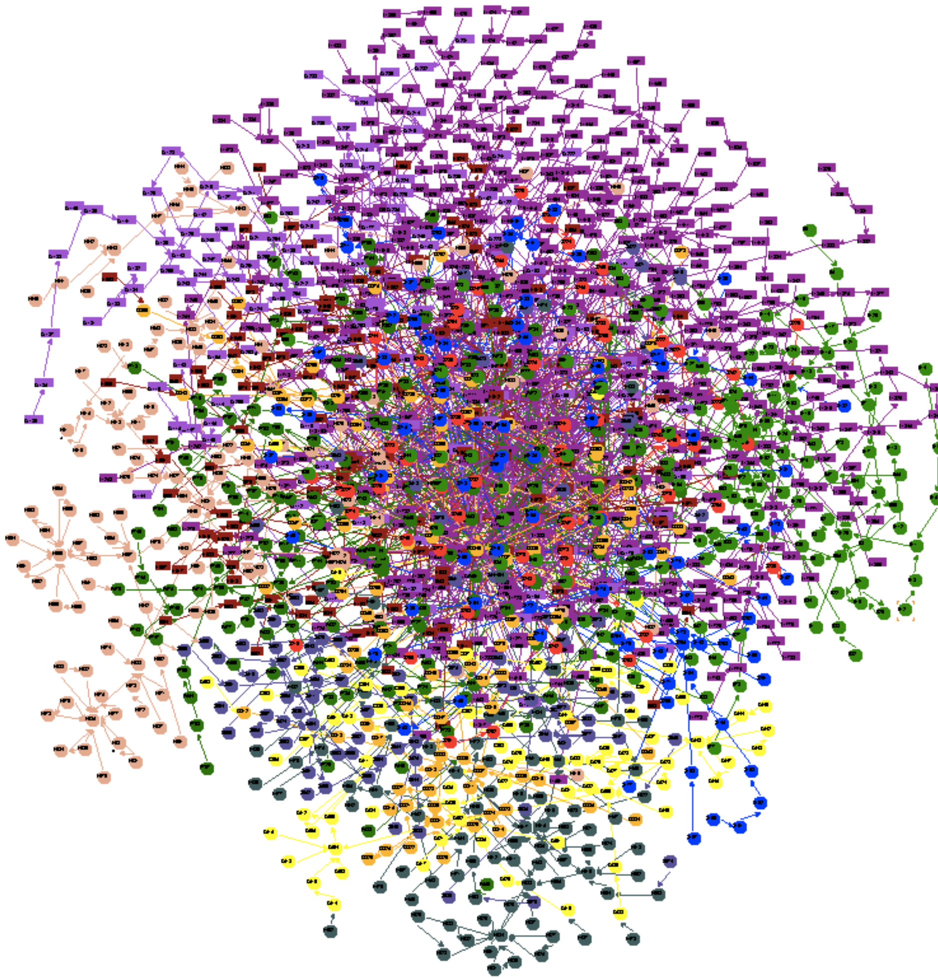

Figure 8.13: Composition of all features and `Logging`

that were required to "modify" the `SaveAs` feature. This observation makes us believe that the development of features is an iterative process, which had to be supported by ChEOPS.

- **Scalability**: The complete composition of the text editor consisted of 1801 changes, of which 1260 were not intensional. The time required to display the diagram in Figure 8.8 was 281873 milliseconds (approximately 5 minutes) on a standard PC. We realise that this time grows exponentially with the number of changes. A closer inspection learned us that our composition algorithm validated the composition in only 183 milliseconds and that the remaining time was used for lay-outing all change objects. In fact, the

required time of the verification algorithm only grows linearly with the number of changes. Still, the number of changes is quite high, considering that FOText is an application of +- 1 KLOC. We realise that the high number of changes might hinder scalability.

- **Compensating actions**: A closer inspection of the change set, learned us that for the text editor, about 20% of the changes are compensating one another. This is a consequence from logging all development actions as additional change objects, and never "throwing away" change objects that are compensating one another. This learns us that one possible optimisation could consist of pruning the change set before compositions are validated or produced.

The experiences we obtained by developing FOText, reveal possibilities for future work, which we elaborate on in Chapter 9.

8.4 Conclusion

The thesis of this dissertation is that software can be automatically restructured in feature modules if it is developed in a development environment that records fine-grained modularisation information resulting from development actions. This chapter consists of two steps for validating that hypothesis. We first discuss a proof-of-concept implementation of a development environment that records fine-grained modularisation information and than show how an application developed in that environment can indeed be automatically restructured in feature modules.

The Change- and Evolution- Oriented Programming Support (ChEOPS) is a tool-suit which we created in VisualWorks for Smalltalk – an interactive development environment for Smalltalk. ChEOPS is a VisualWorks plugin that supports techniques such as differentiation, change-oriented programming and logging for obtaining change objects. It supports manual, semi-automatic and automatic classification of changes in features and allows to specify features to be flexible or monolithic. While a flexible feature will by default only have optional children, a monolithic feature will by default have only mandatory children.

ChEOPS supports the composition of program variations. The developer only has to specify the desired functionality, after which ChEOPS can automatically produce the corresponding variation. For that, ChEOPS first composes the set of change objects that corresponds to the desired variation and afterwards applies those changes in order to get the desired program variation. The current ChEOPS implementation includes only one composition algorithm which produces the maximal composition of a variation. We explain how the implementation has to be adapted in order to add other composition algorithms.

ChEOPS provides two kinds of support with respect to debugging faulty compositions. First, ChEOPS is capable of producing a change diagram that is based on the implementation. As we have demonstrated in Chapter 6, this change diagram is automatically transformable to a feature diagram, that can possibly be

compared to the feature diagram that was intended for the application. When discrepancies are detected, the appropriate actions can be taken. Second, ChEOPS provides support for retrieving the reason of a failure when a specification of a software variation results in an invalid change composition. It uses the fine-grained implementation information that is contained within the change objects to assist the developer in taking appropriate corrective actions.

We show that ChEOPS is coherent with the formal model on which we elaborated in Chapter 6. It provides an implementation for changes, features and all the relations between them and contains an implementation of the maximal composition algorithm. Intensional changes are integrated in ChEOPS by linking the fact base of SOUL to the changes that are collected in ChEOPS and by instantiating the right change classes from within SOUL whenever new changes have to be added by it.

The second part of the validation introduces FOText: an application which we developed in order to validate our bottom-up approach to FOP. FOText is a text editor that consists of 16 features of which three are mandatory and the rest is optional. We developed it in a standard object-oriented way and used ChEOPS to log our development actions. The development of the complete application resulted in approximately 1250 changes and 1350 dependencies. The changes show to be expressive building blocks of features, as they can express additions, modifications or deletions and because they can be specified on the level of granularity of statements. They allow for a fine-grained control over feature modularity.

Afterwards, we let ChEOPS classify the changes in features and made six compositions that each represent one product variation of FOText. We use the graphical views on change compositions to obtain information about the invalid compositions and show how the fine-grained control over changes entails information that can be used to find out why some feature compositions are invalid. Finally, we show how crosscutting functionality can be modularised and specified by means of flexible features and intensional changes in order to increase the reusability of feature modules in different compositions. The findings of this second part acknowledge that FOText was automatically decomposed in feature modules which could afterwards be recomposed to form product variations of FOText.

Chapter 9

Future Work

Many enhancements to the bottom-up approach to feature-oriented programming, the evolution model, the classification model or the implementation are conceivable. A handful is considered below. Moreover, we elaborate on other applications of first-class changes in the software engineering research domain, on how the mapping of our formalism to feature diagrams can be better exploited, on how software applications can be refactored into modules that denote context layers and on how we can use rules of changes for ensuring design contracts when implementing the software.

9.1 Overcoming restrictions

Throughout this dissertation, we made three restrictions with respect to the applicability of our approach. The first track of future work consists in removing these restrictions. The following three subsections subsequently elaborate on the restriction of not allowing parallel development, on true scalability and on the meta-model which can only be used to model the evolution of software systems written in a class-based object-oriented programming language.

9.1.1 Parellel development

In order to simplify matters, we limited ourselves to a setting in which parallel development was excluded. This restriction was based on two premises: (a) All change instances are part of a set that can be completely ordered by the time stamp of the changes. In other words, every change has a unique time stamp. (b) The IDE ensures that no conflicting changes are produced[1].

In a development setting in which main-stream IDE's are used and in which parallel development is allowed, those two premises can be guaranteed only within every separate development track. From the moment two development tracks

[1] Changes conflict if their application violates the rules that are specified by the meta-model of the programming language used

merge, they cannot be guaranteed anymore. As the results of this dissertation are based on both premises, we require a mechanism that makes them hold again. With respect to (a), a distributed clock (such as a Lampart [62] or vector clock [71]) might be used to obtain a global clock in a distributed setting. With respect to (b), an IDE could be conceived that exchanges development information with the IDE's that are used by the other developers and which together avoid that conflicting changes are instantiated.

9.1.2 True scalability

We took as a premise that a bottom-up approach to FOP, which is based on fully automated classification and recomposition strategies, does scale up. This, however, is not always the case, as software composition is often an interactive process in which the developers' intervention is required to resolve composition conflicts. Consequently, both the speed and the memory consumption of the composition process are of great influence on the scalability of our approach. It is for example tolerable to wait for some minutes, but it is not acceptable having to wait for days for a composition to be produced.

Our classification and composition strategies have performance characteristics that grow linearly with the number of available change instances. The number of change instances, however, grows very rapidly. Our studies show that even for small applications (+- 1KLOC), the number of change instances grows to over 2000. In order to speed up the composition of variations, we consider pruning the compensating changes from the change set before compositions are produced. As any decrease in the number of changes is directly translated into a decrease in the time needed to produce a composition, we believe a speed-up of about 20% belongs to the possibilities – considering that the change set usually contains about 20% change instances that are mutually compensating each other (for instance the addition and deletion of the same method). Another action we consider for increasing the scalability is decreasing the level of granularity of the changes. Of course, such action has a tradeoff: a loss of information.

9.1.3 The meta-model

The evolution-model we introduced and described in Chapter 4 can be used to model the evolution of any software system developed in a programming language adhering to the FAMIX model. While many class-based object-oriented programming language (such as Java, C++, Smalltalk or Ada) adhere to this model, others might not. In this dissertation, we only showed that we can model the evolution of software programs developed in one of the programming languages adhering to FAMIX.

This track of future work consists in showing that our evolution model is generic enough to support the evolution of other software systems. In order to do that, three actions need to be taken. First, the evolution model should be extended with a new family of subjects that form the building blocks of the meta-model of the software systems that are to be evolved. Concretely, all these building blocks

will be modeled by new kinds of subjects and implemented by subclassing the `Subject` class correspondingly. Second, different kinds of syntactical dependencies between changes have to be reconsidered, as they are enforced by the meta-model. Finally, the IDE support for change-oriented programming (ChOP) has to be extended in such a way that changes on the new meta-model can be instantiated and dependencies are maintained upon change instantiation.

9.2 Classification strategies

In this work three software classification strategies have been presented: manual classification, semi-automatic classification based on clustering techniques and automatic classification through forward tagging. Further research is necessary to identify other classification strategies and to evaluate their usefulness in the context of bottom-up feature-oriented programming.

It is mostly the semi-automatic and automatic techniques that we consider to be interesting as they improve scalability. The exploitation of more such techniques promises to produce interesting results. What explicitely seems interesting, is the role a logic metaprogramming language can play in the classification of change entities. A language like SOUL (the Smalltalk Open Unification Language which was developed at the programming technology lab of the VUB) can be used to classify changes according to a given logic query. Recently, SOUL was successfully used by Kellens et al. for classifying the code of software systems with respect to the satisfaction of design contracts [55].

9.3 Applications of first-class changes

The centralisation of change in entities that encapsulate the change operations (Chapter 3), enables applications that can contribute in other sub-domains of software evolution. In this section, we elaborate on some of these application domains and show how they form possible tracks of future work.

In principle, any operation should be undoable if and only if it is meaningful (e.g. the referred objects still exist) and if users can achieve the same effect through standard editing, as advocated by many researchers in this area (Dix [69], Berlage [13], Prakash [87] and Knister [88]). In contrast to a *linear* history model, the *non-linear* history model allows users to undo an operation other than the last one. The encapsulation of development actions in dedicated objects, enables the application of the non-linear history model. Moreover, the dependencies amongst the change operations, can be used to achieve a cascading undo, in which all change operations that depend on the undone change operation are also undone in a transitive way. We already implemented cascading undo and included it in the ChEOPS tool suite.

Every operation that was ever undone, is in principle “redoable”. In some cases, however, a change operation c can only be redone if that action is preceded by the redoing of some other changes [35]. The dependencies amongst the change

operations can be used to recover the transitive closure of change operations that must be redone before redoing c (cascading redo). We included an implementation of the cascading redo in ChEOPS.

In some cases, the undoing or redoing of a change, brings along a large cascade of changes that will also have to be respectively undone or redone. In some cases, it is desirable to estimate the impact the undoing or redoing of a change operation might have, before actually carrying it out. The study of impact estimation is called *impact analysis*. We strongly believe that the dependencies between the change operations can be exploited to help predicting the impact of redoing or undoing a certain change.

In Chapter 7, we elaborated on intensional changes. Those changes can also be used to express *intensional undo statements*, which can express requests like: "undo all changes applied by Peter that were done on or before March 23, 2009". While ChEOPS already supports the expression of such queries by means of SOUL, we did not yet test how they are defined and applied in practice. A more extensive study might expose more difficulties, that must be investigated in this track of future work.

This dissertation is about the modularisation of software systems. Concretely, we modularise a software system in such a way that modules represent a functionality and that they can be composed to form variations of the software system that offer different combinations of functionality. Software systems can also be modularised in other ways. They can for instance be modularised in such a way that the *degree of coupling* between the modules is minimised. This degree can be measured by the number of dependencies that hold between the changes of the different modules. Clustering techniques can most probably be used for doing that.

Runtime evolution is another field in the domain of software evolution in which we feel confident contributing to with the notion of change objects that encapsulate development actions. In [33] we introduced the idea of change-oriented advanced round-trip engineering as a technique to support runtime changes, automated testing and refactorings in the context of Agile Software Development (AgSD). We propose to represent changes applied to a system under development, as first-class objects and envision the integration of a change management system into an existing methodology for AgSD called *Advanced Round-Trip Engineering* (ARTE) [83]. We explain how *Change-Oriented ARTE* allows capturing, visualising, replaying and rewinding changes that have been applied on the modelling, implementation and runtime views of an ARTE environment, and automatically synchronizes them with other views. In this setup tests and refactorings can be composed out of changes, while runtime change propagation is realized with different propagation strategies.

9.4 Formalism

The elaboration of the formal ChOP model led us to take a high-level look at ChOP, independently from an implementation. This allowed us to identify new

interesting properties and to make some generalisations. Therefore, an immediate topic for future work is to improve ChEOPS based on the feedback gathered while developing the formal model. For instance, the current version is not expressive enough to cover the whole formal model, given that the relations between a feature and its changes ($F4C$) and between features (Sub) are implemented with the restriction that the optionality of a sub-feature (change) with respect to its parent is an attribute of the parent, and not of the relation. The design and implementation of ChEOPS will be refactored to overcome this issue.

Another example of future work induced by the formal model are the different strategies that can be used to produce legal change compositions. At the time of writing, we only formalised two of them: produce maximal (respective minimal) legal change compositions. We also sketched other conceivable strategies, which would also take into account the nature of a change (addition, deletion, modification). These strategies need to be formalised and can then also be implemented in ChEOPS.

Currently, the dependencies between change objects reflect low-level constraints only. The new connection between feature diagrams and ChOP also allows to express dependencies and constraints on the application level. Another track of future work may consist in how to carry back such high-level dependencies and constraints to the change object level.

A final track of future work that is fed by the formalism consists in extending the range of applications of feature diagrams to ChOP. In this dissertation, we only used feature diagrams to validate change compositions and to express basic properties of change specifications. The state-of-the-art work on feature diagrams, however, includes many more applications such as visual modelling support, specification of metrics, program understanding, etc. which we plan to investigate in order to find out how they can be helpful in ChOP.

9.5 Deriving intensional changes

In this dissertation, we explained how features are formed by grouping changes that model related development actions. These groups can be specified in an *extensional* way (by explicitly listing them) or in an *intensional* way (by describing them). In Chapter 7 we argumented that defining changes in an intensional way is desirable, as it increases the reusability of flexible features.

In Chapter 7 we also explained how intensional changes can be defined by means of change-oriented programming. The other change gathering strategies of Section 5.4.1, however, can not be used to instantiate intensional changes. Both logging and differentiation strategies inherently result in extensional change sets. This track of future work, consists of allowing the specification of intensional changes by means of logging of differentiation.

Consider a development scenario, in which the developer manually adds a method to *all* subclasses of a class C. In a first step, we can allow the developer to replace the extension holding all method additions by an intension that states that a method is added to all the subclasses of C. In a second step, a

pattern matcher could be used to detect refactoring opportunities in the set of changes (suggesting to pull up the method to C).

9.6 Feature refactoring

The research described in this text, is about modularising object-oriented software systems into modules that encapsulate functionality. From one point of view, this corresponds to *refactoring* an object-oriented software application into a feature-oriented software application. The benefit of such refactoring is that the variation points (which are usually implemented by if-stataments or by means of object polymorphism) are not visible in the source code of the application, making it more understandable, reusable, and maintainable.

Driven by the observation that a software application sometimes behaves differently depending on the context in which it is operating, we consider refactoring software applications into context layers. Context layers hold the context-dependent behaviour: the behaviour a software application should have in a certain context. The variation points that model the context-dependent behaviour switches are usually also implemented by if-stataments or polymorphism. The difference between the if-statements of these variation points and the ones explained above, is found in the location of those statements. In context-oriented programming, there should not be any variation point code inside the different context modules.

The goal of this track of future work is to support the refactoring of software applications in a *context-oriented* way. Our change objects – which model the development actions – are aware of which feature they implement. The source code entities which are affected by those changes can also be grouped and extracted into another dimension of modules: context layers. This would allow a multi-dimensional separation of concerns, which increases the reusability of all separated modules.

9.7 Ensuring design contracts

In some cases, design contracts require two different development actions to be taken together. Consider the `=` method, which is implemented by many classes in the VisualWorks Smalltalk development environment. It is a method that compares an object to receiving object and verifies that they are the same. When inspecting the source code of the method, a comment is found mentioning that the `hash` method should be changed accordingly, in case the `=` method is modified. As this *design contract* is only apparent as a source code comment, many violations are made against it, resulting in bugs that are hard to detect.

We propose to make design contracts explicit by including them as first-class rules that denote patterns of change. When a pattern is detected that breaks one of the rules, the developer can instantly be notified. Concretely, we propose to include a user-extensible rule base that contains all design contracts in the development

environment. The rules of the rule base consist of an "IF" (denoting a condition) and a "THEN" part (denoting an action). Both parts consist of change templates, to which the change instances may match. Whenever a change is instantiated from within the development environment (by change-oriented programming, logging or differentiation), the rule base is checked for possible inconsistencies and the developer is proposed the action included in the "THEN" part of the violated design contract rule.

In the scenario from above, the rule would be "IF AddMehod(?class, ?classSide, =) THEN AddMehod(?class, ?classSide, hash)", which states that in case a method `=` is added to any class, a method `hash` should be added to the same class. Note that an inverse rule should also be included as this particular design contract is symmetric. In case a developer instantiates a change that adds a `=` method to any class, he is suggested to also add a `hash` method to that same class. Different suggestion strategies are conceivable. The suggestion can for instance be made proactively or on demand (being less intrusive). The advantages and inconveniences of these strategies remain the subject or future work.

In this chapter, we have enumerated some tracks of future work. While some of them consist of enhancements to the conceptual and technical contributions that we made, others elaborate on the application of our research results in other fields of the software engineering research domain. By enumerating some tracks of future work, I realised that research never ends, but that I will always strive to find closure.

Chapter 10

Conclusions

The objective of this work has been the bottom-up modularisation of software systems around the functionality they provide. Since the modularisation of a software system comprises too many aspects to handle in a dissertation the scope of this work has been narrowed to the modularisation in a context of program variation. The roots of this research being the research on change-oriented programming have driven further scope reduction.

The result is that this work has three foci of attention: First, the recording of modularisation information – which is lost if it is not made explicit at development time; Second, the bottom-up approach to feature-oriented programming – which provides an alternative for all top-down approaches. This topic includes the classification of software building blocks into modules that represent the different functionality a software system provides and which can be used afterwards to construct software variations with different combinations of functionality; Third, the support for a multi-dimensional separation of concerns – which tackles the tyranny of the dominant decomposition.

10.1 Summary

The subject of this dissertation is to manage bottom-up program variation in object-oriented systems. In Chapter 2, we present background material on the research domains directly related to this work: software modularisation by feature-oriented or aspect-oriented programming and the reification of change into first-class change objects. Our thesis is that software can be automatically restructured in recomposable feature modules if it was developed in a software development environment that records fine-grained modularisation information resulting from development actions.

Recording modularisation information In order to facilitate the recording of modularisation information, we propose a new style of programming, which we call *change-oriented programming* (ChOP). ChOP is introduced in Chapter 3.

It centralises change as the main development entity and can be applied to programming paradigms such as object-oriented programing. When developing in a ChOP way, developers have to instantiate and apply changes in order to develop their software systems. A software system in turn, is specified by the sequence of changes that was applied to produce it.

The goal of Chapter 4 is to establish a model that allows expressing the evolution of a computer application as first-class objects. We start out from FAMIX: a model which captures the common features of different object-oriented programming languages needed for software re-engineering activities [27, 30, 105]. We create a model of first-class change classes which is based on the FAMIX model and incorporate the notion of dependencies between different change operations in the model. The result is an evolution model: a model that can be used to express the evolution of software programs written in one of the programming languages adhering to FAMIX.

Bottom-up feature-oriented programming ChOP enables a bottom-up approach to feature-oriented programming (FOP). In Chapter 5, we explain the details of this approach and clarify it with a small example: a Buffer. We present three techniques of capturing changes, a classification model, three classification techniques of classifying changes and an algorithm for composing and validating change compositions. We introduce the concept of a *change specification*: a definition of a software product family based on our model of first-class changes. Throughout the chapter, we use the Buffer case to exemplify all aspects of the approach.

In Chapter 6, we present a formalism of the model behind ChOP. The formalism is based on basic set theory and presents the fundamental concepts and some properties that hold in the context of ChOP. Afterwards, we show that this formalism can be mapped to the better known formalism of feature diagrams and present an algorithm that is capable of translating a change specification to a feature diagram. This formal mapping opens up a broad range of applications that were verified in the feature diagram research domain.

Multi-dimensional modularisation Many software systems suffer from a tyranny of dominant decomposition: they can be modularized in only one way at a time, and the many kinds of concerns that do not align with that modularization end up scattered across many modules and tangled with one another. We claim that every program can be modularised in at least two ways: with respect to the separation of problem domain concerns and with respect to the separation of solution domain concerns. In many cases, those modularisations do not match, which makes the tyranny of dominant decomposition problem very apparent.

We target this problem in the context of feature-oriented programming in Chapter 7. We explain the extension of the ChOP model with the notion of *flexible* features, which consist of at least one optional change that does not have to be included in a composition in order to make it valid. We show that such features provide more flexibility with respect to compositions and that they allow for

more than one composition strategy. We expose a weakness of our change model which is caused by the fact that features always consist of explicit enumerations of change objects. We present a solution for that issue, and call it *intensional changes*: descriptive changes which can evaluate to an enumeration of changes. We explain how such changes can be used to model crosscutting concerns and show how they increase robustness to variability.

In order to validate our work, Chapter 8 presents a proof-of-concept implementation of the evolution model and unveils ChEOPS. It is a tool that supports two techniques of recording modularisation information. First, it allows a developer to apply ChOP in a context of object-oriented software development. All kinds of first-class change objects defined by the evolution model can be instantiated from within ChEOPS. Second, ChEOPS supports logging. For that, we instrumented an interactive development environment in such a way that change objects are created whenever a development action is performed. ChEOPS, includes a graphical user interface which allows visualising and reasoning over compositions of change objects. The chapter concludes with an evaluation of the implementation of a text editor called FOText, which we developed using our bottom-up approach. We show how ChEOPS can be used to validate and produce variations of FOText and discuss some benchmarks related to the production of a handfull of variations.

10.2 Contributions

We conclude by listing the contributions we made while doing the research for this dissertation:

- **Evolution model** – The dissertation includes a presentation of a generic model that can be used to encapsulate the information related to the evolution of object-oriented class-based systems. This model encapsulates development operations as change objects: first-class entities that each denote a development action. The evolution model is based on an extended FAMIX model and is capable of capturing and reproducing the evolution of any software program written in a programming language that adheres to FAMIX (such as Java, Smalltalk or C++). The model is capable of expressing changes down to the level of statements, allows the declaration of changes that add, modify or delete program building blocks and supports the management of dependencies between the change objects.

- **Classification model and techniques** – The software classification model provides simple concepts for organising software systems in manageable modules that can afterwards be recomposed. The software classification techniques provide strategies to set up and recover those modules. Three classification strategies are presented in this dissertation: manual classification, semi-automatic classification based on clustering and automatic classification through development action annotations.

  Very important in this work is the integration of software classification in the software development environment and in the software development process.

The results of the classification are tangible in the development environment and they can be used in subsequent software engineering activities such as the composition, the validation or the debugging of different program variations.

- **Change-oriented programming** – Change-oriented programming is a novel programming style which centralises change as the main development entity. In pure change-oriented programming, developers use an interactive development environment to instantiate a change for every development action they want to take. In order to instantiate a change, the interactive development environment first collects all the required information by means of interactive dialogs and then instantiates the corresponding change class. Afterwards, the change instance can be applied in order to carry out the captured development action. The change instances are maintained and can be replayed in order to reproduce the developed software system.

  As pure change-oriented programming does not seem realistic, we propose a watered-down alternative in which only the coarse grained actions (such as the creation of classes) are instantiated by means of interactive dialogs. The more frequent and fine-grained development actions (such as the addition of a statement to a method body) are automatically logged by the interactive development environment without it to fire dialogs that request information. Instead, it detects such fine-grained development actions whenever a developer carries them out and automatically instantiates the corresponding change objects behind the scenes.

- **Bottom-up approach to feature-oriented programming** – The main contribution of this dissertation is the novel approach to bottom-up feature-oriented programming. It provides developers with an alternative for the various state of the art top-down approaches to feature-oriented programming that does not suffer from limitations such as the limited control over feature modularisation or the specific development process required to do feature-oriented programming. Our bottom-up approach to feature-oriented programming provides more control over feature modularity than the state of the art as it allows for a feature to express building block adaptations (including the removal) down to the level of granularity of code statements. Moreover, it does not require a developer to alter his development process for doing feature-oriented programming.

  The approach is based on capturing relevant modularisation information at development time. We use that information to automatically modularise software from a point of view of the problem domain (in feature modules) and support program variation by automatically validating and generating software compositions that stem from lists of required functionality. Moreover, it supports a customised feature deployment: a deployment mechanism that automatically customises features in such a way that they become combinable with the other features in a feature composition.

- **Change- and evolution-oriented programming support** – As a technical contribution of our dissertation, we implemented the ChEOPS tool suite. This research prototype is a concrete instantiation of the evolution and classification models. It supports logging and change-oriented programming as change gathering techniques. It allows a developer to classify changes in an manual, semi-autoamtic and automatic way and includes an implementation of an algorithm that computes the maximal change composition. By means of this algorithm, developers can specify and automatically generate different program variations.

  Our tool suite offers the developer the SOUL logic language and the Smalltalk language as a means to express intensional changes. This enables ChEOPS to fully support our bottom-up approach. In ChEOPS, the changes and their classification are all first-class entities that are accessible by other software engineering tools.

This dissertation has investigated modularisation of software systems around the functionality they provide. We have distilled conceptual and gathered experimental evidence that the state-of-the-art approaches to feature-oriented programming only provide support for bottom-up modularisation, provide limited control over feature modularisation and require practitioners to alter their development process. We have developed a novel bottom-up approach to feature-oriented programming which takes away those restrictions. We used this approach to show that **software can be automatically restructured in feature modules if it is developed in a development environment that records fine-grained modularisation information resulting from development actions**.

In order to validate this thesis, we developed ChEOPS: a proof-of-concept implementation that supports the three phases of our bottom-up approach to feature-oriented programming: the collection, the classification and composition of first-class change objects. ChEOPS records fine-grained modularisation information resulting from the development actions. By means of this tool, we developed a text editor in a standard object-oriented way and showed that it could be automatically modularised into feature modules. Afterwards, we recomposed those feature modules to construct variations of the text editor that offer different combinations of functionality. Finally, we showed that the granularity of the change objects allows us to validate the compositions and that it provides detailed information about why some compositions are not allowed.

Bibliography

[1] M. Akşit and B. Tekinerdoğan. Aspect-oriented programming using composition filters. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology, ECOOP'98 Workshop Reader*, page 435. Springer Verlag, 1998.

[2] Gentzane Aldekoa, Salvador Trujillo, Goiuria Sagardui Mendieta, and Oscar Díaz. Quantifying maintainability in feature oriented product lines. In *CSMR*, pages 243–247, 2008.

[3] S. Apel, C. Lengauer, D. Batory, B. Möller, and C. Kästner. An algebra for feature-oriented software development. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST)*. Springer-Verlag, 2007.

[4] Sven Apel, Thomas Leich, Marko Rosenmüller, and Gunter Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In Robert Glück and Michael R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2005.

[5] Don Batory. Intelligent components and software generators. Technical report, University of Texas at Austin, Austin, TX, USA, 1997.

[6] Don Batory and Bart J. Geraci. Validating component compositions in software system generators. In *ICSR '96: Proceedings of the 4th International Conference on Software Reuse*, page 72, Washington, DC, USA, 1996. IEEE Computer Society.

[7] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proceedings of the 25th International Conference on Software Engineering*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The genvoca model of software-system generators. *IEEE Softw.*, 11(5):89–94, 1994.

[9] Don S. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th Int. Software Product Line Conference (SPLC)*, pages 7–20, 2005.

[10] Don S. Batory. A tutorial on feature oriented programming and the ahead tool suite. In *GTTSE*, pages 3–35, 2006.

[11] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated Reasoning on Feature Models. *Proceedings of the 17th International Conference (CAiSE'05) LNCS, Advanced Information Systems Engineering.*, 3520:491–503, 2005.

[12] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, 2001.

[13] Thomas Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer-Human Interaction*, 1(3):269 – 294, September 1994.

[14] Koen Bertels, Philip Vanneste, and Carlos De Backer. A cognitive model of programming knowledge for procedural languages. In *ICCAL '92: Proceedings of the 4th International Conference on Computer Assisted Learning*, pages 124–135, London, UK, 1992. Springer-Verlag.

[15] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1997.

[16] Gilad Bracha and William Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM.

[17] John Brant and Don Roberts. Refactoring browser. Technical report, http://wiki.cs.uiuc.edu/RefactoringBrowser, 1999.

[18] Rod Burstall. Christopher strachey—understanding programming languages. *Higher Order Symbol. Comput.*, 13(1-2):51–55, 2000.

[19] P. Centonze, R.J. Flynn, and M. Pistoia. Combining static and dynamic analysis for automatic identification of precise access-control policies. *23th Computer Security Applications Conference*, 2007.

[20] Alistair Cockburn. *Agile software development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[21] Krzystof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

[22] Krzysztof Czarnecki, Ulrich W. Eisenecker, and Patrick Steyaert. Beyond objects: Generative programming. In *Proceedings of the AOP Workshop colocated with ECOOP 1997*, 1997.

[23] Kris De Volder. Aspect-oriented logic meta programming. In *Workshop on Aspect Oriented Programming*, 1998.

[24] Kris De Volder. *Type-Oriented Logic Meta Programming*. Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, September 1998.

[25] Kris De Volder and Theo D'Hondt. Aspect-oriented logic meta programming. In P. Cointe, editor, *Meta-Level Architectures and Reflection, 2nd Int'l Conf. Reflection*, volume 1616 of *LNCS*, pages 250–272. Springer Verlag, 1999.

[26] Kris De Volder, Tom Tourwé, and Johan Brichau. Logic meta programming as a tool for separation of concerns. In *Workshop on Aspect Oriented Programming*, 2000.

[27] S. Demeyer, S. Tichelaar, and P. Steyaert. FAMIX 2.0 - the FAMOOS information exchange model. Technical report, University of Berne, 1999.

[28] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with changeboxes. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 25–49, New York, NY, USA, 2007. ACM.

[29] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, pages 173–188, October 2002.

[30] S. Ducasse and S. Demeyer. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Bern, 1999.

[31] P. Ebraert, E. Van Paesschen, and T. D'Hondt. Change-oriented round-trip engineering. Technical report, Vrije Universiteit Brussel, 2007.

[32] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen, and Theo D'Hondt. Change-oriented software engineering. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 3–24, New York, NY, USA, 2007. ACM.

[33] Peter Ebraert, Ellen Van Paesschen, and Theo D'Hondt. Change-oriented round-trip engineering. In *Atelier RIMEL: Rapport de recherche (VAL-RR2007-01)*, 2007.

[34] Don Batory Ed Jung, Chetan Kapoor. Automatic code generation for actuator interfacing from a declarative specification. In *International Conference on Intelligent Robots and Systems. (IROS 2005). 2005 IEEE/RSJ*, pages 2839 – 2844, 2005.

[35] W. Keith Edwards, Takeo Igarashi, Anthony LaMarca, and Elizabeth D. Mynatt. A temporal model for multi-level undo and redo. In *UIST*, 2000.

[36] Niklas Eén and Niklas Sörensson. An extensible sat-solver. *Theory and Applications of Satisfiability Testing*, pages 502–518, 2004.

[37] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing aspects of aop. *Communications of the ACM*, 44(10):33–38, October 2001.

[38] Johan Fabry and Tom Mens. Language independent detection of object-oriented design patterns. *Computer Languages, Systems and Structures,*, 30(1-2):21–33, April 2004.

[39] François Fleuret. Fast binary feature selection with conditional mutual information. *J. Mach. Learn. Res.*, 5:1531–1555, 2004.

[40] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[41] Andreas Gal, Wolfgang Schroeder-Preikschat, and Olaf Spinczyk. AspectC++: Language proposal and prototype implementation. In OOPSLA-AOP01 [80].

[42] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[43] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Bern, 2005.

[44] Object Management Group. Unified modeling language 1.3. Technical report, Rational Software Corporation, June 1999.

[45] Kris Gybels. Aspect-oriented programming using a logic meta programming language to express cross-cutting through a dynamic joinpoint structure. Bachelors thesis, Programming Technology Lab, Vrije Universiteit Brussel, August 2001.

[46] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, 2005.

[47] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C+*. Addison-Wesley, 1999.

[48] Michael Van Hilst and David Notkin. Using role components to implement collaboration-based designs. In *Proceedings of the 11th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 359–369, 1996.

[49] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), March 2008.

[50] Tim Howard and Adele Goldberg. *VisualWorks - Application Developer's Guide*. Cincom Systems, 1993-2005.

[51] Jim Hugunin. The next steps for aspect-oriented programming languages. Technical report, Xerox Palo Alto Research Center, 2001.

[52] Guillermo Jiménez-Pérez and Don Batory. Memory simulators and software generators. In *SSR '97: Proceedings of the 1997 symposium on Software reusability*, pages 136–145, New York, NY, USA, 1997. ACM.

[53] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, Carnegie Mellon University, November 1990.

[54] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[55] Andy Kellens. *Maintaining the causality between design regularities and source code*. PhD thesis, Vrije Universiteit Brussel, 2007.

[56] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In Dave Thomas, editor, *Proceedings of the 20th European Conference on Object-Oriented Programming*, volume 4067. Springer Verlag, 2006.

[57] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with AspectJ. *Comm. ACM*, 44(10):59–65, 2001.

[58] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and Willian G. Griswold. An overview of aspectJ. In *Proceedings of the European Conference on Object-Oriented Programming, Lecture Notes in Computer Science,*, volume 2072, pages 327 – 353. Springer Verlag, 2001. http://aspectj.org.

[59] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer-Verlag, June 1997.

[60] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM.

[61] Milan Kratochvíl and Charles Carson. *Growing Modular. Mass Customization of Complex Products, Services and Software*. Springer, March 2005.

[62] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Comm. ACM*, 21(7), 1978.

[63] M. M. Lehman and B. Belady. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[64] M. M. Lehman and J. F. Ramil. Software evolution and software evolution processes. *Annals of Software Engineering*, 14(1-4):275–309, 2002.

[65] Karl Lieberherr, Doug Orleans, and Johan Ovlinger. Aspect-oriented programming with adaptive methods. *Comm. ACM*, 44(10):39–41, 2001.

[66] Adrian Lienhard, Stéphane Ducasse, and Tudor Gîrba. Object flow analysis - taking an object-centric view on dynamic analysis. In *Proceedings of International Conference on Dynamic Languages*, pages 121 –140, 2007.

[67] Jia Liu, Don Batory, and Srinivas Nedunuri. Modeling interactions in feature oriented software designs. In Stephan Reiff-Marganiec and Mark Ryan, editors, *FIW*, pages 178–197. IOS Press, 2005.

[68] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 68–77, New York, NY, USA, 2006. ACM.

[69] Roberta Mancini, Alan Dix, and Stefano Levialdi. Reflections on undo. Technical report, Dipartimento di Scienze dell'Informazione, Universita degli Studi di Roma "La Sapienza", Via Salaria 113, 00198, Rome, Italy, 1996.

[70] Tomi Männistö, Timo Soininen, and Reijo Sulonen. Modeling configurable products and software product families. In *in Proc. of the International Joint Conference on Artificial Intelligence (IJCAI-2001) - Workshop on Configuration*, 2001.

[71] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[72] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Journal on Expert Systems with Applications*, 23(4):405–413, 2002.

[73] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[74] Andreas Metzger, Patrick Heymans, Klaus Pohl, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE'07)*, pages 243–253, New Delhi, India, October 2007.

[75] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: an agile information visualization framework. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 135–144, New York, NY, USA, 2006. ACM.

[76] R.T. Mittermeir. Facets of software evolution, 2006.

[77] Sathit Nakkrasae and Peraphon Sophatsathit. A formal approach for specification and classification of software components. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 773–780, New York, NY, USA, 2002. ACM.

[78] OMG. Omg unified modeling language (omg uml) infrastructure. Technical Report formal/2009-02-04, OMG, 2009.

[79] OMG. Omg unified modeling language (omg uml), superstructure. Technical Report formal/2009-02-02, OMG, 2009.

[80] *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, 2001.

[81] Ro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *In Proceedings of the International Conference on Software Engineering*, pages 491–500, 2004.

[82] Harold Ossher and Peri Tarr. Using subject-oriented programming to overcome common problems in object-oriented software development/evolution. In *Proc. 21st Int'l Conf. Software Engineering*, pages 687–688. IEEE Computer Society Press, 1999.

[83] Ellen Van Paesschen. *Advanced Round-Trip Engineering.* PhD thesis, Vrije UNiversiteit Brussel, 2006.

[84] Andy Podgurski and Lori A. Clarke. A formal model of program dependencies and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(9):965 – 979, 1990.

[85] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, 2005.

[86] Klaus Pohl, Gunter Bockle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer, July 2005.

[87] Atul Prakash. Undoing actions in collaborative work. Technical Report MI 48109-2122, University Of Michigan, 1992.

[88] Atul Prakash and Michael J. Knister. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction*, 1(4):295 – 330, 1994.

[89] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. *Lecture Notes in Computer Science*, 1241:419–434, 1997.

[90] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358 – 366, March 1953.

[91] Romain Robbes and Michele Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, pages 93–109, 2007.

[92] Charles Romesburg. *Cluster Analysis for Researchers*. Wadsworth, 2004.

[93] A. Rosdal. Empirical study of software evolution and architecture in open source software projects. Technical report, Norwegian University of Science and Technology, 2005.

[94] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53, New York, NY, USA, 2001. ACM.

[95] Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks (2006), special issue on feature interactions in emerging application domains*, page 38, 2006.

[96] Nathanael Shärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. Technical report, Oregon Graduate Institute School of Science and Engineering, 2002.

[97] Cincom's Smalltalk. *Tool Guide*. Cincom Systems Inc, 2007.

[98] Yannis Smaragdakis and Don Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[99] Maximilian Stoerzer and Juergen Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 653–656, Washington, DC, USA, 2005. IEEE Computer Society.

[100] Maximilian Störzer and Christian Koppen. Pcdiff: Attacking the fragile pointcut problem, abstract. In *European Interactive Workshop on Aspects in Software*, Berlin, Germany, September 2004.

[101] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley, January 1998.

[102] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, New York, NY, USA, 1999. ACM.

[103] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104, New York, NY, USA, 2007. ACM.

[104] The Eclipse Corporation. Eclipse. http://eclipse.org, 2007.

[105] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, 2001.

[106] Laurence Tratt and Roel Wuyts. Guest editors' introduction: Dynamically typed languages. *IEEE Software*, 24(5):28–30, 2007.

[107] University of Illinois at Urbana-Champaign. Visualworks: Change list tool. http://wiki.cs.uiuc.edu/VisualWorks/Change+List+Tool, 2007.

[108] Tijs van der Storm. Generic feature-based composition. In Markus Lumpe and Wim Vanderperren, editors, *Proceedings of the Workshop on Software Composition (SC'07)*, volume 4829 of *LNCS*. Springer, 2007.

[109] Yves Vandewoude. *Dynamically updating component-oriented systems*. PhD thesis, Katholieke Universiteit Leuven, 2007.

[110] John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors. *Pattern Languages of Program Design 2*. Addison–Wesley, Reading, MA, 1996.

[111] Roel Wuyts. *A logic meta-programming approach to support the co-evolution of Object-Oriented design and implementation*. Phd thesis, Programming Technology Lab, Vrije Universiteit Brussel, January 2001.

[112] Roel Wuyts. Roeltyper, a fast type reconstructor for smalltalk. Technical report, Université Libre de Bruxelles, 2005.

[113] Z. Xing and E. Stroulia. Umldiff: An algorithm for object-oriented design differencing. In *Proceedings of the 20th International Conference on Automated Software Engineering*, 2005.

List of Publications

Articles in international reviewed journals

1. Yves Vandewoude, Peter Ebraert, Yolande Berbers and Theo D'Hondt. *Tranquility: a low disruptive alternative to quiescence for ensuring safe dynamic updates.* In ”Transactions On Software Engineering”, Volume 33, number 12 (1), published by IEEE Computer Society, 2007.

Contributions at international conferences, reviewed and published in proceedings

1. Peter Ebraert, Andreas Classen, Patrick Heymans and Theo D'Hondt. *Feature Diagrams for Change-Oriented Programming*, In the 10th book in the series on ”Feature Interactions in Software and Communication Systems”, published by IOS Press, 2009

2. Peter Ebraert, Jorge Antonio Vallejos Vargas, Yves Vandewoude, Yolande Berbers and Theo D'Hondt. *Flexible features: Making feature modules more reusable*, In ”Proceedings of the 24th Annual ACM Symposium on Applied Computing”, published by ACM, 2009

3. Peter Ebraert. *First-class change objects for feature-oriented programming*, In ”Proceedings of the 15th Working Conference on Reverse Engineering”, published by IEEE Computer Society, 2008

4. Peter Ebraert, Jorge Antonio Vallejos Vargas, Pascal Costanza, Ellen Van Paesschen and Theo D'Hondt. *Change-Oriented Software Engineering*, In ”Proceedings of the 2007 International Conference on Dynamic Languages”, published by ACM, 2007

5. Jorge Antonio Vallejos Vargas, Peter Ebraert, Brecht Desmet, Tom Van Cutsem, Stijn Mostinckx and Pascal Costanza. *The Context-Dependent Role Model*, In ”Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems”, published by Springer Verlag, 2007

6. Yves Vandewoude, Peter Ebraert, Yolande Berbers and Theo D'Hondt. *An alternative to quiescence: tranquility*, In "Proceedings of the 22th International Conference on Software Maintenance", published by IEEE Computer Society, 2006

7. Kris Steenhaut, Peter Ebraert, Jes Fink-jensen and Ann Nowe. *Introducing elements of knowledge management for E-learning*, In "Proceedings of the IADIS International Conference WWW/Internet 2002, Lisbon." published by IADIS, 2002

Workshop papers and local publications

1. Peter Ebraert and Theo D'Hondt. *On the classification of first-class changes*, In "Proceedings of the 7th BElgian-NEtherlands software eVOLution workshop", published digitally, 2008

2. Peter Ebraert, Leonel Merino and Theo D'Hondt. *Software variation by means of first-class change objects*, In "Proceedings of the Software Variability: a Programmers' Perspective symposium", published digitally, 2008

3. Peter Ebraert and Theo D'Hondt. *Feature-oriented programming based on first-class changes*, In "2nd Workshop on FAMIX and Moose in Reengineering", published digitally, 2008

4. Peter Ebraert and Theo D'Hondt. *A Meta-model for expressing first-class changes*, In "Proceedings of the Third International ERCIM Workshop on Software Evolution", published by ERCIM, 2007

5. Peter Ebraert, Ellen Van Paesschen and Theo D'Hondt. *Change-Oriented Round-Trip Engineering*, In "Proceedings of the RIMEL workshop", published by ACM, 2007

6. Peter Ebraert and Olivier Le Goar. *Evolution styles: change patterns for Software Evolution*, In "Proceedings of the Third International ERCIM Workshop on Software Evolution", published by ERCIM, 2007

7. Peter Ebraert, Theo D'Hondt, Yolande Vandewoude and Yolande Berbers. *User-centric dynamic evolution*, In "Proceedings of the International ERCIM Workshop on Software Evolution", published by ERCIM, 2006

8. Jorge Antonio Vallejos Vargas, Peter Ebraert and Brecht Desmet. *A Role-Based Implementation of Context-Dependent Communications Using Split Objects*, In "Proceedings of the workshop on Revival of Dynamic Languages", published digitally, 2006

9. Peter Ebraert and Theo D'Hondt. *Dynamic Refactorings: improving the program structure at runtime*, In "Proceedings of the 3rd Workshop on Reflection, AOP and Meta-Data for Software Evolution", published digitally, 2006

10. Peter Ebraert, Yves Vandewoude, Theo D'Hondt and Yolande Berbers. *Pitfalls in unanticipated dynamic software evolution*, In "Proceedings of the 2rd Workshop on Reflection, AOP and Meta-Data for Software Evolution", published digitally, 2005

11. Peter Ebraert and Eric Tanter. *A Concern-based Approach to Dynamic Software Evolution*, In "Proceedings of the Dynamic Aspects Workshop", published digitally, 2004

12. Peter Ebraert and Tom Tourwe. *A Reflective Approach to Dynamic Software Evolution*, In "Proceedings of the 1st Workshop on Reflection, AOP and Meta-Data for Software Evolution", published digitally, 2004

13. Peter Ebraert, Tom Mens and Theo D'Hondt. *Enabling Dynamic Software Evolution through Automatic Refactorings*, In "Proceedings of the Workshop on Software Evolution Transformations", published digitally, 2004

14. Eric Tanter and Peter Ebraert. *A Flexible Approach to Interactive Runtime Inspection*, In "1st Workshop on Advancing the State-of-the-Art in Runtime Inspection", published digitally, 2003

Technical reports and posters

1. Peter Ebraert, Yves Vandewoude, Yolande Berbers and Theo D'Hondt. *Influence of type systems on dynamic software evolution.* In "Technical Report CW410, KULeuven, Belgium", 2005

2. Yves Vandewoude, Peter Ebraert, Theo D'Hondt and Yolande Berbers. *Influence of type systems on dynamic software evolution*, In "Poster proceedings of the International Conference on Software Maintenance", published by IEEE Computer Society, 2005

3. Kris Steenhaut, Peter Ebraert, Jes Fink-jensen and Ann Nowe. *Introducing Elements Of Knowledge Management For E-learning*, In "Proceedings of the IADIS International Conference", published by IADIS, 2002

Biography

Peter Ebraert was born on January 14, 1980 in Brussels, Belgium. He holds a licentiate's degree of Science in Applied Computer Science, a European Master in Object-Oriented Software Engineering degree and a Master in Business Administration – all from the Vrije Universiteit Brussel. He graduated cum laude in July 2001 with a thesis titled "*Program suggestions based on Community-Profiles*", supervised by Prof. Theo D'Hondt. The same year, he started working in the IT sector while pursuing his economics studies in the evening. In July 2002, he obtained his second diploma. After handing in a thesis titled "*Tool Support for Partial Behavioral Reflection*" – which was again supervised by Prof. Theo D'Hondt – he obtained his third diploma with magna cum laude. In January 2004, Peter started working as a researcher in the PROG (programming technology) research group at the VUB department of Computer Science and was funded by a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). From January 2008 and on, he worked as a teaching assistent at the VUB teaching several courses to first and second bachelor students in computer science.