Towards More Robust Advice: Message Flow Analysis for Composition Filters and its Application

A thesis submitted for the degree
of Master of Science at
the University of Twente

A.J. de Roo

Enschede, March 29, 2007

Graduation committee:
Prof. dr. ir. M. Akşit
Dr. ir. L.M.J. Bergmans
Ir. P.E.A. Dürr
Ir. T. Staijen

Twente Research and Education
on Software Engineering
Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science
University of Twente

TRESE
Twente Research & Education
on Software Engineering

University of Twente
Enschede - The Netherlands

Abstract

Aspect oriented programming improves the quality of software by allowing a better separation of concerns. Composition filters is a delegation based AOP approach. It introduces advice by filtering messages sent between objects.

The declarative syntax of composition filters opens possibilities for powerful reasoning about the behavior of a set of filters. This reasoning includes control flow analysis. But control flow analysis is only the basis of a much more powerful reasoning technique, called message flow analysis. Message flow analysis reasons about the behavior of a filter set for a specific message. It is a combination of control flow analysis and data flow analysis on the message entity.

Message flow analysis brings opportunities for powerful conflict detection techniques, analyzing concern signature modifications, inlining of a set of filters and more.

This thesis presents a new approach to message flow analysis, called the message flow simulation approach. This new approach improves upon existing approaches, like the message-action tree approach, by providing better granularity, traceability and efficiency.

This thesis also works out four different applications of filter reasoning. First, it explains how consistency reasoning can do better reachability analysis and how the results from message flow analysis are used to create a cause and effect relationship between consistency conflicts. Second, it explains how message flow analysis is used to analyze signature modification. Third, it explains how message flow analysis makes behavioral reasoning more precise and more efficient. Finally, it explains how message flow analysis is used to translate a filter set to executable code, which can be woven in the base program.

Acknowledgements

With the finishing of my thesis my graduation project has come to an end. There are a number of people that supported me during my graduation project, for which I am grateful.

First, I would like to thank the members of my graduation committee. I would like to thank Lodewijk Bergmans, my supervisor, for being an example of a true scientist to me. With his always critic eye and expert knowledge he was of great guidance to me. I also would like to thank Pascal Dürr. He inspired me with his never ending enthusiasm about composition filters and all its facets. I would like to thank him for always being available for answering questions, giving advice and enlightening discussions. Next, I would like to thank Tom Staijen. Although a less prominent member of my committee, his initial work on this subject gave me a good start of my graduation project.

Next, I would like to thank Michiel van Oudheusen and Sverre Boschman for the enjoyable time we had together during the StarLight project and all other Compose* developers for their valuable input.

Last, but not least, I would like to thank my parents and my sister for their love and support during all those years and for their unconditional faith in me.

Arjan de Roo
March 29, 2007
Enschede, The Netherlands

Contents

Abstract **i**

Acknowledgements **iii**

I Introduction **1**

1 Introduction to AOSD **2**
1.1 Introduction 2
1.2 Traditional Approach 4
1.3 AOP Approach 5
1.3.1 AOP Composition 6
1.3.2 Aspect Weaving 6
1.4 AOP Solutions 8
1.4.1 AspectJ Approach 9
1.4.2 Hyperspaces Approach 10
1.4.3 Composition Filters 11

2 Compose⋆ **13**
2.1 Evolution of Composition Filters 13
2.2 Composition Filters in Compose⋆ 14
2.3 Demonstrating Example 16
2.3.1 Initial Object-Oriented Design 17
2.3.2 Completing the Pacman Example 17
2.4 Compose⋆ Architecture 22
2.4.1 Integrated Development Environment 22
2.4.2 Compile Time 22
2.4.3 Adaptation 22
2.4.4 Runtime 22
2.5 Platforms 23
2.6 Features Specific to Compose⋆ 23
2.7 StarLight 24
2.7.1 StarLight Architecture 25
2.7.2 Explicit Modeling of the Returning Flow 25
2.7.3 Defining New Filter Types and Filter Actions 27
2.7.4 Conditional Superimposition 29

3 Introduction to the .NET Framework **30**
3.1 Introduction . . . 30
3.2 Architecture of the .NET Framework . . . 31
3.2.1 Version 2.0 of .NET . . . 32
3.3 Common Language Runtime . . . 33
3.3.1 Java VM vs .NET CLR . . . 34
3.4 Common Language Infrastructure . . . 34
3.5 Framework Class Library . . . 36
3.6 Common Intermediate Language . . . 37

II Message Flow Analysis **40**

4 Motivation **41**
4.1 The Purpose of Filter Reasoning . . . 41
4.1.1 Conflict Analysis and Error Detection . . . 42
4.1.2 Signature Generation . . . 43
4.1.3 Filter Inlining . . . 45
4.1.4 Further Applicability . . . 46
4.2 Problem Description . . . 46
4.3 Requirements . . . 46
4.3.1 Conceptual Requirements . . . 47
4.4 Existing Approaches . . . 48
4.4.1 Logical Expressions Approach . . . 48
4.4.2 Message-Action Tree Approach . . . 50

5 The Message Flow Simulation Approach **52**
5.1 Transforming the Abstract Syntax Tree to a Flowchart . . . 52
5.1.1 Transformation Rules . . . 54
5.2 Simulating the Execution of a Message in the Filter Set . . . 59
5.2.1 Adding a Frame . . . 59
5.2.2 Execution Steps . . . 60
5.2.3 Maintaining State . . . 63
5.3 Simulating with each Possible Message . . . 64
5.3.1 Different Messages that Behave the Same . . . 64
5.3.2 Identifying Equivalence Classes . . . 65
5.3.3 State Space Example . . . 67
5.4 Computational Complexity . . . 68
5.4.1 From AST to Flowchart . . . 68
5.4.2 Simulating the Execution . . . 72
5.4.3 Total Time Complexity . . . 73
5.5 Discussion . . . 74
5.5.1 Evaluating Condition Expressions . . . 74
5.5.2 The Meta Filter Uncertainty . . . 76
5.5.3 Target Matching . . . 77
5.5.4 Message Lists . . . 78

6 The Filter Reasoning Engine **80**
6.1 Filter Reasoning Engine Overview . . . 80
6.2 Using Graphs and Graph Transformations . . . 81
6.2.1 Transforming the AST to a Flowchart . . . 82
6.2.2 Simulating the Flowchart . . . 82
6.2.3 Advantages and Disadvantages of using GROOVE . . . 83
6.3 The Filter Reasoning Engine Preprocessor . . . 83
6.3.1 Overview of the process . . . 84
6.3.2 Class Structure . . . 85
6.3.3 Preprocessing each filter module? . . . 85
6.4 Combining the Information of Different Filter Modules in a Filter Set . . . 87
6.4.1 Combining Different Flowcharts . . . 87
6.4.2 Combining Different Execution Models . . . 88
6.5 The Filter Reasoning Model . . . 93
6.5.1 The FIRE Model Structure . . . 93
6.5.2 The Flowchart Structure . . . 94
6.5.3 The Execution Model Structure . . . 96
6.6 Filter Reasoning Engine Tools . . . 97
6.6.1 Iterators . . . 98
6.6.2 Regular Expression Checker . . . 98
6.6.3 Query Engine . . . 98
6.6.4 Viewer . . . 101
6.7 Implementation issues . . . 101
6.7.1 Is Execution Model Expansion Needed? . . . 101
6.7.2 GROOVE's inefficiency . . . 103

III Message Flow Analysis Applied **107**

7 Consistency Reasoning **108**
7.1 Theory and Concepts . . . 109
7.1.1 Identifying Unreachable Components . . . 109
7.1.2 From Unreachable Component to Conflict . . . 110
7.1.3 Computational Complexity . . . 115
7.2 Design and Implementation . . . 117
7.2.1 Position in the Compose⋆ Architecture . . . 117
7.2.2 Structure of the Consistency Reasoning Engine . . . 117
7.2.3 Implementation of the Consistency Reasoning Algorithm . . . 118
7.2.4 Reporting Conflicts . . . 120

8 Signature Generation **122**
8.1 Theory and Concepts . . . 122
8.1.1 Old Approach to Signature Generation and Type Checking . . . 123
8.1.2 A Different Interpretation of Holljen's Definition . . . 124
8.1.3 Representing the Signature Mapping as a Dependency Graph . . . 125
8.1.4 Using the Dependency Graph to Generate the Signatures . . . 128

8.1.5 Conflicts and Type Errors . . . 132
8.1.6 Using Other Filter Actions as Sufficient Requirement for Inclusion in the Signature . . . 139
8.2 From Theory to Practice . . . 140
8.2.1 Incrementally Building a Sufficient Subset of the Dependency Graph 140
8.2.2 Initialization . . . 141
8.2.3 Start Signatures . . . 141
8.2.4 Final Signatures . . . 154
8.2.5 Type Checking and Conflict Detection . . . 156
8.2.6 Using Other Filter Actions as Sufficient Requirement for Inclusion in the Signature . . . 160
8.3 Design and Implementation . . . 163
8.3.1 Location in the Compilation Process . . . 163
8.3.2 Class Structure of the Signature Generation Engine . . . 163

9 Behavioral Reasoning 166
9.1 Theory and Concepts . . . 166
9.1.1 Introduction to Behavioral Reasoning . . . 167
9.1.2 Using Filter Reasoning for Behavioral Reasoning . . . 167
9.1.3 An Algorithm with Polynomial Time Complexity . . . 168
9.2 Design and Implementation . . . 173
9.2.1 Implemented Regular Expression Language . . . 173
9.2.2 Class Structure of the Regular Expression Matcher . . . 175
9.2.3 Implementation Details . . . 178
9.2.4 Integration with the Behavioral Reasoning Engine . . . 181

10 Filter Inlining 185
10.1 Theory and Concepts . . . 185
10.1.1 Flow Based Inlining . . . 186
10.1.2 Condition Based Inlining . . . 190
10.1.3 Flow Based Inlining with Jump Instructions . . . 193
10.1.4 Flow Based Inlining with Message Conditions . . . 194
10.1.5 Incorporating Conditional Superimposition . . . 195
10.1.6 Weaving Input Filters . . . 197
10.1.7 Weaving Output Filters . . . 201
10.2 Design and Implementation . . . 202
10.2.1 Overview of the Inlining Process . . . 202
10.2.2 The Abstract Instruction Model . . . 202
10.2.3 The Inlining Engine and Model Builder . . . 204
10.2.4 The Emitter: Communication from Java to .NET . . . 205
10.2.5 Weaving the Abstract Instruction Model into Common Intermediate Language Assemblies . . . 208
10.2.6 Filter Inlining vs Runtime . . . 213
10.2.7 Inlining problems . . . 214

IV Conclusion **216**

11 Conclusion **217**
11.1 Discussion . . . 217
11.1.1 Message Flow Analysis . . . 217
11.1.2 Consistency Reasoning . . . 218
11.1.3 Signature Generation . . . 218
11.1.4 Behavioral Reasoning . . . 219
11.1.5 Filter Inlining . . . 219
11.2 Related Work . . . 220
11.2.1 Message Flow Analysis . . . 220
11.2.2 Consistency Reasoning . . . 220
11.2.3 Signature Generation . . . 221
11.2.4 Behavioral Reasoning . . . 221
11.2.5 Filter Inlining . . . 221
11.3 Future Work . . . 222
11.3.1 Message Flow Analysis . . . 222
11.3.2 Consistency Reasoning . . . 222
11.3.3 Signature Generation . . . 223
11.3.4 Behavioral Reasoning . . . 223
11.3.5 Filter Inlining . . . 223
11.4 Contributions . . . 224
11.4.1 Message Flow Analysis . . . 224
11.4.2 Consistency Reasoning . . . 224
11.4.3 Signature Generation . . . 224
11.4.4 Behavioral Reasoning . . . 225
11.4.5 Filter Inlining . . . 225

V Appendices **226**

A Motivation Example **227**
A.1 Base system . . . 227
A.2 LogMail Concern . . . 228
A.3 BufferMail Concern . . . 228
A.4 SecureConnection Concern . . . 229

List of Figures

1.1 Dates and ancestry of several important languages 3

2.1 Components of the composition filters model 15
2.2 Class diagram of the object-oriented Pacman game 18
2.3 Overview of the Compose⋆ architecture 21
2.4 Explicit modeling of the returning flow 26
2.5 Filter actions can either continue, return or exit 26

3.1 Context of the .NET framework 32
3.2 Relationships in the CTS 35
3.3 Main components of the CLI and their relationships 36
3.4 From source code to machine code 37

4.1 Structure of the mail system example 42
4.2 New signature of `Connection` 44
4.3 The message-action tree 50

5.1 Example of an abstract syntax tree 53
5.2 Example of a flowchart 54
5.3 The Transformation rules 55
5.3 The Transformation rules (Continued) 56
5.3 The Transformation rules (Continued) 57
5.3 The Transformation rules (Continued) 58
5.3 The Transformation rules (Continued) 58
5.4 A frame element is used to simulate the execution 59
5.5 Doing an execution step 60
5.6 Branching execution steps 61
5.7 Handling the substitution part 62
5.8 Creating a state space during the simulation 63
5.9 Example of a state space 69
5.10 Execution model with an exponential number of states 79

6.1 Components in the Filter Reasoning Engine 80
6.2 The FIRE preprocessing process 84
6.3 Class structure of the FIRE Preprocessor 86
6.4 The flowchart of a filter module 87
6.5 First step: adding the filter module condition 88

6.6 Combining two flowcharts 89
6.7 The execution models corresponding to the separate filter modules 90
6.8 Adding filter module conditions to the execution model 90
6.9 Extended execution model of the first filter module 92
6.10 The execution models corresponding to the separate filter modules 93
6.11 Combined execution model 94
6.12 The `FireModel` component 95
6.13 The `Flowchart` structure 96
6.14 The `ExecutionModel` structure 97
6.15 The `Iterator` components 98
6.16 The `QueryEngine` components 99
6.17 The internal structure of the `CTLQueryEngine` 100
6.18 The `Viewer` components 101
6.19 The execution models corresponding to the separate filter modules 102
6.20 The combined execution model with and without expansion 102
6.21 Time GROOVE took to do the transformation and simulation for varying sizes of filter sets on a linear scale 104
6.22 Time GROOVE took to do the transformation and simulation for varying sizes of filter sets on a logarithmic scale 105
6.23 Time the FIRE Model took to combine the results of different filter modules in a filter set 106

7.1 Finding unreachable nodes and edges by relating the states and transitions in the execution model back to the flowchart 110
7.2 The cause and effect relation between conflicts 115
7.3 CORE class diagram 117
7.4 Sequence diagram showing how reachable components are found 118
7.5 Sequence diagram showing how unreachable components are identified and checked for conflicts 119

8.1 Partial representation of the dependency graph 128
8.2 The existence of methods resolved in the dependency graph 131
8.3 Cyclic dependency in the dependency graph 133
8.4 An `Unknown` method node does not need to be in the dependency cycle . . 133
8.5 Example of a cyclic dispatch conflict 136
8.6 Example of an infinite signature conflict 137
8.7 Example of a dependency graph with a type error 138
8.8 The rules to create the signature/type set 143
8.9 Class structure of the signature generation engine 163

9.1 Schematic representation of the flowchart 168
9.2 Regular expression matcher public structure 175
9.3 Regular expression matcher internal structure 177
9.4 Schematic representation of the automaton corresponding to regular expression e_1 178
9.5 A single word regular expression 179
9.6 Concatenation of e_1 and e_2 179

9.7 Union of e_1 and e_2 180
9.8 Kleene star operator on e_1 180
9.9 Sequence diagram of SECRET's original analysis phase 182
9.10 Sequence diagram of SECRET's new analysis phase 182
9.11 Regular expression matcher integrated with SECRET 183

10.1 Dividing the state space into filter blocks 186
10.2 Dividing a filter block into filter element blocks 187
10.3 States within a filter element block 188
10.4 Substitution breaks sequential flow 190
10.5 The condition based inlining rules 191
10.6 Substitution leads to parallel flow traces through the filter set 193
10.7 Generate code layer by layer to ensure the correct execution order 195
10.8 Dividing the state space into filter blocks and conditional superimposition states 196
10.9 Approach 1: Using method wrapping to weave input filters 198
10.10 Overriding can lead to problems with method wrapping 199
10.11 Approach 2: Inserting filter code at the beginning of the method body . . 199
10.12 Overriding can lead to problems in the inner call context if only a Boolean value is used 200
10.13 Output filter code replaces the original call 201
10.14 An overview of the inlining process 202
10.15 The abstract instruction model 203
10.16 Structure of the inlining engine and the model builder 204
10.17 FilterCode structure attached to the method or call 205
10.18 The weave specification object model 206

Listings

1.1 Modeling addition, display, and logging without using aspects 4
1.2 Modeling addition, display, and logging with aspects 6
1.3 Example of dynamic crosscutting in AspectJ 9
1.4 Example of static crosscutting in AspectJ 10
1.5 Creation of a hyperspace 11
1.6 Specification of concern mappings 11
1.7 Defining a hypermodule 12
2.1 Abstract concern template 14
2.2 `DynamicScoring` concern in Compose⋆ 19
2.3 Implementation of class `Score` 20
2.4 `DynamicStrategy` concern in Compose⋆ 21
2.5 Defining the `LoggingIn` filter actions 27
2.6 Defining the `Logging` filter type 27
2.7 Conditional superimposition example 29
3.1 Adding example in IL code 38
3.2 Adding example in the C# language 39
3.3 Adding example in the VB.NET language 39
4.1 The `LogMail` concern 42
4.2 The `BufferMail` concern 42
4.3 The superimposed filter set 43
4.4 The `SecureConnection` concern 44
4.5 Inlining of `send` with pure control flow analysis 45
4.6 Inlining of `send` with message specific analysis 45
4.7 Message-action tree example 50
5.1 Example of a filter set 52
5.2 Example of same flow behavior, but different external behavior 65
5.3 Example of same external behavior, but different flow behavior 65
5.4 Example leading to pseudo-equivalence classes 67
5.5 Not taking conditions into account leads to more execution paths 74
5.6 A more elaborate example of how not taking conditions into account leads to more execution paths 74
5.7 An example of a `Meta` action changing a message by using a `Substitution` filter 77
5.8 An example of how instance matching leads to the acceptance of a matching part 77

5.9 An example of a message list filter set that leads to an exponential number of states . . . 78
6.1 The filter modules . . . 88
7.1 Consistency conflicts example . . . 108
7.2 Consistency reasoning complete output . . . 120
7.3 Consistency reasoning primary conflict output . . . 120
7.4 Consistency reasoning leaf conflict + root cause . . . 120
7.5 Consistency reasoning level output . . . 121
8.1 Example of the incomplete signature problem . . . 124
8.2 Example of a type problem . . . 124
8.3 Example to illustrate the dependency graph concept . . . 127
8.4 Example of a cyclic dependency conflict . . . 132
8.5 Example leading to an infinite cyclic dependency conflict set . . . 134
8.6 Example leading to a type finite cyclic dependency conflict set . . . 134
8.7 Example of a paradoxical cyclic dependency conflict . . . 135
8.8 Example of a cyclic dispatch conflict . . . 136
8.9 Example of an infinite signature conflict . . . 137
8.10 Example of a concern with type errors . . . 138
8.11 Example to illustrate why a selector is added to the distinguishable set of the target concern . . . 146
9.1 Example of a filter set with exponentially many execution paths . . . 168
9.2 The regular expression grammar in EBNF . . . 173
10.1 Generating code for filter blocks . . . 187
10.2 Generating code for filter element blocks that might accept as well as reject 189
10.3 Generating code for filter element blocks that always accept . . . 189
10.4 Generating code for filter element blocks that always rejects . . . 189
10.5 filter set leading to large condition expressions . . . 192
10.6 Generated code . . . 192
10.7 Using jump instructions to jump between parallel filter flows . . . 193
10.8 Using message conditions to select between parallel filter flows . . . 194
10.9 Generate code layer by layer to ensure the correct execution order . . . 195
10.10 Generating code for a `FilterModuleCondition` state . . . 196
10.11 Example with conditional superimposition . . . 197
10.12 Using filter context to cope with inner calls . . . 199
10.13 Setting the inner call context . . . 200
10.14 IL structure of the Branch instruction . . . 210
10.15 Executing actions on return . . . 212
10.16 Executing actions on return . . . 212
10.17 Filter module used for the performance measurements . . . 213
11.1 Filter set that leads to a reachability conflicts in the second filter . . . 222
A.1 The `MailSystem` class . . . 227
A.2 The `Connection` class . . . 227
A.3 The `LogMail` concern . . . 228
A.4 The `Logger` class . . . 228
A.5 The `BufferMail` concern . . . 228
A.6 The `MailBuffer` class . . . 229

A.7 The `SecureConnection` concern 229
A.8 The `Encryption` class 230

List of Algorithms

8.1 InitializeSignatures() 141
8.2 StartSignatures() 150
8.3 StartSignature(Concern c) 150
8.4 StartSignatureDistinguishable(Concern c) 150
8.5 StartSignatureClass1(Selector p, ExecutionState state) 151
8.6 StartSignatureClass2(Selector p, ExecutionState state, Set typeSet) 151
8.7 StartSignatureUndistinguishable(Concern c) 152
8.8 StartSignatureClass3(ExecutionState state) 153
8.9 StartSignatureClass4(ExecutionState state, Set signatureSet) 153
8.10 StartSignatureClass6(ExecutionState state, Set signatureSet) 153
8.11 StartSignatureClass7(ExecutionState state, Set signatureSet, Set typeSet) . 153
8.12 FinalSignatures() 154
8.13 FinalSignature(Concern c) 154
8.14 checkDispatchable(Method m, Concern c) 155
8.15 Checking() 156
8.16 CyclicDependencyConflictCheck() 156
8.17 TypeChecking() 157
8.18 CyclicDispatchConflictCheck() 158
8.19 CyclicDispatchConflictCheckInit() 158
8.20 CyclicDispatchConflictCheckProcess() 159
8.21 CyclicDispatchConflictCheckFinal() 160
9.1 Constructing the intersection of two ϵNFA's 181

Part I

Introduction

Chapter 1

Introduction to AOSD

"The superior man cannot be known in little matters,
but he may be entrusted with great concerns.
The small man may not be entrusted with great concerns,
but he may be known in little matters."
Confucius
Chinese philosopher & reformer (551 BC - 479 BC)

The first two chapters have originally been written by seven M. Sc. students [4, 5, 11, 17, 19, 47, 54] at the University of Twente. The chapters have been rewritten for use in the following theses [6, 7, 8, 20, 21, 46, 52, 53]. They serve as a general introduction into Aspect-Oriented Software Development and Compose⋆ in particular.

1.1 Introduction

The goal of software engineering is to solve a problem by implementing a software system. The things of interest are called concerns. They exist at every level of the engineering process. A recurrent theme in engineering is that of modularization: separation and localization of concerns. The goal of modularization is to create maintainable and reusable software. A programming language is used to implement concerns.

Fifteen years ago the dominant programming language paradigm was procedural programming. This paradigm is characterized by the use of statements that update state variables. Examples are Algol-like languages such as Pascal, C, and Fortran.

Other programming paradigms are the functional, logic, object-oriented, and aspect-oriented paradigms. Figure 1.1 summarizes the dates and ancestry of several important languages [56]. Every paradigm uses a different modularization mechanism for separating concerns into modules.

Functional languages try to solve problems without resorting to variables. These languages are entirely based on functions over lists and trees. Lisp and Miranda are examples of functional languages.

A logic language is based on a subset of mathematical logic. The computer is programmed to infer relationships between values, rather than to compute output values from input values. Prolog is currently the most used logic language [56].

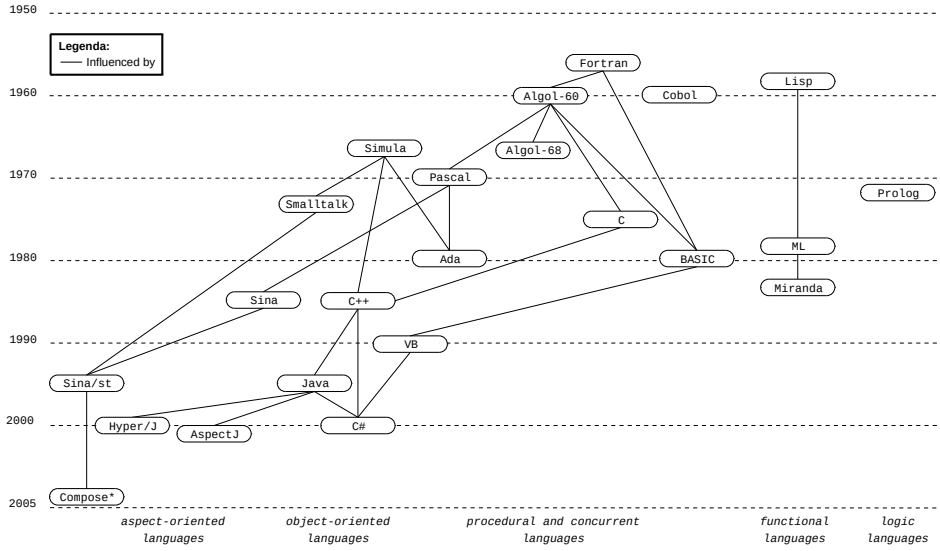


Figure 1.1: Dates and ancestry of several important languages

A shortcoming of procedural programming is that global variables can potentially be accessed and updated by any part of the program. This can result in unmanageable programs because no module that accesses a global variable can be understood independently from other modules that also access that global variable.

The Object-Oriented Programming (OOP) paradigm improves modularity by encapsulating data with methods inside objects. The data may only be accessed indirectly, by calling the associated methods. Although the concept appeared in the seventies, it took twenty years to become popular [56]. The most well known object-oriented languages are C++, Java, C#, and Smalltalk.

The hard part about object-oriented design is decomposing a system into objects. The task is difficult because many factors come into play: encapsulation, granularity, dependency, adaptability, reusability, and others. They all influence the decomposition, often in conflicting ways [14].

Existing modularization mechanisms typically support only a small set of decompositions and usually only a single dominant modularization at a time. This is known as the tyranny of the dominant decomposition [51]. A specific decomposition limits the ability to implement other concerns in a modular way. For example, OOP modularizes concerns in classes and only fixed relations are possible. Implementing a concern in a class might prevent another concern from being implemented as a class.

Aspect-Oriented Programming (AOP) is a paradigm that solves this problem.

AOP is commonly used in combination with OOP but can be applied to other paradigms as well. The following sections introduce an example to demonstrate the problems that may arise with OOP and show how AOP can solve this. Finally, we look at three particular AOP methodologies in more detail.

```java
1  public class Add extends Calculation{
2
3    private int result;
4    private CalcDisplay calcDisplay;
5    private Tracer trace;
6
7    Add() {
8      result = 0;
9      calcDisplay = new CalcDisplay();
10     trace = new Tracer();
11   }
12
13   public void execute(int a, int b) {
14     trace.write("void Add.execute(int,
           int)");
15     result = a + b;
16     calcDisplay.update(result);
17   }
18
19   public int getLastResult() {
20     trace.write("int Add.getLastResult()
           ");
21     return result;
22   }
23 }
```

(a) Addition

```java
1  public class CalcDisplay {
2    private Tracer trace;
3
4    public CalcDisplay() {
5      trace = new Tracer();
6    }
7
8    public void update(int value){
9      trace.write("void CalcDisplay.update
           (int)");
10     System.out.println("Printing new
           value of calculation: "+value);
11   }
12 }
```

(b) CalcDisplay

Listing 1.1: Modeling addition, display, and logging without using aspects

1.2 Traditional Approach

Consider an application containing an object `Add` and an object `CalcDisplay`. `Add` inherits from the abstract class `Calculation` and implements its method `execute(a, b)`. It performs the addition of two integers. `CalcDisplay` receives an update from `Add` if a calculation is finished and prints the result to screen. Suppose all method calls need to be traced. The objects use a `Tracer` object to write messages about the program execution to screen. This is implemented by a method called `write`. Three concerns can be recognized: addition, display, and tracing. The implementation might look something like Listing 1.1.

From our example, we recognize two forms of crosscutting: *code tangling* and *code scattering*.

The addition and display concerns are implemented in classes `Add` and `CalcDisplay` respectively. Tracing is implemented in the class `Tracer`, but also contains code in the other two classes (lines 5, 10, 14, and 20 in (a) and 2, 5, and 9 in (b)). If a concern is implemented across several classes it is said to be scattered. In the example of Listing 1.1 the tracing concern is scattered.

Usually a scattered concern involves code *replication*. That is, the same code is implemented a number of times. In our example the classes `Add` and `CalcDisplay` contain similar tracing code.

In class `Add` the code for the addition and tracing concerns are intermixed. In class `CalcDisplay` the code for the display and tracing concerns are intermixed. If more then one concern is implemented in a single class they are said to be tangled. In our example the addition and tracing concerns are tangled. Also display and tracing concerns are tangled. Crosscutting code has the following consequences:

Code is difficult to change
: Changing a scattered concern requires us to modify the code in several places. Making modifications to a tangled concern class requires checking for side-effects with all existing crosscutting concerns;

Code is harder to reuse
: To reuse an object in another system, it is necessary to either remove the tracing code or reuse the (same) tracer object in the new system;

Code is harder to understand
: Tangled code makes it difficult to see which code belongs to which concern.

1.3 AOP Approach

To solve the problems with crosscutting, several techniques are being investigated that attempt to increase the expressiveness of the OO paradigm. Aspect-Oriented Programming (AOP) introduces a modular structure, the aspect, to capture the location and behavior of crosscutting concerns. Examples of Aspect-Oriented languages are Sina, AspectJ, Hyper/J, and Compose⋆. A special syntax is used to specify aspects and the way in which they are combined with regular objects. The fundamental goals of AOP are twofold [16]: first, to provide a mechanism to express concerns that crosscut other components. Second, to use this description to allow for the separation of concerns.

Join points are well-defined places in the structure or execution flow of a program where additional behavior can be attached. The most common join points are method calls. *Pointcuts* describe a set of join points. This allows us to execute behavior at many places in a program by one expression. *Advice* is the behavior executed at a join point.

In the example of Listing 1.2 the class `Add` does not contain any tracing code and only implements the addition concern. Class `CalcDisplay` also does not contain tracing code. In our example, the tracing aspect contains all the tracing code. The pointcut `tracedCalls` specifies at which locations tracing code is executed.

The crosscutting concern is explicitly captured in aspects instead of being embedded within the code of other objects. This has several advantages over the previous code.

Aspect code can be changed
: Changing aspect code does not influence other concerns;

Aspect code can be reused
: The coupling of aspects is done by defining pointcuts. In theory, this low coupling allows for reuse. In practice reuse is still difficult;

Aspect code is easier to understand
: A concern can be understood independent of other concerns;

Aspect pluggability
: Enabling or disabling concerns becomes possible.

```java
public class Add extends Calculation{
  private int result;
  private CalcDisplay calcDisplay;

  Add() {
    result = 0;
    calcDisplay = new CalcDisplay();
  }

  public void execute(int a, int b) {
    result = a + b;
    calcDisplay.update(result);
  }

  public int getLastResult() {
    return result;
  }
}
```

(a) Addition concern

```aspectj
aspect Tracing {
  Tracer trace = new Tracer();

  pointcut tracedCalls():
    call(* (Calculation+).*(..)) ||
    call(* CalcDisplay.*(..));

  before(): tracedCalls() {
    trace.write(thisJoinPoint.getSignature
        ().toString());
  }
}
```

(b) Tracing concern

Listing 1.2: Modeling addition, display, and logging with aspects

1.3.1 AOP Composition

AOP composition can be either symmetric or asymmetric. In the symmetric approach every component can be composed with any other component. This approach is followed by e.g. Hyper/J.

In the asymmetric approach, the base program and aspects are distinguished. The base program is composed with the aspects. This approach is followed by e.g. AspectJ (covered in more detail in the next section).

1.3.2 Aspect Weaving

The integration of components and aspects is called *aspect weaving*. There are three approaches to aspect weaving. The first and second approach rely on adding behavior to the program, either by weaving the aspect in the source code, or by weaving directly in the target language. The target language can be intermediate language (IL) or machine code. Examples of IL are Java byte code and Common Intermediate Language (CIL). The remainder of this chapter considers only intermediate language targets. The third approach relies on adapting the virtual machine. Each method is explained briefly in the following sections.

1.3.2.1 Source Code Weaving

The source code weaver combines the original source with aspect code. It interprets the defined aspects and combines them with the original source, generating input for the native compiler. For the native compiler there is no difference between source code with and without aspects. Hereafter, the compiler generates an intermediate or machine

language output (depending on the compiler-type).

The advantages of using source code weaving are:

High-level source modification
Since all modifications are done at source code level, there is no need to know the target (output) language of the native compiler;

Aspect and original source optimization
First, the aspects are woven into the source code. Then, the source code is compiled by the native compiler. The produced target language has all the benefits of the native compiler optimization passes. However, optimizations specific to exploiting aspect knowledge are not possible;

Native compiler portability
The native compiler can be replaced by any other compiler as long as it has the same input language. Replacing the compiler with a newer version or another target language can be done with little or no modification to the aspect weaver.

However, the drawbacks of source code weaving are:

Language dependency
Source code weaving is written explicitly for the syntax of the input language;

Limited expressiveness
Aspects are limited to the expressive power of the source language. For example, when using source code weaving, it is not possible to add multiple inheritance to a single inheritance language.

1.3.2.2 Intermediate Language Weaving

Weaving aspects through an intermediate language gives more control over the executable program and solves some issues, as identified in Section 1.3.2.1 on source code weaving. Weaving at this level allows for creating combinations of intermediate language constructs that cannot be expressed at the source code level. Although IL can be hard to understand, IL weaving has several advantages over source code weaving:

Programming language independence
All compilers generating the target IL output can be used;

More expressiveness
It is possible to create IL constructs that are not possible in the original programming language;

Source code independence
Can add aspects to programs and libraries without using the source code (which may not be available);

Adding aspects at load- or runtime
A special class loader or runtime environment can decide and do dynamic weaving. The aspect weaver adds a runtime environment into the program. How and when aspects can be added to the program depend on the implementation of the runtime environment.

However, IL weaving also has drawbacks that do not exist for source code weaving:

Hard to understand
Specific knowledge about the IL is needed;

More error-prone
Compiler optimization may cause unexpected results. Compiler can remove code that breaks the attached aspect (e .g., inlining of methods).

1.3.2.3 Adapting the Virtual Machine

Adapting the virtual machine (VM) removes the need to weave aspects. This technique has the same advantages of intermediate language weaving and can also overcome some of the disadvantages of intermediate language weaving, mentioned in Section 1.3.2.2. Aspects can be added without recompilation, redeployment, and restart of the application [40, 41].

Modifying the virtual machine also has its disadvantages:

Dependency on adapted virtual machines
Using an adapted virtual machine requires that every system should be upgraded to that version;

Virtual machine optimization
People have spend a lot of time optimizing virtual machines. By modifying the virtual machine these optimizations should be revisited. Reintegrating changes introduced by newer versions of the original virtual machine, might have substantial impact.

1.4 AOP Solutions

As the concept of AOP has been embraced as a useful extension to classic programming, different AOP solutions have been developed. Each solution has one or more implementations to demonstrate how the solution is to be used. As described by [12] these differ primarily in:

How aspects are specified
Each technique uses its own aspect language to describe the concerns;

Composition mechanism
Each technique provides its own composition mechanisms;

Implementation mechanism
Whether components are determined statically at compile time or dynamically at run time, the support for verification of compositions, and the type of weaving.

Use of decoupling
Should the writer of the main code be aware that aspects are applied to his code;

Supported software processes
The overall process, techniques for reusability, analyzing aspect performance of aspects, is it possible to monitor performance, and is it possible to debug the aspects.

This section will give a short introduction to AspectJ [24] and Hyperspaces [37], which together with Composition Filters [3] are three main AOP approaches.

```
aspect DynamicCrosscuttingExample {
  Log log = new Log();

  pointcut traceMethods():
    execution(edu.utwente.trese.*.*(..));

  before() : traceMethods {
    log.write("Entering " + thisJointPoint.getSignature());
  }

  after() : traceMethods {
    log.write("Exiting " + thisJointPoint.getSignature());
  }
}
```

Listing 1.3: Example of dynamic crosscutting in AspectJ

1.4.1 AspectJ Approach

AspectJ [24] is an aspect-oriented extension to the Java programming language. It is probably the most popular approach to AOP at the moment, and it is finding its way into the industrial software development. AspectJ has been developed by Gregor Kiczales at Xerox's PARC (Palo Alto Research Center). To encourage the growth of the AspectJ technology and community, PARC transferred AspectJ to an open Eclipse project. The popularity of AspectJ comes partly from the various extensions based on it, build by several research groups. There are various projects that are porting AspectJ to other languages, resulting in tools such as AspectR and AspectC.

One of the main goals in the design of AspectJ is to make it a compatible extension to Java. AspectJ tries to be compatible in four ways:

Upward compatibility
: All legal Java programs must be legal AspectJ programs;

Platform compatibility
: All legal AspectJ programs must run on standard Java virtual machines;

Tool compatibility
: It must be possible to extend existing tools to support AspectJ in a natural way; this includes IDEs, documentation tools and design tools;

Programmer compatibility
: Programming with AspectJ must feel like a natural extension of programming with Java.

AspectJ extends Java with support for two kinds of crosscutting functionality. The first allows defining additional behavior to run at certain well-defined points in the execution of the program and is called the *dynamic crosscutting mechanism*. The other is called the *static crosscutting mechanism* and allows modifying the static structure of classes (methods and relationships between classes). The units of crosscutting implementation are called aspects. An example of an aspect specified in AspectJ is shown in Listing 1.3.

The points in the execution of a program where the crosscutting behavior is inserted

```
aspect StaticCrosscuttingExample {
  private int Log.trace(String traceMsg) {
    Log.write(" --- MARK --- " + traceMsg);
  }
}
```

Listing 1.4: Example of static crosscutting in AspectJ

are called *join points*. A *pointcut* has a set of join points. In Listing 1.3 is `traceMethods` an example of a pointcut definition. The pointcut includes all executions of any method that is in a class contained by package `edu.utwente.trese`.

The code that should execute at a given join point is declared in an advice. Advice is a method-like code body associated with a certain pointcut. AspectJ supports *before*, *after* and *around* advice, which specifies where the additional code is to be inserted. In the example both before and after advice are declared to run at the join points specified by the `traceMethods` pointcut.

Aspects can contain anything permitted in class declarations including definitions of pointcuts, advice and static crosscutting. For example, static crosscutting allows a programmer to add fields and methods to certain classes as shown in Listing 1.4.

The shown construct is called inter-type member declaration and adds a method `trace` to class `Log`. Other forms of inter-type declarations allow developers to declare the parents of classes (super classes and realized interfaces), declare where exceptions need to be thrown, and allow a developer to define the precedence among aspects.

With its variety of possibilities, AspectJ can be considered a useful approach for realizing software requirements.

1.4.2 Hyperspaces Approach

The *Hyperspaces* approach is developed by H. Ossher and P. Tarr at the IBM T.J. Watson Research Center. The Hyperspaces approach adopts the principle of multi-dimensional separation of concerns [37], which involves:

- Multiple, arbitrary dimensions of concerns;
- Simultaneous separation along these dimensions;
- Ability to dynamically handle new concerns and new dimensions of concern as they arise throughout the software life cycle;
- Overlapping and interacting concerns. It is appealing to think of many concerns as independent or orthogonal, but they rarely are in practice.

We explain the Hyperspaces approach by an example written in the *Hyper/J* language. Hyper/J is an implementation of the Hyperspaces approach for Java. It provides the ability to identify concerns, specify modules in terms of those concerns, and synthesize systems and components by integrating those modules. Hyper/J uses byte code weaving on binary Java class files and generates new class files to be used for execution. Although the Hyper/J project seems abandoned and there has not been any update in the code or documentation for a while, we still mention it because the Hyperspaces approach offers a unique AOP solution.

```
Hyperspace Pacman
  class edu.utwente.trese.pacman.*;
```

Listing 1.5: Creation of a hyperspace

As a first step, developers create hyperspaces by specifying a set of Java class files that contain the code units that populate the hyperspace. To do this, you create a hyperspace specification, as demonstrated in Listing 1.5.

Hyper/J will automatically create a hyperspace with one dimension—the class file dimension. A dimension of concern is a set of concerns that are disjoint. The initial hyperspace will contain all units within the specified package. To create a new dimension you can specify concern mappings, which describe how existing units in the hyperspace relate to concerns in that dimension, as demonstrated in Listing 1.6.

The first line indicates that, by default, all of the units contained within the package `edu.utwente.trese.pacman` address the kernel concern of the feature dimension. The other mappings specify that any method named `trace` or `debug` address the logging and debugging concern respectively. These later mappings override the first one.

Hypermodules are based on concerns and consist of two parts. The first part specifies a set of hyperslices in terms of the concerns identified in the concern matrix. The second part specifies the integration relationships between the hyperslices. A hyperspace can contain several hypermodules realizing different modularizations of the same units. Systems can be composed in many ways from these hypermodules.

Listing 1.7 shows a hypermodule with two concerns, kernel and logging. They are related by a `mergeByName` integration relationship. This means that units in the different concerns correspond if they have the same name (`ByName`) and that these corresponding units are to be combined (`merge`). For example, all members of the corresponding classes are brought together into the composed class. The hypermodule results in a hyperslice that contains all the classes without the debugging feature; thus no `debug` methods will be present.

The most important feature of the hyperspaces approach is the support for on-demand remodularisation: the ability to extract hyperslices to encapsulate concerns that were not separated in the original code. Which makes hyperspaces especially useful for evolution of existing software.

1.4.3 Composition Filters

Composition Filters is developed by M. Akşit and L. Bergmans at the TRESE group, which is a part of the Department of Computer Science of the University of Twente, The Netherlands. The composition filters (CF) model predates aspect-oriented programming.

```
package edu.utwente.trese.pacman: Feature.Kernel
operation trace: Feature.Logging
operation debug: Feature.Debugging
```

Listing 1.6: Specification of concern mappings

```
hypermodule Pacman_Without_Debugging
  hyperslices: Feature.Kernel, Feature.Logging;
  relationships: mergeByName;
end hypermodule;
```

Listing 1.7: Defining a hypermodule

It started out as an extension to the object-oriented model and evolved into an aspect-oriented model. The current implementation of CF is Compose⋆, which covers .NET, Java, and C.

One of the key elements of CF is the *message*, a message is the interaction between objects, for instance a method call. In object-oriented programming the message is considered an abstract concept. In the implementations of CF it is therefore necessary to reify the message. This *reified message* contains properties, like where it is send to and where it came from.

The concept of CF is that messages that enter and exit an object can be intercepted and manipulated, modifying the original flow of the message. To do so, a layer called the *interface part* is introduced in the CF model, this layer can have several properties. The interface part can be placed on an object, which behavior needs to be altered, and this object is referred to as *inner*.

There are three key elements in CF: messages, filters, and superimposition. Messages are sent from one object to another, if there is an interface part placed on the receiver, then the message that is sent goes through the input filters. In the filters the message can be manipulated before it reaches the inner part, the message can even be sent to another object. How the message will be handled depends on the filter type. An output filter is similar to an input filter, the only difference is that it manipulates messages that originate from the inner part. The latest addition to CF is superimposition, which is used to specify which interfaces needs to be superimposed on which inner objects.

Chapter 2

Compose⋆

"The difficult part of composition filters is understanding its simplicity."
Lodewijk Bergmans
Dutch scientist (1967 -)

Compose⋆ is an implementation of the composition filters approach. There are three target environments: the .NET, Java, and C. This chapter is organized as follows, first the evolution of Composition Filters and its implementations are described, followed by an explanation of the Compose⋆ language and a demonstrating example. In the third section, the Compose⋆ architecture is explained, followed by a description of the features specific to Compose⋆.

2.1 Evolution of Composition Filters

Compose⋆ is the result of many years of research and experimentation. The following time line gives an overview of what has been done in the years before and during the Compose⋆ project.

1985 The first version of Sina is developed by Mehmet Akşit. This version of Sina contains a preliminary version of the composition filters concept called semantic networks. The semantic network construction serves as an extension to objects, such as classes, messages, or instances. These objects can be configured to form other objects, such as classes, from which instances can be created. The object manager takes care of synchronization and message processing of an object. The semantic network construction can express key concepts like delegation, reflection, and synchronization [28].

1987 Together with Anand Tripathi of the University of Minnesota the Sina language is further developed. The semantic network approach is replaced by declarative specifications and the interface predicate construct is added.

1991 The interface predicates are replaced by the dispatch filter, and the wait filter manages the synchronization functions of the object manager. Message reflection and real-time specifications are handled by the meta filter and the real-time filter [2].

1995 The Sina language with Composition Filters is implemented using Smalltalk [28]. The implementation supports most of the filter types. In the same year, a preprocessor providing C++ with support for Composition Filters is implemented [15].

1999 The composition filters language ComposeJ [57] is developed and implemented. The implementation consists of a preprocessor capable of translating composition filter specifications into the Java language.

2001 ConcernJ is implemented as part of a M. Sc. thesis [45]. ConcernJ adds the notion of superimposition to Composition Filters. This allows for reuse of the filter modules and to facilitate crosscutting concerns.

2003 The start of the Compose⋆ project, the project is described in further detail in this chapter.

2004 The first release of Compose⋆, based on .NET.

2005 The start of the Java port of Compose⋆.

2006 Porting Compose⋆ to C is started.

2006 Start of the StarLight project. This project is described in detail in Section 2.7.

2.2 Composition Filters in Compose⋆

```
concern {
  filtermodule {
    internals
    externals
    conditions
    inputfilters
    outputfilters
  }

  superimposition {
    selectors
    filtermodules
    annotations
    constraints
  }

  implementation
}
```

Listing 2.1: Abstract concern template

A Compose⋆ application consists of concerns that can be divided in three parts: filter module specifications, superimposition, and implementation. A filter module contains the filter logic to filter on incoming or outgoing messages on superimposed objects. Messages have a target, which is an object reference, and a selector, which is a method name. A superimposition part specifies which filter modules, annotations, conditions, and methods are superimposed on which objects. An implementation part contains the class implementation of a concern. How these parts are placed in a concern is shown in Listing 2.1.

The working of a filter module is depicted in Figure 2.1. A filter module can contain input and output filters. The difference between these two sets of filters is that the first is used to filter on incoming messages, while the second is used to filter on outgoing messages. The return of a method is not considered an outgoing message. A filter has three parts: a filter identifier, a filter type, and one or more filter elements. A filter element exists out of an optional condition part, a matching part, and a substitution part. These parts are shown below:

$$\overbrace{stalker\_filter}^{identifier} : \overbrace{Dispatch}^{filter\ type} = \{\overbrace{!pacmanIsEvil}^{condition\ part} =>$$
$$\overbrace{[*.getNextMove]}^{matching\ part}\ \overbrace{stalk\_strategy.getNextMove}^{substitution\ part}\ \}$$

A filter identifier is a unique name for a filter in a filter module. Filters match when both the condition part and the matching part evaluate to true. In the demonstrated filter, every message where the selector is `getNextMove` matches. If an asterisk (`*`) is used in the target, every target will match. If the condition part and the matching part are true, the message is substituted with the values provided in the substitution part. How these values are substituted, and how the message continues, depends on the type of filter used.

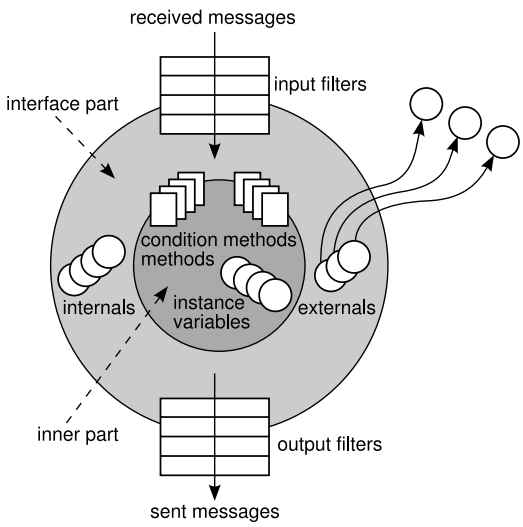


Figure 2.1: Components of the composition filters model

At the moment, there are four basic filter types defined in Compose⋆. It is, however, possible to write custom filter types.

Dispatch If the message is accepted, it is dispatched to the specified target of the message, otherwise the message continues to the subsequent filter. This filter type can only be used for input filters;

Send If the message is accepted, it is sent to the specified target of the message, otherwise the message continues to the subsequent filter. This filter type can only be used for output filters;

Error If the filter rejects the message, it raises an exception, otherwise the message continues to the next filter in the set;

Meta If the message is accepted, the message is sent as a parameter of another meta message to an internal or external object, otherwise the message just continues to the next filter. The object that receives the meta message can observe and manipulate the message and can re-activate the execution of the message.

The identifier `pacmanIsEvil`, used in the condition part, must be declared in the conditions section of a filter module. Targets that are used in a filter can be declared as internal or external. An internal is an object that is unique for each instance of a filter module, while an external is an object that is shared between filter modules.

Filter modules are superimposed on classes using filter module binding, which specifies a selection of objects on the one side, and a filter module on the other side. The selection is specified in a selector definition. This selector definition uses predicates to select objects, such as `isClassWithNameInList`, `isNamespaceWithName`, and `namespaceHasClass`. In addition to filter modules, it is possible to bind conditions, methods, and annotations to classes using superimposition.

The last part of the concern is the implementation part, which can be used to define the behavior of a concern. For a logging concern, for example, we can define specific log functions and use them as internal.

2.3 Demonstrating Example

To illustrate the Compose⋆ toolset, this section introduces a *Pacman* example. The Pacman game is a classic arcade game in which the user, represented by pacman, moves in a maze to eat vitamins. Meanwhile, a number of ghosts try to catch and eat pacman. There are, however, four mega vitamins in the maze that make pacman evil. In its evil state, pacman can eat ghosts. A simple list of requirements for the Pacman game is briefly discussed here:

- One live is taken from pacman when eaten by a ghost;
- A game should end when pacman has no more lives;
- The score of a game should increase when pacman eats a vitamin or a ghost;
- A user should be able to use a keyboard to move pacman around the maze;
- Ghosts should know whether pacman is evil or not;

- Ghosts should know where pacman is located;
- Ghosts should hunt or flee from pacman, depending on the state of pacman.

2.3.1 Initial Object-Oriented Design

Figure 2.2 shows an initial object-oriented design for the Pacman game. Note that this UML class diagram does not show the trivial accessors. The classes in this diagram are:

Game This class encapsulates the control flow and controls the state of a game;

Ghost This class is a representation of a ghost chasing pacman. Its main attribute is a property that indicates whether it is scared or not (depending on the evil state of pacman);

GhostView This class is responsible for painting ghosts;

Glyph This is the superclass of all mobile objects (pacman and ghosts). It contains common information like direction and speed;

Keyboard This class accepts all keyboard input and makes it available to pacman;

Main This is the entry point of a game;

Pacman This is a representation of the user controlled element in the game. Its main attribute is a property that indicates whether pacman is evil or not;

PacmanView This class is responsible for painting pacman;

RandomStrategy By using this strategy, ghosts move in random directions;

View This class is responsible for painting a maze;

World This class has all the information about a maze. It knows where the vitamins, mega vitamins and most importantly the walls are. Every class derived from class `Glyph` checks whether movement in the desired direction is possible.

2.3.2 Completing the Pacman Example

The initial object-oriented design, described in the previous section, does not implement all the stated system requirements. The missing requirements are:

- The application does not maintain a score for the user;
- Ghosts move in random directions instead of chasing or fleeing from pacman.

In the next sections, we describe why and how to implement these requirements in the Compose⋆ language.

2.3.2.1 Implementation of Scoring

The first system requirement that we need to add to the existing Pacman game is scoring. This concern involves a number of events. First, the score should be set to zero when a game starts. Second, the score should be updated whenever pacman eats a vitamin, mega vitamin or ghost. And finally, the score itself has to be painted on the maze canvas to relay it back to the user. These events scatter over multiple classes: `Game` (initializing

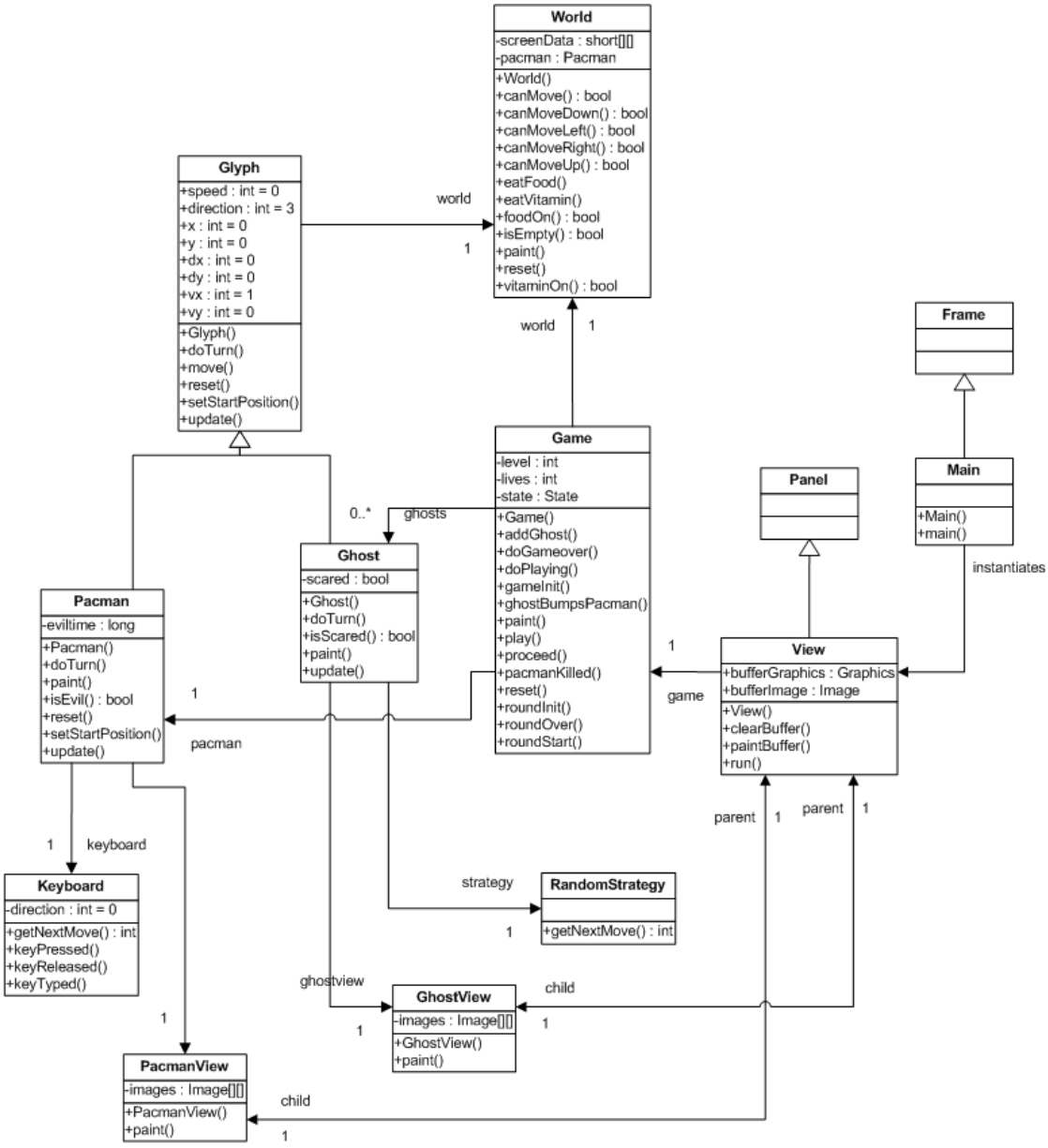


Figure 2.2: Class diagram of the object-oriented Pacman game

score), `World` (updating score), `Main` (painting score). Thus scoring is an example of a crosscutting concern.

To implement scoring in the Compose⋆ language, we divide the implementation into two parts. The first part is a Compose⋆ concern definition stating which filter modules to superimpose. Listing 2.2 shows an example Compose⋆ concern definition of scoring.

```
1  concern DynamicScoring in Pacman {
2    filtermodule dynamicscoring {
3      externals
4        score : pacman.Score = pacman.Score.instance();
5      inputfilters
6        score_filter : Meta = {[*.eatFood] score.eatFood,
7                               [*.eatGhost] score.eatGhost,
8                               [*.eatVitamin] score.eatVitamin,
9                               [*.gameInit] score.initScore,
10                              [*.setForeground] score.setupLabel}
11   }
12   superimposition {
13     selectors
14       scoring = { C | isClassWithNameInList(C, ['pacman.World',
15                                  'pacman.Game', 'pacman.Main']) };
16     filtermodules
17       scoring <- dynamicscoring;
18   }
19 }
```

Listing 2.2: `DynamicScoring` concern in Compose⋆

This concern definition is called `DynamicScoring` (line 1) and contains two parts. The first part is the declaration of a filter module called `dynamicscoring` (lines 2–11). This filter module contains one *meta filter* called `score_filter` (line 6). This filter intercepts five relevant calls and sends the message in a reified form to an instance of class `Score`. The final part of the concern definition is the superimposition part (lines 12–18). This part defines that the filter module `dynamicscoring` is to be superimposed on the classes `World`, `Game` and `Main`.

The final part of the scoring concern is the so-called *implementation part*. This part is defined by a class `Score`. Listing 2.3 shows an example implementation of class `Score` . Instances of this class receive the messages sent by `score_filter` and subsequently perform the events related to the scoring concern. In this way, all scoring events are encapsulated in one class and one Compose⋆ concern definition.

2.3.2.2 Implementation of Dynamic Strategy

The last system requirement that we need to implement is the dynamic strategy of ghosts. This means that a ghost should, depending on the state of pacman, hunt or flee from pacman. We can implement this concern by using the strategy design pattern. However, in this way, we need to modify the existing code. This is not the case when we use Compose⋆ *dispatch filters*. Listing 2.4 demonstrates this.

This concern uses dispatch filters to intercept calls to method `getNextMove` of the class `RandomStrategy`. These calls are redirected to either `StalkerStrategy.getNextMove` or `FleeStrategy.getNextMove`. If pacman is not evil, the intercepted call matches the

```
public class Score
{
  private int score = -100;
  private static Score theScore = null;
  private Label label = new java.awt.Label("Score: 0");

  private Score() {}

  public static Score instance() {
    if(theScore == null) {
      theScore = new Score();
    }
    return theScore;
  }

  public void initScore(ReifiedMessage rm) {
    this.score = 0;
    label.setText("Score: "+score);
  }

  public void eatGhost(ReifiedMessage rm) {
    score += 25;
    label.setText("Score: "+score);
  }

  public void eatVitamin(ReifiedMessage rm) {
    score += 15;
    label.setText("Score: "+score);
  }

  public void eatFood(ReifiedMessage rm) {
    score += 5;
    label.setText("Score: "+score);
  }

  public void setupLabel(ReifiedMessage rm) {
    rm.proceed();
    label = new Label("Score: 0");
    label.setSize(15*View.BLOCKSIZE+20,15*View.BLOCKSIZE);
    Main main = (Main)Composestar.Runtime.FLIRT.message.MessageInfo.
        getMessageInfo().getTarget();
    main.add(label,BorderLayout.SOUTH);
  }
}
```

Listing 2.3: Implementation of class `Score`

first filter, which dispatches the intercepted call to method `StalkerStrategy.getNextMove` (line 9). Otherwise, the intercepted call matches the second filter, which dispatches the intercepted call to method `FleeStrategy.getNextMove` (line 11).

```
concern DynamicStrategy in Pacman {
  filtermodule dynamicstrategy {
    internals
      stalk_strategy : pacman.Strategies.StalkerStrategy;
      flee_strategy : pacman.Strategies.FleeStrategy;
    conditions
      pacmanIsEvil : pacman.Pacman.isEvil();
    inputfilters
      stalker_filter : Dispatch = {!pacmanIsEvil =>
                          [*.getNextMove] stalk_strategy.getNextMove};
      flee_filter : Dispatch = {
                          [*.getNextMove] flee_strategy.getNextMove}
  }
  superimposition {
    selectors
      random = { C | isClassWithName(C,
                          'pacman.Strategies.RandomStrategy') };
    filtermodules
      random <- dynamicstrategy;
  }
}
```

Listing 2.4: `DynamicStrategy` concern in Compose⋆

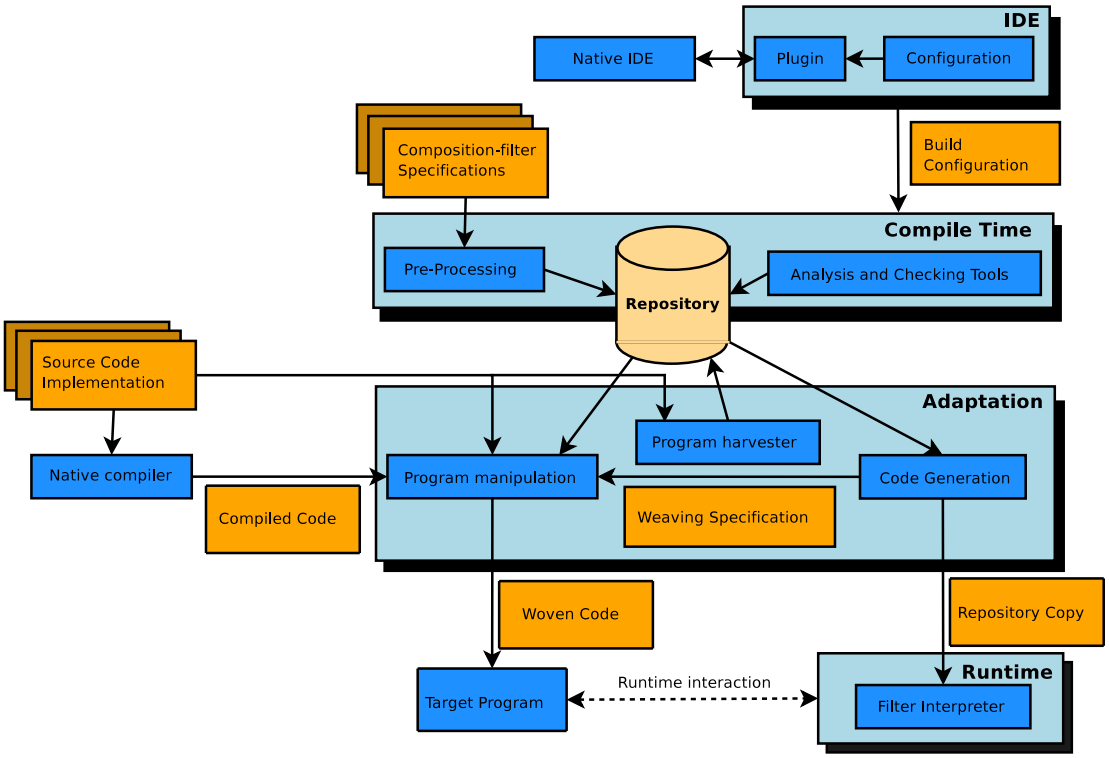


Figure 2.3: Overview of the Compose⋆ architecture

2.4 Compose⋆ Architecture

An overview of the Compose⋆ architecture is illustrated in Figure 2.3. The Compose⋆ architecture can be divided in four layers [36]: IDE, compile time, adaptation, and runtime.

2.4.1 Integrated Development Environment

Some of the purposes of the Integrated Development Environment (IDE) layer are to interface with the native IDE and to create a build configuration. In the build configuration it is specified which source files and settings are required to build a Compose⋆ application. After creating the build configuration the compile time is started.

The creation of a build configuration can be done manually or by using a plug-in. Examples of these plug-ins are the Visual Studio add-in for Compose⋆/.NET and the Eclipse plug-in for Compose⋆/J and Compose⋆/C.

2.4.2 Compile Time

The compile time layer is platform independent and reasons about the correctness of the composition filter implementation with respect to the program. This allows the target program to be build by the adaptation.

The compile time 'pre-processes' the composition filter specifications by parsing the specification, resolving the references, and checking its consistency. To provide an extensible architecture to facilitate this process, a blackboard architecture is chosen. This means that the compile time uses a general knowledge base, which is called the 'repository'. This knowledge base contains the structure and metadata of the program. Different modules can use this knowledge base to execute their activities. Examples of modules within analysis and validation are the three modules SANE, LOLA and FILTH. These three modules are responsible for (some) of the analysis and validation of the super imposition and its selectors.

2.4.3 Adaptation

The adaptation layer consists of the program manipulation, harvester, and code generator. These components connect the platform independent compile time to the target platform. The harvester is responsible for gathering the structure and the annotations within the source program and adding this information to the knowledge base. The code generation generates a reduced copy of the knowledge base and the weaving specification. This weaving specification is then used by the weaver, which is contained in the program manipulation component, to weave in the calls to the runtime into the target program. The end result of the adaptation layer is the woven target program. This program interfaces with the runtime.

2.4.4 Runtime

The runtime layer is responsible for executing the concern code at the join points. It is activated at the join points by function calls that are woven in by the weaver. A

reduced copy of the knowledge base, containing the necessary information for filter evaluation and execution, is enclosed with the runtime. Method calls are intercepted by the runtime. The runtime evaluates the filter set for the intercepted message and executes the corresponding filter actions. The runtime also facilitates the debugging of the composition filter implementations.

2.5 Platforms

The composition filters concept of Compose⋆ can be applied to any programming language, given that certain assumptions are met. Currently, Compose⋆ supports three platforms: .NET, Java and C. For each platform different tools are used for compilation and weaving. They all share the same platform independent compile-time.

Compose⋆/.NET targets the .NET platform. It is the oldest implementation of Compose⋆. Its weaver operates on CIL byte code. Compose⋆/.NET is programming language independent as long as the programming language can be compiled to CIL code. An add-in for Visual Studio is provided for ease of development. Compose⋆/J targets the Java platform and provides a plug-in for integration with Eclipse. Compose⋆/C contains support for the C programming language. The implementation is different from the Java and .NET counterparts, because it does not have a run-time environment. The filter logic is woven directly in the source code. Because the language C is not based on objects, filters are woven on functions based on membership of sets of functions. Like the Java platform, Compose⋆/C provides a plug-in for Eclipse.

2.6 Features Specific to Compose⋆

The Composition Filters approach uses a restricted (pattern matching) language to define filters. This language makes it possible to reason about the semantics of the concern. Compose⋆ offers three features that use this possibility, which originate in more control and correctness over an application under construction. These features are:

Ordering of filter modules
: It is possible to specify how the superimposition of filter modules should be ordered. Ordering constraints can be specified in a fixed, conditional, or partial manner. A fixed ordering can be calculated exactly, whereas a conditional ordering is dependent on the result of filter execution and therefore evaluated at runtime. When there are multiple valid orderings of filter modules on a join point, partial ordering constraints can be applied to reduce this number. These constraints can be declared in the concern definition;

Filter consistency checking
: When superimposition is applied, Compose⋆ is able to detect if the ordering and conjunction of filters creates a conflict. For example, imagine a set of filters where the first filter only lets method `m` continue and the second filter only accepts for methods `a` and `b`. Because only method `m` can reach the second filter, it never accepts. This might indicate a conflict.

Reason about semantic problems
When multiple pieces of advice are added to the same join point, Compose⋆ can reason about problems that may occur. An example of such a conflict is the situation where a real-time filter is followed by a wait filter. Because the wait filter can wait indefinitely, the real-time property imposed by the real-time filter may be violated.

The above mentioned conflict analyzers all work on the assumption that the behavior of every filter is well-defined. This is not the case for the meta filter, its user-undefined, and therefore unpredictable, behavior poses a problem to the analysis tools.

Furthermore, Compose⋆ is extended with features that enhance the usability. These features are briefly described below:

Integrated Development Environment support
The Compose⋆ implementations all have an IDE plug-in; Compose⋆/.NET for Visual Studio, Compose⋆/J and Compose⋆/C for Eclipse;

Debugging support
The debugger shows the flow of messages through the filters. It is possible to place breakpoints to view the state of the filters;

Incremental building process
When a project is build and not all the modules are changed, incremental building saves time.

Some language properties of Compose⋆ can also be seen as features, being:

Language independent concerns
A Compose⋆ concern can be used for all the Compose⋆ platforms, because the composition filters approach is language independent;

Reusable concerns
The concerns are easy to reuse, through the dynamic filter modules and the selector language;

Expressive selector language
Program elements of an implementation language can be used to select a set of objects to superimpose on;

Support for annotations
Using the selector, annotations can be woven at program elements. At the moment annotations can be used for superimposition.

2.7 StarLight

In 2006 development began on a lightweight branch of Compose⋆/.NET, called StarLight. The aim of this new branch is to provide a more robust and efficient variant of Compose⋆. Certain more advanced features, such as multiple inheritance, have been excluded from this new branch, while new features have been implemented to meet industry demands. This section describes the differences in the architecture between Compose⋆ and StarLight and certain new features of StarLight.

2.7.1 StarLight Architecture

This section describes differences between the StarLight Architecture and the Compose⋆ architecture, as shown in Figure 2.3.

Integrated Development Environment
StarLight has its own Visual Studio integration. This integration provides the following functionality:

- A project service to open and use StarLight projects in Visual Studio;
- Syntax highlighting and IntelliSense for concern files;
- MSBuild is used to analyze and weave the concerns.

Compile Time
StarLight uses the same compile time as Compose⋆. An inlining engine is added to the compile time. This inlining engine translates the filter set to a platform independent abstract instruction model for each individual message. This abstract instruction model represents a procedural structure. It can be easily translated to program code for a specific procedural platform.

Adaptation
StarLight has its own adaptation layer. It contains an analyzer that creates the language model. It also contains a weaver that translates the abstract instruction models for each message to IL code and weaves this code in the appropriate places of the target assemblies.

Runtime
The StarLight version does not have a runtime, because a complete filter set translation is woven in the target program.

2.7.2 Explicit Modeling of the Returning Flow

Originally, composition filters only modeled the calling flow: sending the message from the sender to the target. There is, however, also a returning flow: the action of returning the control to the sender after the target has ended the execution of the message. This returning flow possibly contains a return value, but this is not necessary. In the new StarLight implementation we also want to model this returning flow explicitly. This makes it possible to execute filter actions after the message has been dispatched.

Figure 2.4 shows the returning flow in the composition filters model. Both input filters as output filters have a returning flow.

When is the Flow Returned? Explicit modeling of the returning flow raises questions like when is the flow returned and is there always a returning flow. From the composition filters perspective, it is the filter action that decides whether to return the flow or to continue to the next filter. There are actually three possibilities, as shown in Figure 2.5.

Continue
The flow continues to the next filter. Examples of filter actions that continue the flow are the `Substitution` action and the `Advice` action.

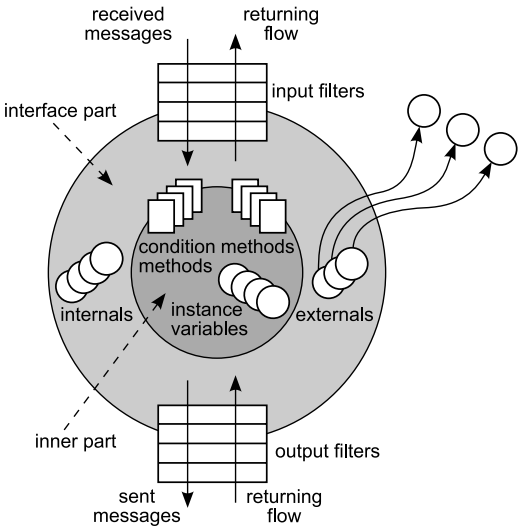


Figure 2.4: Explicit modeling of the returning flow

Return

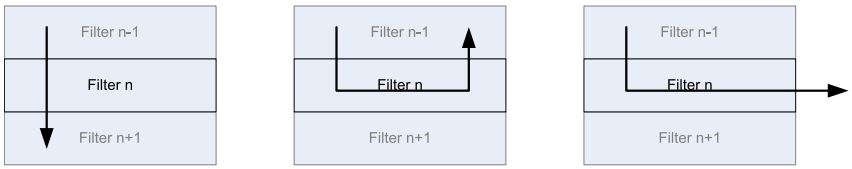
A filter action can return the flow. In this case, the calling flow turns into a returning flow. An example of a filter action that returns the flow is the `Dispatch` action.

Exit

The third option available for filter actions is to exit the filter set. This equals an exception or abnormal return. An example of a filter action that exits the filter set is the `Error` action.

4-Action Filter Types Because StarLight models the returning flow explicitly, it is possible to execute actions on the returning flow. The actions that are executed, are specified by the filter type. Therefore, instead of 2-action filter types, StarLight has 4-action filter types.

Evaluation of Filters on the Returning Flow? The filters are not evaluated on the returning flow. They are only evaluated on the calling flow. The filter action that is executed by a filter on the returning flow depends on whether the filter accepted or



(a) Continue (b) Return (c) Exit

Figure 2.5: Filter actions can either continue, return or exit.

	Calling Flow	Returning Flow
Accept	accept-call action	accept-return action
Reject	reject-call action	reject-return action

Table 2.1: The four filter actions of a filter type.

rejected on the calling flow.

The actions on the returning flow are executed in reverse order of the filter set.

2.7.3 Defining New Filter Types and Filter Actions

StarLight provides a new way to define new filter types and new filter actions. Actually, there are no primitive filter types and filter actions anymore, as there are in Compose⋆. There are, however, certain common filter types and filter actions provided by the StarLight API. But they are defined in the same way as any other new filter type or filter action.

```
[FilterActionAttribute("LoggingInAction", FilterActionAttribute.
    FilterFlowBehavior.Continue, FilterActionAttribute.
    MessageSubstitutionBehavior.Original)]
public class LoggingInAction : FilterAction
{
  public override void Execute(JoinPointContext context)
  {
    //Logging-in implementation
  }
}
```

Listing 2.5: Defining the `LoggingIn` filter actions

New filter actions can be defined by extending the `FilterAction` class. Listing 2.5 shows an example of a new filter action. By overriding the `Execute` method, the action can be implemented. When the action is executed at runtime, a call is done to its `Execute` method. A mandatory custom attribute on the filter action specifies the following information:

- The name of the filter action;
- The flow behavior of the filter action: continue, return or exit;
- The substitution behavior of the filter action. This specifies whether the message continues substituted or not after the filter action. An example of a filter action that leaves the message substituted is the `Substitution` action.

```
[FilterTypeAttribute("Logging", "LoggingInAction", FilterAction.
    ContinueAction, "LoggingOutAction",FilterAction.ContinueAction)]
class LoggingFilterType : FilterType
{
  //No implementation
}
```

Listing 2.6: Defining the `Logging` filter type

A new filter type can be defined by extending the `FilterType` class. Listing 2.6 shows an example of a new filter type. This class has no implementation methods, as it is not used at runtime. It is only used to specify a new filter type at compile time. A mandatory custom attribute specifies the following information:

- The name of the filter type. This name can be used in filter specifications;
- The filter actions, in the order accept-call, reject-call, accept-return and reject-return.

The example shows the definition of a `Logging` filter type that logs both on call as on return, when the filter accepts.

2.7.3.1 Built-in Filter Actions

The StarLight API provides several filter actions:

Continue This filter action continues the execution of the filter set to the next filter. It does not have any other behavior

Dispatch This action dispatches the message to the specified target.

Advice This action executes a specific advice method, specified by the substitution part. This is a lightweight replacement of the `Meta` action. It cannot change the message or change the flow behavior of the message. It also does not have the multithreading functionality of the `Meta` action. After an `Advice` action, the flow continues to the next filter with an unchanged message.

Error This action raises an exception and causes the flow to exit the filter set.

Substitution This action substitutes the message with the message specified by the substitution part. The execution of the filter set continues to the next filter.

2.7.3.2 Built-in Filter Types

The StarLight API provides several filter types:

		Calling Flow	Returning Flow
Dispatch	Accept	Dispatch action	Continue action
	Reject	Continue action	Continue action

		Calling Flow	Returning Flow
Before	Accept	Advice action	Continue action
	Reject	Continue action	Continue action

		Calling Flow	Returning Flow
After	Accept	Continue action	Advice action
	Reject	Continue action	Continue action

		Calling Flow	Returning Flow
Error	Accept	Error action	Continue action
	Reject	Continue action	Continue action

		Calling Flow	Returning Flow
Substitution	Accept	Subst. action	Continue action
	Reject	Continue action	Continue action

2.7.4 Conditional Superimposition

Another new feature in the StarLight version of Compose★ is conditional superimposition. Conditional superimposition makes it possible to superimpose a filter module conditionally. At runtime the condition is evaluated to decide whether the filter module should be executed. This makes it possible to turn crosscutting behavior on and off at runtime. It is even possible to turn the filter module off for specific classes, because the condition can get the classname as parameter.

```
concern LoggingConcern
{
  filtermodule LoggingFM
  {
    inputfilters
      logging  : Logging = { True => [*.*] }
  }

  superimposition
  {
    conditions
        loggingEnabled : Logging.LoggingEnabled;
    selectors
      loggingClasses = { ... };
    filtermodules
      loggingEnabled => loggingClasses <- LoggingFM;
  }
}
```

Listing 2.7: Conditional superimposition example

Listing 2.7 shows an example of conditional superimposition. In this example, the `LoggingFM` filter module is superimposed conditionally. Before the filter module is executed at runtime, the condition `loggingEnabled` is checked.

Chapter 3

Introduction to the .NET Framework

"The best way to prepare [to be a programmer] is to write programs,
and to study great programs that other people have written.
In my case, I went to the garbage cans at the Computer Science Center
and fished out listings of their operating system."
William Henry Gates III
US computer software designer & industrialist (1955 -)

This chapter gives an introduction to the .NET Framework of Microsoft. First, the architecture of the .NET Framework is introduced. This section includes terms like the Common Language Runtime, the .NET Class Library, the Common Language Infrastructure and the Intermediate Language. These are discussed in more detail in the sections following the architecture.

3.1 Introduction

Microsoft defines [33] .NET as follows; ".NET is the Microsoft Web services strategy to connect information, people, systems, and devices through software.". There are different .NET technologies in various Microsoft products providing the capabilities to create solutions using web services. Web services are small, reusable applications that help computers from many different operating system platforms to work together by exchanging messages. Based on industry standards like XML (Extensible Markup Language), SOAP (Simple Object Access Protocol), and WSDL (Web Services Description Language) they provide a platform and language independent way to communicate.

Microsoft products, such as Windows Server System (providing web services) or Office System (using web services) are some of the .NET technologies. The technology described in this chapter is the .NET Framework. Together with Visual Studio, an integrated development environment, they provide the developer tools to create programs for .NET.

Many companies are largely dependent on the .NET Framework, but need or want to use AOP. Currently there is no direct support for this in the Framework. The

Compose⋆/.NET project is addressing these needs with its implementation of the Composition Filters approach for the .NET Framework.

This specific Compose⋆ version for .NET has two main goals. First, it combines the .NET Framework with AOP through Composition Filters. Second, Compose⋆ offers superimposition in a language independent manner. The .NET Framework supports multiple languages and is, as such, suitable for this purpose. Composition Filters are an extension of the object-oriented mechanism as offered by .NET, hence the implementation is not restricted to any specific object-oriented language.

3.2 Architecture of the .NET Framework

The .NET Framework is Microsoft's platform for building, deploying, and running Web Services and applications. It is designed from scratch and has a consistent API providing support for component-based programs and Internet programming. This new Application Programming Interface (API) has become an integral component of Windows. The .NET Framework was designed to fulfill the following objectives [31]:

Consistency
: Allow object code to be stored and executed locally, executed locally but Internet-distributed, or executed remotely and to make the developer experience consistent across a wide variety of types of applications, such as Windows-based applications and Web-based applications;

Operability
: The ease of operation is enhanced by minimizing version conflicts and providing better software deployment support;

Security
: All the code is executed safely, including code created by an unknown or semi-trusted third party;

Efficiency
: The .NET Framework compiles applications to machine code before running, thus eliminating the performance problems of scripted or interpreted environments;

Interoperability
: Code based on the .NET Framework can integrate with other code because all communication is built on industry standards.

The .NET Framework consists of two main components [31]: the Common Language Runtime (CLR, simply called the .NET Runtime or Runtime for short) and the .NET Framework Class Library (FCL). The CLR is the foundation of the .NET Framework, executing the code and providing the core services such as memory management, thread management and exception handling. The CLR is described in more detail in Section 3.3. The class library, the other main component of the .NET Framework, is a comprehensive, object-oriented collection of reusable types that can be used to develop applications ranging from traditional command-line or graphical user interface (GUI) applications to applications such as Web Forms and XML Web services. Section 3.5 describes the class libraries in more detail.

The code run by the runtime is in a format called Common Intermediate Language (CIL), further explained in Section 3.6. The Common Language Infrastructure (CLI)

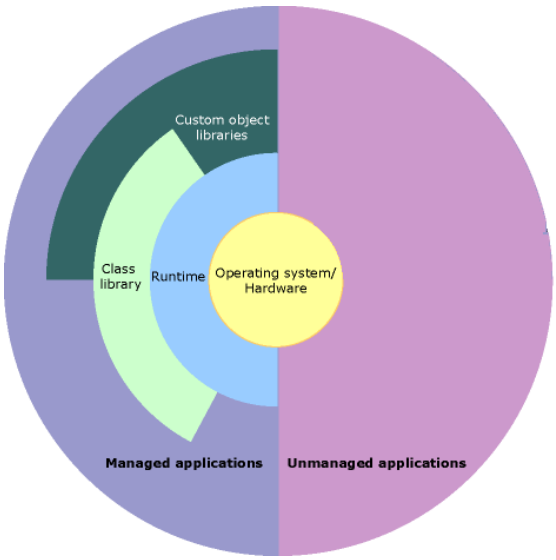


Figure 3.1: Context of the .NET Framework (Modified) [31]

is an open specification that describes the executable code and runtime environment that form the core of the Microsoft .NET Framework. Section 3.4 tells more about this specification.

Figure 3.1 shows the relationship of the .NET Framework to other applications and to the complete system. The two parts, the class library and the runtime, are managed, i .e., applications managed during execution. The operating system is in the core, managed and unmanaged applications operate on the hardware. The runtime can use other object libraries and the class library, but the other libraries can use the same class library themselves.

Besides the Framework, Microsoft also provides a developer tool called the Visual Studio. This is an IDE with functionality across a wide range of areas, allowing developers to build applications with decreased development time in comparison with developing applications using command line compilers.

3.2.1 Version 2.0 of .NET

In November 2005, Microsoft released a successor of the .NET Framework. Major changes are the support for generics, the addition of nullable types, 64 bit support, improvements in the garbage collector, new security features and more network functionality.

Generics make it possible to declare and define classes, structures, interfaces, methods and delegates with unspecified or generic type parameters instead of specific types. When the generic is used, the actual type is specified. This allows for type-safety at compile-time. Without generics, the use of casting or boxing and unboxing decreases performance. By using a generic type, the risks and costs of these operations is reduced.

Nullable types allow a value type to have a normal value or a null value. This null value can be useful for indicating that a variable has no defined value because the

information is not currently available.

Besides changes in the Framework, there are also improvements in the four main Microsoft .NET programming languages (C#, VB.NET, J# and C++). The language elements are now almost equal for all languages. For instance, additions to the Visual Basic language are the support for unsigned values and new operators. Additions to the C# language include the ability to define anonymous methods, thus eliminating the need to create a separate method.

A new Visual Studio 2005 edition was released to support the new Framework and functionalities to create various types of applications.

3.3 Common Language Runtime

The Common Language Runtime executes code and provides core services. These core services are memory management, thread execution, code safety verification and compilation. Apart from providing services, the CLR also enforces code access security and code robustness. Code access security is enforced by providing varying degrees of trust to components, based on a number of factors, e .g., the origin of a component. This way, a managed component might or might not be able to perform sensitive functions, like file-access or registry-access. By implementing a strict type-and-code-verification infrastructure, called the Common Type System (CTS), the CLR enforces code robustness. Basically there are two types of code;

Managed
: Managed code is code that has its memory handled and its types validated at execution by the CLR. It has to conform to the Common Type Specification (CTS Section 3.4). If interoperability with components written in other languages is required, managed code has to conform to an even more strict set of specifications, the Common Language Specification (CLS). The code is run by the CLR and is typically stored in an intermediate language format. This platform independent intermediate language is officially known as Common Intermediate Language (CIL Section 3.6) [55].

Unmanaged
: Unmanaged code is not managed by the CLR. It is stored in the native machine language and is not run by the runtime but directly by the processor.

All language compilers (targeting the CLR) generate managed code (CIL) that conforms to the CTS.

At runtime, the CLR is responsible for generating platform specific code, which can actually be executed on the target platform. Compiling from CIL to the native machine language of the platform is executed by the just-in-time (JIT) compiler. Because of this language independent layer it allows the development of CLRs for any platform, creating a true interoperability infrastructure [55]. The .NET Runtime from Microsoft is actually a specific CLR implementation for the Windows platform. Microsoft has released the *.NET Compact Framework* especially for devices such as personal digital assistants (PDAs) and mobile phones. The .NET Compact Framework contains a subset of the normal .NET Framework and allows .NET developer to write mobile applications.

Components can be exchanged and web services can be used so an easier interoperability between mobile devices and workstations/servers can be implemented [30].

At the time of writing, the .NET Framework is the only advanced Common Language Infrastructure (CLI) implementation available. A shared-source[1] implementation of the CLI for research and teaching purposes was made available by Microsoft in 2002 under the name Rotor [49]. In 2006 Microsoft released an updated version of Rotor for the .NET platform version two. Also Ximian is working on an open source implementation of the CLI under the name Mono[2], targeting both Unix/Linux and Windows platforms. Another, somewhat different approach, is called Plataforma.NET[3]. Plataforma.NET aims to be a hardware implementation of the CLR, so that CIL code can be run natively.

3.3.1 Java VM vs .NET CLR

There are many similarities between Java and .NET technology. This is not strange, because both products serve the same market.

Both Java and .NET are based on a runtime environment and an extensive development framework. These development frameworks provide largely the same functionality for both Java and .NET. The most obvious difference between them is lack of language independence in Java. While Java's strategy is 'One language for all platforms' the .NET philosophy is 'All languages on one platform'. However these philosophies are not as strict as they seem. As noted in Section 3.5 there is no technical obstacle for other platforms to implement the .NET Framework. There are compilers for non-Java languages like Jython (Python) [23] and WebADA [1] available for the JVM. Thus, the JVM in its current state has difficulties supporting such a vast array of languages as the CLR. However, the multiple language support in .NET is not optimal and has been the target of some criticism.

Although the JVM and the CLR provide the same basic features, they differ in some ways. While both CLR and the modern JVM use JIT (Just In Time) compilation, the CLR can directly access native functions. This means that with the JVM an indirect mapping is needed to interface directly with the operating system.

3.4 Common Language Infrastructure

The entire CLI has been documented, standardized and approved [22] by the European association for standardizing information and communication systems, Ecma International[4]. Benefits of this CLI for developers and end-users are:

- Most high level programming languages can easily be mapped onto the Common Type System (CTS);
- The same application will run on different CLI implementations;

[1] Only non-commercial purposes are allowed.

[2] `http://www.go-mono.com/`

[3] `http://personals.ac.upc.edu/enric/PFC/Plataforma.NET/p.net.html`

[4] An European industry association founded in 1961 and dedicated to the standardization of Information and Communication Technology (ICT) Systems. Their website can be found at `http://www.ecma-international.org/`.

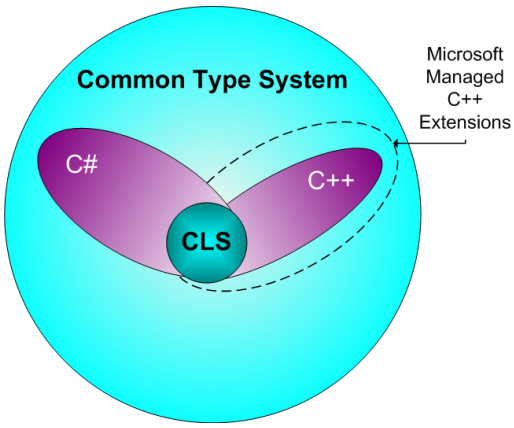


Figure 3.2: Relationships in the CTS

- Cross-programming language integration, if the code strictly conforms to the Common Language Specification (CLS);
- Different CLI implementations can communicate with each other, providing applications with easy cross-platform communication means.

This interoperability and portability is, for instance, achieved by using a standardized meta data and intermediate language (CIL) scheme as the storage and distribution format for applications. In other words, (almost) any programming language can be mapped to CIL, which in turn can be mapped to any native machine language.

The Common Language Specification is a subset of the Common Type System, and defines the basic set of language features that all .NET languages should adhere to. In this way, the CLS helps to enhance and ensure language interoperability by defining a set of features that are available in a wide variety of languages. The CLS was designed to include all the language constructs that are commonly needed by developers (e.g., naming conventions, common primitive types), but no more than most languages are able to support [32]. Figure 3.2 shows the relationships between the CTS, the CLS, and the types available in C++ and C#. In this way the standardized CLI provides, in theory[1], a true cross-language and cross-platform development and runtime environment.

To attract a large number of developers for the .NET Framework, Microsoft has released CIL compilers for C++, C#, J#, and VB.NET. In addition, third-party vendors and open-source projects also released compilers targeting the .NET Framework, such as Delphi.NET, Perl.NET, IronPython, and Eiffel.NET. These programming languages cover a wide-range of different programming paradigms, such as classic imperative, object-oriented, scripting, and declarative languages. This wide coverage demonstrates the power of the standardized CLI.

Figure 3.3 shows the relationships between all the main components of the CLI. The top of the figure shows the different programming languages with compiler support for the CLI. Because the compiled code is stored and distributed in the Common Intermediate Language format, the code can run on any CLR. For cross-language usage this code

[1]Unfortunately Microsoft did not submit all the framework classes for approval and at the time of writing only the .NET Framework implementation is stable.

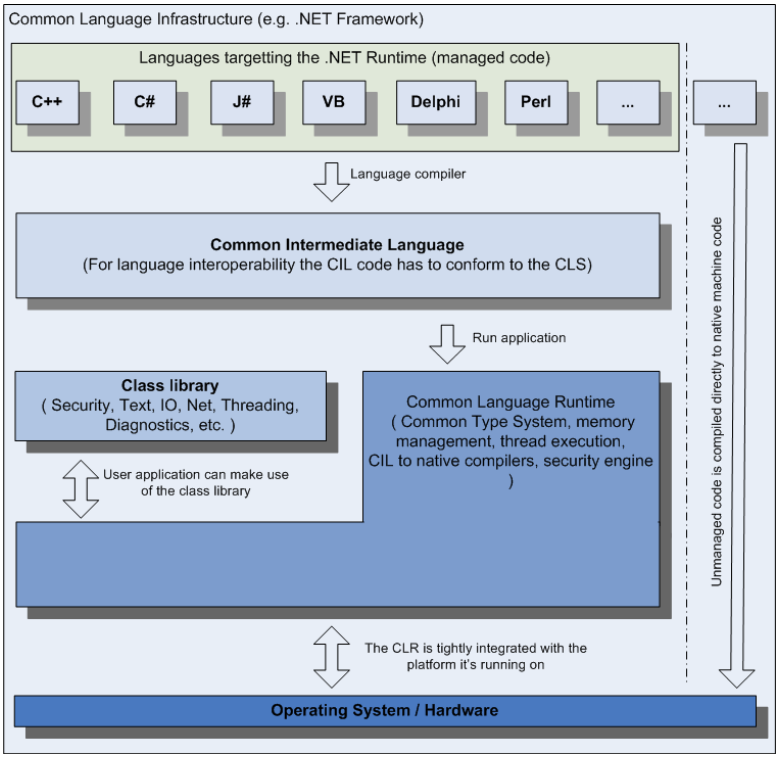


Figure 3.3: Main components of the CLI and their relationships. The right hand side of the figure shows the difference between managed code and unmanaged code.

has to comply with the CLS. Any application can use the class library (the FCL) for common and specialized programming tasks.

3.5 Framework Class Library

The .NET Framework class library is a comprehensive collection of object-oriented reusable types for the CLR. This library is the foundation on which all the .NET applications are built. It is object oriented and provides integration of third-party components with the classes in the .NET Framework. Developers can use components provided by the .NET Framework, other developers and their own components. A wide range of common programming tasks (e .g., string management, data collection, reflection, graphics, database connectivity or file access) can be accomplished easily by using the class library. Also, a great number of specialized development tasks are extensively supported, like:

- Console applications;
- Windows GUI applications (Windows Forms);
- Web applications (Web Forms);
- XML Web services;
- Windows services.

All the types in this framework are CLS compliant and can therefore be used from any programming language whose compiler conforms to the Common Language Specification

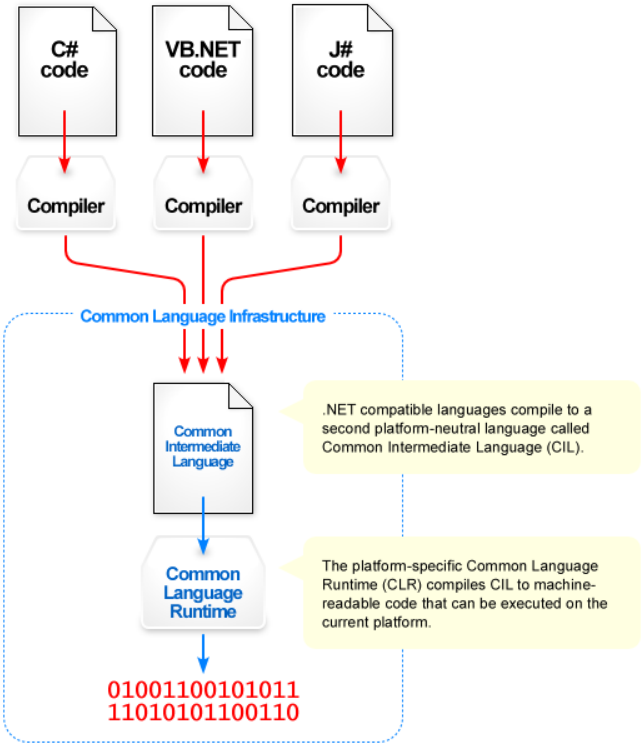


Figure 3.4: From source code to machine code

(CLS).

3.6 Common Intermediate Language

The Common Intermediate Language (CIL) has already been mentioned briefly in the sections before, but this section will describe the CIL in more detail. All the languages targeting the .NET Framework compile to this CIL (see Figure 3.4).

A .NET compiler generates a *managed module*, which is an executable designed to be run by the CLR [42]. There are four main elements inside a managed module:

- A Windows Portable Executable (PE) file header;
- A CLR header containing important information about the module, such as the location of its CIL and metadata;
- Metadata describing everything inside the module and its external dependencies;
- The CIL instructions generated from the source code.

The Portable Executable file header allows the user to start the executable. This small piece of code will initiate the just-in-time compiler which compiles the CIL instructions to native code when needed, while using the metadata for extra information about the program. This native code is machine dependent while the original IL code is still machine independent. In this way, the same IL code can be JIT-compiled and executed on any supported architecture. The CLR cannot use the managed module directly but needs an assembly.

An assembly is the fundamental unit of security, versioning, and deployment in the .NET Framework. It is a collection of one or more files grouped together to form a logical unit [42]. Besides managed modules inside an assembly, it is also possible to include resources like images or text. A manifest file is contained in the assembly, describing not only the name, culture and version of the assembly but also the references to other files in the assembly and security requests.

The CIL is an object oriented assembly language with around 100 different instructions called OpCodes. It is stack-based, meaning objects are placed on an evaluation stack before the execution of an operation, and when applicable, the result can be found on the stack after the operation. For instance, if two numbers have to be added, first those numbers are placed onto the stack, then the add operation is called and finally the result can be retrieved from the stack.

```
.assembly AddExample {}

.method static public void main() il managed
{
  .entrypoint            // entry point of the application
  .maxstack 2

  ldc.i4 3               // Place a 32-bit (i4) 3 onto the stack
  ldc.i4 7               // Place a 32-bit (i4) 7 onto the stack

  add                    // Add the two and
                         // leave the sum on the stack

  // Call static System.Console.Writeline function
  // (function pops integer from the stack)
  call void [mscorlib]System.Console::WriteLine(int32)

  ret
}
```

Listing 3.1: Adding example in IL code

To illustrate how to create a .NET program in IL code we use the previous example of adding two numbers and show the result. In Listing 3.1 a new assembly is created with the name `AddExample`. In this assembly, a function `main` is declared as the starting point (`entrypoint`) of the assembly. The `maxstack` command indicates there can be a maximum of two objects on the stack. This is enough for the example method. Next, the values 3 and 7 are placed onto the stack. The `add` operation is called and the result stays on the stack. The method `WriteLine` from the .NET Framework Class Library is called. This method resides inside the `Console` class placed in the `System` assembly. It expects one parameter with `int32` as its type. This parameter will be retrieved from the stack. The `call` operation will transfer the control flow to this method, passing along the parameters as objects on the stack. The `WriteLine` method does not return a value. The `ret` operation returns the control flow from the main method to the calling method, in this case the runtime. This will exit the program.

To be able to run this example, we need to compile the IL code to byte code where each OpCode is represented as one byte. To compile this example, save it as a text file and run the *ILASM* compiler with the filename as parameter. This will produce an

executable that is runnable on all the platforms where the .NET Framework is installed.

This example was written directly in IL code, but we could have used a higher level language such as C# or VB.NET. For instance, the same example in C# code is shown in Listing 3.2 and the VB.NET version is listed in Listing 3.3. When this code is compiled to IL, it will look like the code in Listing 3.1.

```
public static void main()
{
      Console.WriteLine((int) (3 + 7));
}
```

Listing 3.2: Adding example in the C# language

```
Public Shared Sub main()
      Console.WriteLine(CType((3 + 7), Integer))
End Sub
```

Listing 3.3: Adding example in the VB.NET language

Part II

Message Flow Analysis

Chapter 4

Motivation

"The significant problems we have
cannot be solved at the same level of thinking
with which we created them."
Albert Einstein (1879 - 1955)
US (German-born) physicist

This thesis is concerned with reasoning about composition filters, especially about sets of filters on a single, possibly shared, join point. Composition filters provide a declarative way to specify a set of filters. This makes it possible to reason statically about composition filters. The information obtained by filter reasoning can be used for various purposes. This chapter motivates why filter reasoning is relevant.

Filter reasoning includes control flow analysis. However, control flow analysis alone is not enough. Certain applications of filter reasoning require a more powerful reasoning technique to solve their problems. The first section discusses three of these applications. It explains why control flow analysis does not give enough information to solve their problems. It also describes the additional information that filter reasoning should provide to solve them. The second section formalizes the information that filter reasoning should provide in a problem description. This problem description is extended with a set of additional requirements in the third section. The final section describes two existing approaches to filter reasoning. These existing approaches have, however, a number of problems that make it necessary to develop a new approach.

4.1 The Purpose of Filter Reasoning

This section describes three different applications of filter reasoning. They will be explained by using an example mail system application. This chapter only presents the necessary code fragments. The complete code can be found in Appendix A. Figure 4.1 shows the base structure of this application.

The basic mail system contains two classes, `MailSystem` and `Connection`. Both are singletons. `MailSystem` can be used to send and receive mail. It thereby uses the `Connection` object to actually send the data. `Connection` contains two methods to send and receive data: `send` and `receive`. It also contains three methods for connection management:

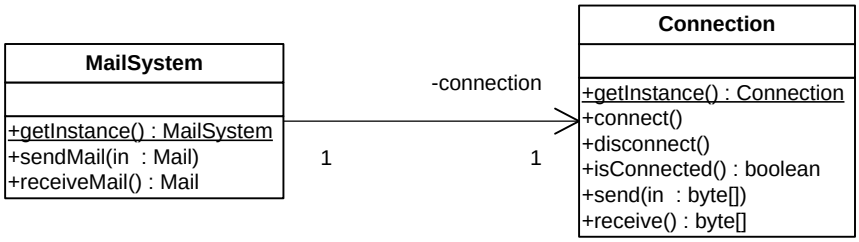


Figure 4.1: Structure of the mail system example.

`connect`, `disconnect` and `isConnected`.

4.1.1 Conflict Analysis and Error Detection

Filter reasoning can be used by various tools for conflict analysis and error detection. To illustrate this, the mail system example is extended with two concerns. The first concern is the `LogMail` concerns. This concern adds logging to the mail system. It logs each sent and received mail. The `LogMail` concern is shown in Listing 4.1.

```
concern LogMail {
  filtermodule LogFM {
    internals
      logger : Logger;
    inputfilters
      before : Before = { [*.sendMail] logger.logSend };
      after  : After  = { [*.receiveMail] logger.logReceipt }
  }
...
}
```

Listing 4.1: The `LogMail` concern

The `LogFM` filter module is superimposed on `MailSystem`. It has two filters: one `Before` filter to log a sent message and one `After` filter to log a received message.

The second concern is the `BufferMail` concern. The `BufferMail` concern adds buffering to the `MailSystem`; if the `Connection` object has no connection, the mail is stored in the buffer until there is a connection. Listing 4.2 shows the `BufferMail` concern.

```
concern BufferMail {
  filtermodule BufferFM {
    externals
      connection : Connection = Connection.getInstance();
      buffer     : MailBuffer = MailBuffer.getInstance();
    conditions
      connected  : connection.isConnected();
    inputfilters
      disp : Dispatch = { !connected => [*.sendMail] buffer.storeMail,
                          True => [*.sendMail] inner.sendMail}
  }

  filtermodule CheckConnectionFM {
    externals
      buffer : MailBuffer = MailBuffer.getInstance();
    inputfilters
```

```
17        after : After = { [*.connect] buffer.sendMail}
18    }
19  ...
20  }
```

Listing 4.2: The `BufferMail` concern

The `BufferMail` concern has two filter modules. The first filter module is superimposed on the `MailSystem`. It contains one `Dispatch` filter. This `Dispatch` filter checks whether there is a connection. If there is no connection, a dispatch is done to the `storeMail` method in the `buffer` external. If there is a connection, a dispatch is done to `inner.sendMail`.

The second filter module is superimposed on the `Connection` concern. It contains one `After` filter. After the method `connect` is called in the `Connection` object, this `After` filter calls `buffer.sendMail` to send all mail in the buffer.

Filter Module Ordering Problem In this example, two filter modules are superimposed on the `MailSystem` concern: the `LogFM` and the `BufferFM`. No ordering constraints are specified. Now suppose that the `BufferFM` filter module is placed on top of the `LogFM` filter module. This gives the filter set shown in Listing 4.3.

```
1  disp   : Dispatch = { !connected => [*.sendMail] buffer.storeMail,
2                        True => [*.sendMail] inner.sendMail};
3  before : Before   = { [*.sendMail] logger.logSend };
4  after  : After    = { [*.receiveMail] logger.logReceipt }
```

Listing 4.3: The superimposed filter set

The problem with this filter set is that a sent mail is never logged. A dispatch of the `sendMail` message is done in the `BufferFM` filter module before the `Before` filter in the `LogFM` filter module is reached.

This problem can be identified by performing static analysis on the filter set. Control flow analysis is not sufficient, however. Control flow analysis finds that the `Before` filter can be reached and that it can accept. Control flow analysis cannot find this problem because it does not evaluate the expressions in branching statements. The problem in this example is caused by the fact that the `Dispatch` filter always accepts for the message `sendMail`.

To detect this problem, we need to analyze how specific messages behave in the filter set. We need to know which parts of the filter set are reachable by specific messages, which matching parts accept etc. In this way, it can be detected that the `Before` filter never accepts, because the message `sendMail` cannot reach it.

4.1.2 Signature Generation

Input filters can change the signature of a concern. Filter reasoning can be used to determine how a filter set changes the signature of a concern. To illustrate this, the mail system example is extended with the `SecureConnection` concern. This concern adds encryption to the `Connection` class, to make it more secure. The `SecureConnection` concern is shown in Listing 4.4.

```
concern SecureConnection {
  filtermodule SecureFM {
    internals
      encryption : Encryption
    inputfilters
      before : Before   = { [*.send] encryption.encrypt };
      after  : After    = { [*.receive] encryption.decrypt };
      disp   : Dispatch = { [*.setEncryptionKey] encryption.
          setEncryptionKey }
  }
...
}
```

Listing 4.4: The `SecureConnection` concern

The `SecureConnection` concern has one filter module, which is superimposed on the `Connection` class. This filter module has two filters for encryption and decryption: a `Before` filter that encrypts the data given to the `send` method and an `After` filter that decrypts the data returned by the `receive` method. Both filters use the *internal* `encryption` to execute the encryption.

The filter module also has a `Dispatch` filter that dispatches message `setEncryptionKey` to the *internal* `encryption`. But class `Connection` does not have a method `setEncryptionKey`. A call to `setEncryptionKey` can, however, be executed, because the filter set dispatches it to the *internal* `encryption`. Therefore, `setEncryptionKey` should be in the signature of the concern. The new signature of `Connection` is shown in Figure 4.2.

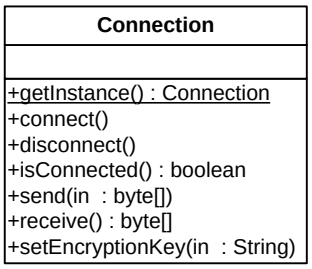


Figure 4.2: New signature of `Connection`.

It is the task of signature generation to create the new signature of a concern. A method should be in the signature of a concern if it can be dispatched. Therefore, the signature generation engine needs to check whether a specific message can reach a `Dispatch` action. Control flow analysis is insufficient for this purpose. With control flow analysis it can be detected whether messages might reach a `Dispatch` action, but it cannot be detected whether a specific message actually reaches the `Dispatch` action. Therefore, we need a technique that analyzes the behavior of specific messages in the filter set. In this way, it can be checked whether a specific message can reach a `Dispatch` action.

4.1.3 Filter Inlining

Currently, composition filters are implemented in Compose⋆ by using a runtime. This runtime has an interpreter for composition filters. A problem with an interpreter is that it is inefficient. A faster solution is to translate the filters to base language code. This code is then woven at specific places in the base program. To translate a filter set to program code, we need to know the behavior of the filter set. Filter reasoning can be used to analyze the behavior.

We first describe how a filter set can be translated using control flow analysis. We will illustrate this by using the `SecureConnection` concern from the mail system example. The filter set is translated to code for the message `send`.

Listing 4.5 shows the possible generated code if control flow analysis is used. Control flow analysis performs only a general flow analysis. Therefore, message matching and substitution needs to be evaluated in the generated code. For this purpose, a `message` variable is initialized with the *entrance message*. The entrance message is the message as it enters the filter set. This `message` variable is maintained throughout the generated code. Regularly, this `message` variable is checked whether it equals a certain message. This corresponds to the evaluation of the matching parts.

The generated code is almost the same for all messages. The only difference is the initial value of the `message` variable.

```
String message = "send";
if ( message.equals("send") ) {
  encryption.encrypt(context);
}
if ( message.equals("receive") ) {
  after1 = true;
}
if ( message.equals("setEncryptionKey") ) {
  encryption.setEncryptionKey(key);
  goto AFTER;
}

dispatch to inner.[message];

AFTER:
if ( after1 ) {
  encryption.decrypt(context);
}
```

Listing 4.5: Inlining of `send` with pure control flow analysis

The problem with this code is that it is too general. It is the same for all messages. Expressions are built-in to execute the message specific behavior. This can be further optimized by analyzing the behavior of the filter set for a specific message. In this way, code can be generated for that specific message. Listing 4.6 shows the generated code if the specific behavior of the `send` message is analyzed. There is no need anymore to maintain a `message` variable, because the code is specific for one message. Only code has to be generated for the specific actions executed by that message. For the `send` message, this is a call to the encryption advice method, followed by a dispatch to `inner`.

```
encryption.encrypt(context);
```

```
2 inner.send(data);
```

Listing 4.6: Inlining of `send` with message specific analysis

4.1.4 Further Applicability

This section presented three applications of filter reasoning. These applications are only a few examples of the applicability of filter reasoning; any tool that needs information about how a message behaves in a filter set can benefit from filter reasoning.

4.2 Problem Description

The previous section explained that control flow analysis of composition filters is not sufficient. Control flow analysis only reasons about the paths that messages can take through the filter set. It does not check whether expressions in branching elements are satisfied or not. For example, in control flow analysis both outgoing paths from a matching part are possible. In reality, a specific message can take only one path, due to the matching expression.

The previous section made clear that certain problems can only be solved by doing *data flow analysis* on the message object. Because this data flow analysis is specific on the message object, we call it *message flow analysis*. The declarative specification of message matching in composition filters makes message flow analysis possible.

Definition 4.2.1 (Message flow analysis) *Message flow analysis is the static analysis of the behavior of a specific message within a filter set. This involves:*

- Flow behavior*: which paths can a specific message take through the filter set.*
- Substitution behavior*: how is the message changed along a certain path in the filter set.*

Message flow analysis provides answers to questions like:

- Which matching parts accept and which matching parts reject when a specific message goes through the filter set
- How does a specific message change in the filter set.
- Can a specific message reach a `Dispatch` action.

These are all questions that cannot be answered by control flow analysis. A filter reasoning approach should therefore include message flow analysis.

In the rest of this thesis, the term *filter reasoning* is used as a general term including both *control flow analysis* as well as *message flow analysis*.

4.3 Requirements

The previous section specified that filter reasoning should include message flow analysis. This section presents a number of additional requirements that filter reasoning should

comply to. They are divided into conceptual requirements and design requirements. The conceptual requirements are requirements for the message flow analysis approach. The design requirements are requirements for the implementation of this approach in Compose⋆.

4.3.1 Conceptual Requirements

Traceability It should be possible to trace the results of message flow analysis back to the elements in the abstract syntax tree. For example, if message flow analysis determines that a certain message can reach a `Dispatch` action, it should be possible to trace this `Dispatch` action back to the corresponding `Dispatch` filter.

Fine-Grained Analysis Besides being traceable, the results should also be fine-grained. The reasoning algorithm should be able to give information about each part of the abstract syntax tree of the filter set. This means that the reasoning procedure should be able to give precise information about which parts of the filter set can be reached by a specific message: which matching parts accept, which matching parts reject, how and where is a message substituted in the filter set, which substitution part caused the substitution, etc.

Coarse-grained analysis does not give information that precise. It might, for example, only give information about the filter actions that are executed and the conditions expressions under which these filter actions are executed.

Efficient To be practically applicable, the reasoning procedure should be very efficient. It should have at least polynomial time complexity in the size of the filter set.

4.3.1.1 Design Requirements

Message flow analysis also needs to be implemented in the Compose⋆ project. We will now describe some requirements to the design of this implementation. The module that performs message flow analysis is called the *Filter Reasoning Engine*.

Provide a Generic Interface to Other Modules The filter reasoning engine should provide an interface for other modules. This interface should comply to the following requirements:

- The filter reasoning engine should provide models that contain all information obtained by the reasoning process. In this way, all information generated by the filter reasoning engine is available to other modules.
- The filter reasoning engine should provide generic tools to traverse and query the models. This makes the information provided by the filter reasoning engine more accessible.
- The filter reasoning engine should not contain functionality specific for one application or be tightly coupled with another module. Functionality specific for one application should be in that module and not in the interface of the filter reasoning engine.

Easy to Extend The design of the filter reasoning engine should be easy to extend with new filter types and new composition filter constructs:

- The reasoning engine should be independent of specific filter types. It should be possible to add new filter types without changing the implementation of the filter reasoning engine.

- New filter constructs should be easy to add. The syntax of composition filters might change. If this happens, the filter reasoning engine also needs to be adapted to cope with these changes. It should be possible to apply these changes to the filter reasoning engine without changing the entire reasoning process or redefining the implementation of other, unaffected filter constructs.

4.4 Existing Approaches

The previous sections described filter reasoning and message flow analysis. Filter reasoning is not new, however. There already exist a few approaches to reason about composition filters. They have been developed by Bosman [5]. These approaches, however, have a number of problems. This chapter shortly introduces the existing approaches and explains the problems with these approaches.

4.4.1 Logical Expressions Approach

The first approach developed by Bosman to reason about composition filters is the logical expression approach [5]. As Bosman already noted, this approach is not practically applicable due to a number of problems. This section gives a short introduction into this approach. It also explains the problems with this approach, especially the problems related to the requirements described in Section 4.3. More information about this approach can be found in [5].

4.4.1.1 Description of the Approach

Translating the filter set to a logical expression The logical expression approach uses logical expressions to reason about the filter set. It translates the filter set to a proposition logical expression. The propositions in the expression are the filter actions, the target and selector of the messages and the condition variables.

Each element of a filter set can be translated to a logical expression. A `Dispatch` filter is, for example, translated to the logical expression $([Match] \wedge DispatchAction) \vee (\neg[Match] \wedge [NextFilter])$. In this expression, $[Match]$ is the translation of the matching of the filter elements within the filter and $[NextFilter]$ is the translation of the next filter. This formula expresses that either the `Dispatch` filter matches and a `Dispatch` action is executed or that the `Dispatch` filter does not match and the next filter is executed.

If the complete filter set is translated to a logical expression, this logical expression can be used to reason about the filter set.

Querying The filter set can be queried by also translating the queries to logical expressions. The logical expression of the filter set is combined with the logical expression of the query. The combined logical expression is simplified using a theorem prover. The resulting logical expression is the answer to the query.

For example, if we want to query which messages can reach a `Dispatch` action, the query is translated to the logical expression $DispatchAction$. This is combined with the logical expression of the filter set, $[filterset]$, resulting in the following logical expression: $[filterset] \wedge DispatchAction$. This logical expression states that both the logical expression of the filter set as well as the logical expression $DispatchAction$ must be `true`, to indicate that a `Dispatch` action must be reached. Simplifying this combined expression leads to the desired result.

4.4.1.2 Problems with this Approach

Although this approach performs message flow analysis, it however has a number of problems that do not make it a suitable candidate for filter reasoning.

Coarse-grained Analysis The logical expressions created from the filter set only contain the condition variables, the filter actions and the target and selector of the message as propositions. It therefore can only answer questions about the actions executed, the valuation of the condition variables and the message. It cannot give exact information about which matching part accepted, which filter accepted, etc. For example, it can be queried that a certain message leads to a `Error` action. But if there are two `Error` filters, it is not known which filter caused the `Error` action.

This problem might be solved by adding the different parts of the filter set as propositions to the formula. For example, a matching part matching on $t.s$ is currently translated to the formula $t \wedge s$. This might be changed to $t \wedge s \wedge MatchingPart_i$, where i is a unique number for that matching part. In this way, the resulting formula for a query also gives information about the different parts of the filter set reached. No research has been done whether this solution has other implications or introduces new problems.

So, the logical expressions approach in its current state violates the requirement of fine-grained analysis. It can, however, be modified to do fine-grained analysis, but the implications of this modification have not been further investigated.

Inefficient Bosman describes that the time complexity for this approach is NP-complete [5]. This violates the requirement that the filter reasoning approach should have polynomial time complexity.

Other problems Bosman also identified the following problems [5]:

- The logical expressions resulting from a query may contain lots of unwanted information, which has to be filtered out.
- Certain elements in the filter set cause a dramatic increase in the size of the corresponding logical expression.

Because of these problems, Bosman developed a second approach to filter reasoning, the message-action tree approach.

4.4.2 Message-Action Tree Approach

Bosman also developed the message-action tree approach to filter reasoning [5]. This section describes this approach and the problems with this approach.

4.4.2.1 Description of the Approach

In the message-action tree approach a tree structure is created that indicates for each entrance message and each valuation of the condition variables, which actions are executed in the filter set.

```
err   : Error        = { [A.p] };
subst : Substitution = { [A.p] B.q };
disp  : Dispatch     = { [*.*] }
```

Listing 4.7: Message-action tree example

Listing 4.7, for example, results in the message-action tree shown in Figure 4.3.

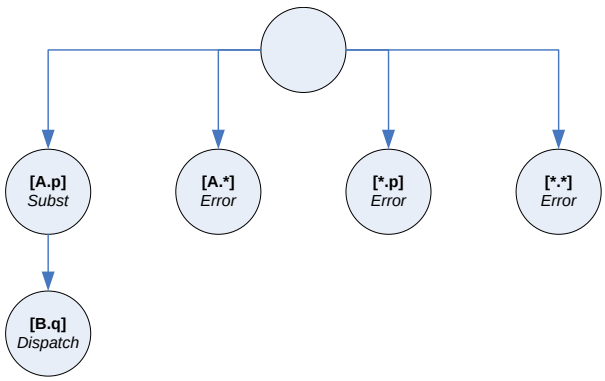


Figure 4.3: The message-action tree.

This message-action tree shows that message `A.p` leads to a `Substitution` action that substitutes the message to `B.q`. The `Substitution` action is followed by a `Dispatch` action. All other messages lead to an `Error` action.

The message-action tree is created by traversing the abstract syntax tree for each valuation of the condition variables and each possible message. During this traversal, condition expressions and message matching are evaluated and the correct branch is taken. This leads to an execution through the filter set. The filter actions during this execution are maintained and stored in the message-action tree.

Querying Querying can be done by traversing the message-action tree and searching for the needed information. For example, to check whether a certain message can reach a `Dispatch` action, all branches concerning that message in the message-action tree can be traversed to check for a `Dispatch` action.

4.4.2.2 Problems with this Approach

This approach also has a number of problems that does not make it a suitable candidate for filter reasoning.

Traceability This approach gives information about the actions executed by the filter set for a specific entrance message. But these actions can, however, not be traced back to the corresponding filter. A solution to this problem is to maintain the corresponding filter in the generation procedure.

Coarse-grained Analysis The resulting message-action tree only gives information about the actions executed and the order of the actions executed for a certain valuation of the conditions and a certain entrance message. From this information the accepting filters can be extracted, if the traceability mentioned earlier is implemented. But more fine-grained information is not available.

A solution to this problem is to maintain more information in the procedure that generates the message-action tree. Currently, the procedure only maintains the actions that occur. But because the procedure is applied to the complete abstract syntax tree of the filter set, information about other elements of the filter set can also be maintained. For example, the procedure might also maintain which matching parts accept and which matching parts reject. It has not been investigated what the implications of these changes are and how this information should be incorporated in the message-action tree.

Inefficient The reasoning is done for every valuation of the condition variables. This means that the reasoning is exponential in the number of condition variables; every additional condition variable doubles the number of valuations, which causes a doubling in the number of reasoning steps. This can be prevented by leaving the valuation of the condition variables open. This, however, reduces the power of the approach; certain paths might be found that can never occur in practice. We assume here that the valuation of the condition variables does not change during the execution of the filter set. This assumption is questioned in Chapter 5.

The message-action tree approach also assumes that a `Meta` filter might change the message and the valuation of the condition variables into any other message and any other valuation. This causes branching in the tree structure. When multiple `Meta` filters are placed after each other, this branching causes exponential growth of the message-action tree.

This exponential growth can be avoided by not using a tree structure, but a graph structure. In this case, different nodes that indicate the same filter, the same message and the same valuation of the condition variables can be taken together as one node. This is possible, because the remainder of the execution of the filter set is the same for each of these nodes. This solution limits the size of the filter set to the product of the number of filters, the number of messages and the number of valuations of the condition variables. Note that the size is still exponential in the number of condition variables, because the number of valuations of the condition variables is exponential in the number of condition variables.

So, the current message-action tree approach has exponential time complexity. This is due to two causes. One cause, the exponential growth of the tree due to `Meta` actions, can be prevented without loosing power. The other cause, the exponential number of valuations of the condition variables, cannot be prevented without loosing power.

Chapter 5

The Message Flow Simulation Approach

"Where all is but dream,
reasoning and arguments are of no use,
truth and knowledge nothing."
John Locke
English philosopher (1632 - 1704)

The previous chapter discussed a few approaches to message flow analysis. These approaches have a number of issues. This chapter presents a new approach to message flow analysis that solves these issues. This new approach is inspired by the work of Tom Staijen on using graph transformations for modeling aspect semantics [48]. In this new approach, the *abstract syntax tree* (AST) of the filter set is transformed to a *flowchart*. This flowchart is used to simulate the execution of a specific message in the filter set. Message matching is evaluated during this simulation. The result of the simulation is a model that represents the message flow through the filter set for that specific message.

The first section explains how the abstract syntax tree of a filter set is transformed to a flowchart. The second section explains how the execution of one specific message in the filter set is simulated using the flowchart. This simulation results in a state-space called the *execution model*. For some applications not only the execution model representing the execution of one specific message is needed, but the execution model representing all possible executions of all messages in the filter set. How this execution model can be created is explained in the third section. The fourth section presents the time complexity of the reasoning algorithm. Finally, some problems and issues are discussed.

5.1 Transforming the Abstract Syntax Tree to a Flowchart

The first step in the reasoning process is to transform the *abstract syntax tree* (AST) of the filter set to a *flowchart*. The AST is a representation of the syntax of the filter set. It does not represent any semantics. Figure 5.1 shows an example of an AST. This AST corresponds to the code in Listing 5.1.

```
error : Error    = { C1 => [*.*] };
```

```
disp  : Dispatch = { True => <inner.a> *.b }
```

Listing 5.1: Example of a filter set

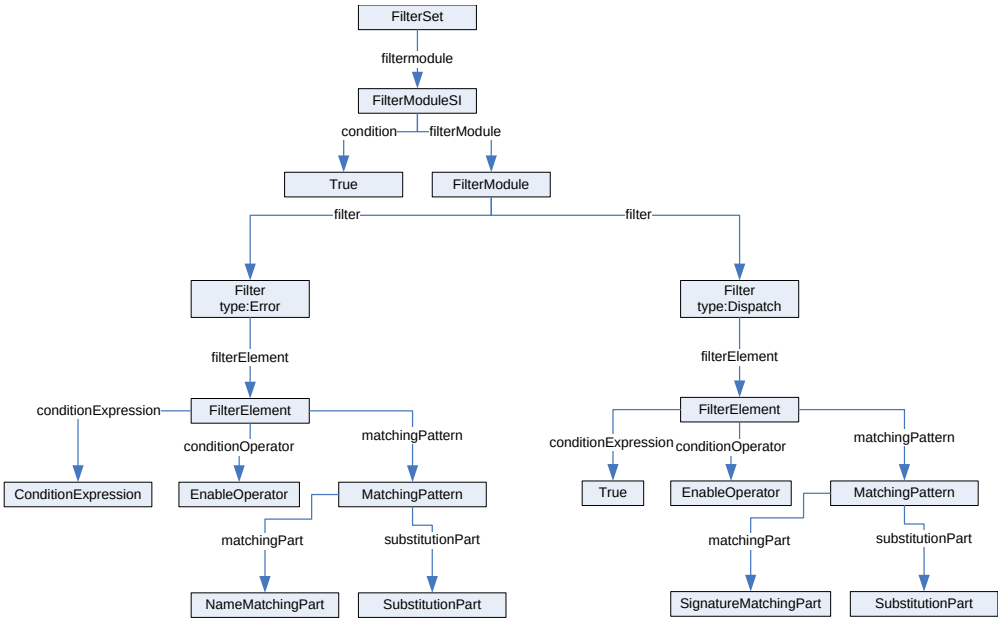


Figure 5.1: Example of an abstract syntax tree

To reason about composition filters we need a representation of the *semantics*. One part of the semantics is the *flow semantics*. Flow semantics specifies how a message can flow through the filter set: what parts of the filter set are visited in which order, where does branching take place and where does a message leave the filter set. The flow semantics is represented in a *flowchart*. The first step in the reasoning algorithm is to create the flowchart of the given filter set. The flowchart is created by transforming the abstract syntax tree, using transformation rules that introduce the flow semantics.

Figure 5.2 shows the flowchart corresponding to the previous AST. The nodes in a flowchart correspond to a certain point in the evaluation of the filter set. They represent the start of the evaluation of the corresponding element in the AST. For example, the `Filter` node in the flowchart represents the start of the evaluation of the corresponding filter in the filter set.

An edge in the flowchart represents an evaluation step in the filter set. Note that an edge does not indicate the end of the evaluation of the element corresponding to the start node. A flow edge from a parent node in the AST to a child node represents that the evaluation of the child node is started within the evaluation of the parent node. For example, a flow edge from a `Filter` node to a `FilterElement` node indicates the start of the evaluation of the filter element within the evaluation of the filter. The end of the evaluation of an element is implicit in the flowchart. The evaluation of a specific element in the filter set ends when a flow edge is targeted at a flow node corresponding to another element in the filter set that is not a descendant of that specific element.

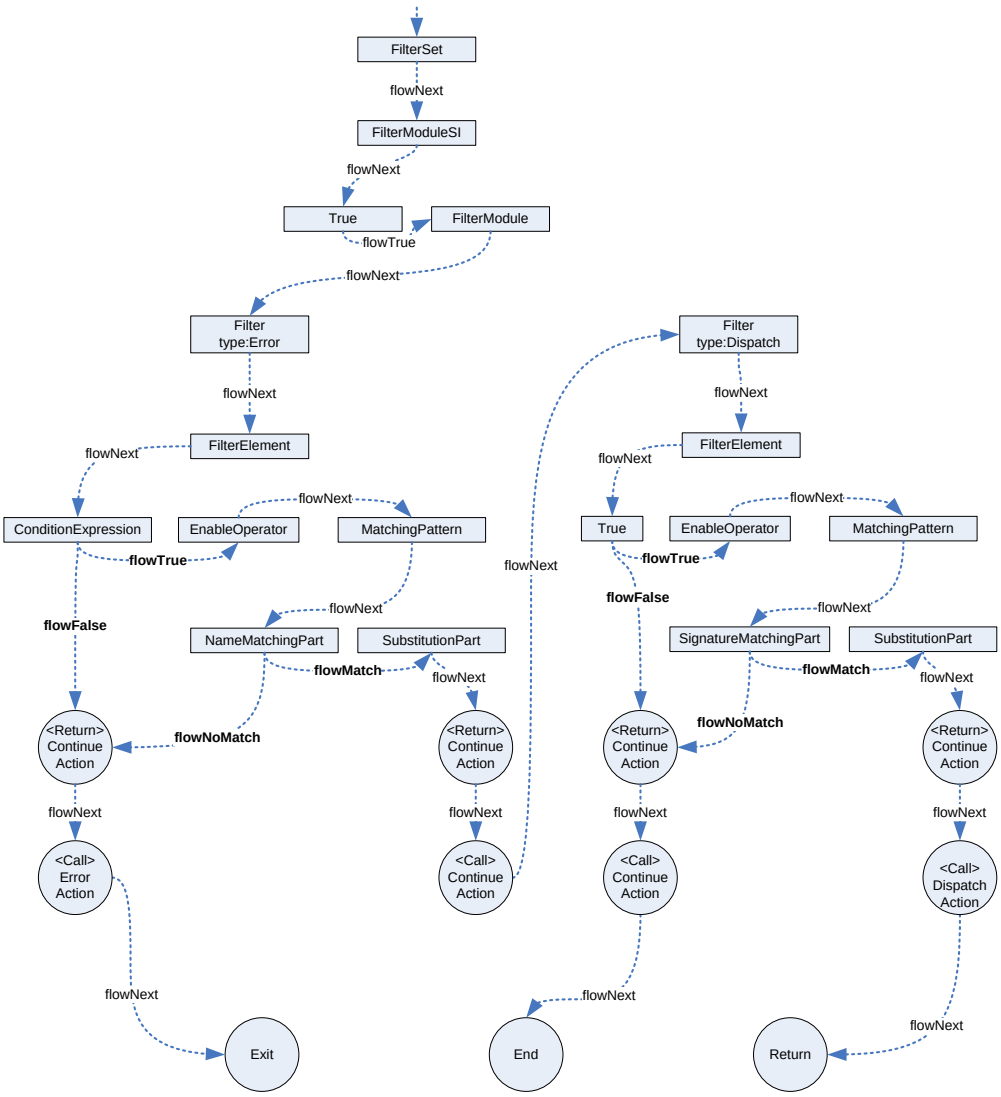


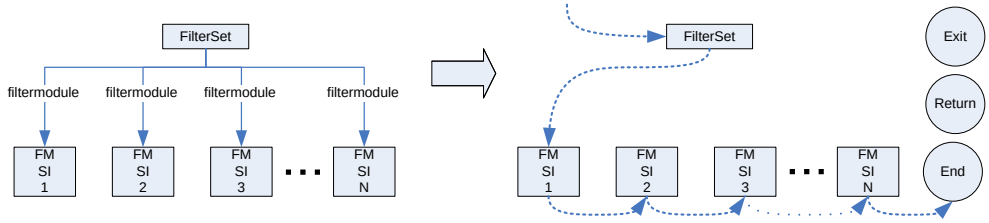
Figure 5.2: Example of a flowchart

5.1.1 Transformation Rules

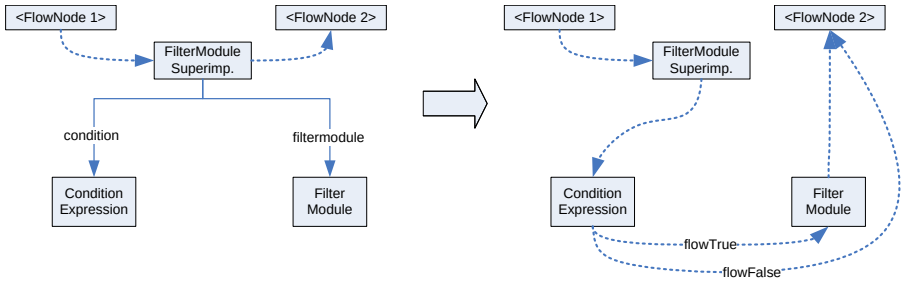
Because a flowchart partly represents the semantics of the filter set, transforming the AST to a flowchart is not a trivial task. The flow semantics of composition filters need to be implemented in specific transformation rules. We will now present a set of transformation rules to transform an AST to a flowchart. A top-down approach is used. This approach starts with the top level components and gradually works down to the leaf components. It should be noted that this is not the only possible approach to transform an AST to a flowchart.

Filter set rule

The first rule is the *filter set rule*. Figure 5.3a shows that flow goes sequentially through the superimposed filter modules in a filter set. This rule assumes that



(a) The filter set rule



(b) The conditional superimposition rule

Figure 5.3: The Transformation rules

the order of the filter modules in the filter set is fixed. This is also the case in a real execution of a filter set. The order of the filter modules is, however, not always specified. In this case, multiple orders of the filter modules are possible. To create the flowchart for all orders, the flowchart for each specific order needs to be created. The first step in the combined flowchart is a decision step to choose the specific ordering and to direct the flow to the corresponding flowchart. This is, however, not implemented in the current algorithm. The current algorithm assumes one specific filter module ordering.

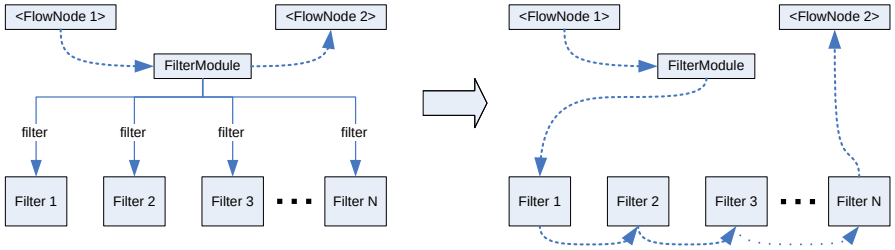
Note that a `FilterSet` node does not have the `FilterModule` nodes as direct child nodes, but `FilterModuleSuperimposition` nodes. A `FilterModuleSuperimposition` node represents the, possibly conditional, superimposition of a filter module on the concern.

Conditional superimposition rule

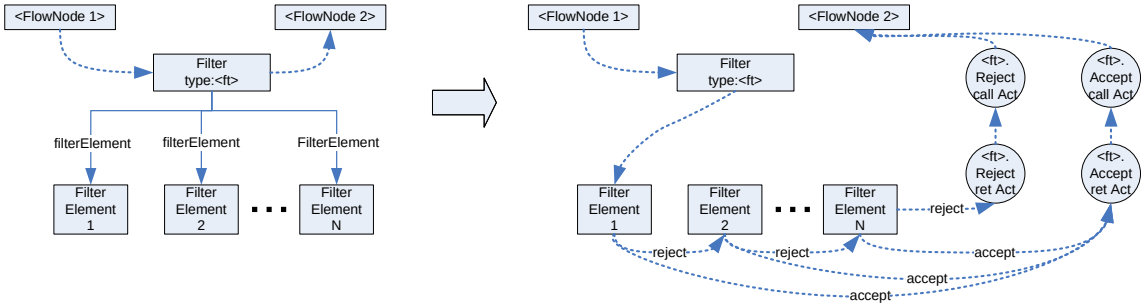
Figure 5.3b shows the *conditional superimposition rule*. In this rule, the nodes $\langle FlowNode1 \rangle$ and $\langle FlowNode2 \rangle$ represent respectively the entrance flow node and the exit flow node. A conditional superimposition of a filter module contains a condition expression and a filter module. Flow first goes to the condition expression. If the condition expression is `false`, flow goes to the exit node. If the condition expression is `true`, flow goes to the filter module. After the filter module, flow goes to the exit node.

Filter module rule

Different filters in a filter module are separated with an operator specifying the flow relationship between the filters. Currently, only the sequential flow operator is used. This operator specifies that flow goes sequentially through the filters in the



(c) The filter module rule



(d) The filter rule

Figure 5.3: The Transformation rules (Continued)

filter set. The *filter module rule* shown in Figure 5.3c assumes sequential flow. If other operators are going to be used, other transformation rules need to be defined.

Filter rule

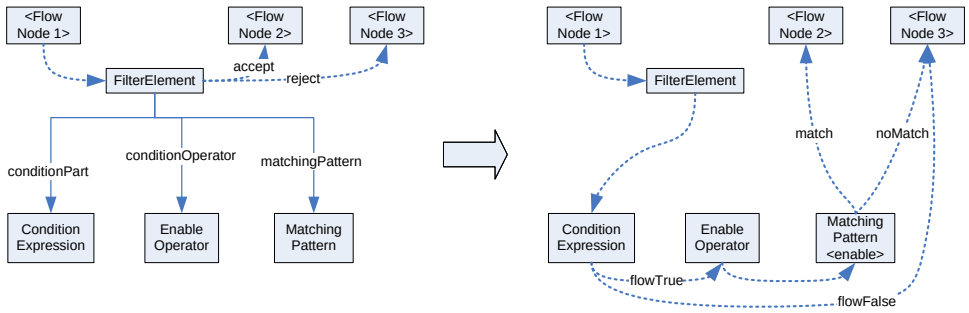
Figure 5.3d shows the *filter rule*. A filter has one or more filter elements. But it also has an *accept call* action, an *accept return* action, a *reject call* action and a *reject return* action. These four filter actions are implicit to the filter type. We want to model those filter actions as specific nodes in the flowchart, to make the execution of a filter action an explicit step. Therefore, the transformation rule adds the four filter actions as new nodes to the flowchart.

A filter element can accept as well as reject. So, there are two outgoing flow edges from a filter element; one accept edge and one reject edge. If a filter element accepts, flow immediately continues to the accept actions. If a filter element rejects, flow continues to the next filter element. If the last filter element rejects, the reject actions are executed.

Note that the evaluation of the two actions on return does not represent the execution of those actions but the statement that those actions are executed when the message returns.

Filter element rule

Figure 5.3e shows the *filter element rule*. Inside a filter element, flow first goes to the condition expression. In general, a condition expression can be `true` as well as `false`. If a condition expression is `false`, flow immediately continues to the reject exit node of the filter element (the filter element rejects). If a condition expression is `true`, flow goes to the condition operator.



(e) The filter element-enable rule

Figure 5.3: The Transformation rules (Continued)

There are two types of condition operators; the enable operator and the disable operator. The type of the condition operator influences the flow within a matching pattern. Therefore, the matching pattern is annotated with the type of the condition operator ('enable' or 'disable'). This annotation is used in the matching pattern rule. The type of the condition operator has no consequences for the flow within a filter element. Therefore, only the *filter element-enable rule* is shown here. The *filter element-disable rule* only differs in the annotation being added to the matching pattern.

Flow goes from the condition operator to the matching pattern. A matching pattern can either match or not match. If a matching pattern matches, flow goes to the accept exit node of the filter element (the filter element accepts). If a matching pattern does not match, flow goes to the reject exit node of the filter element (the filter element rejects).

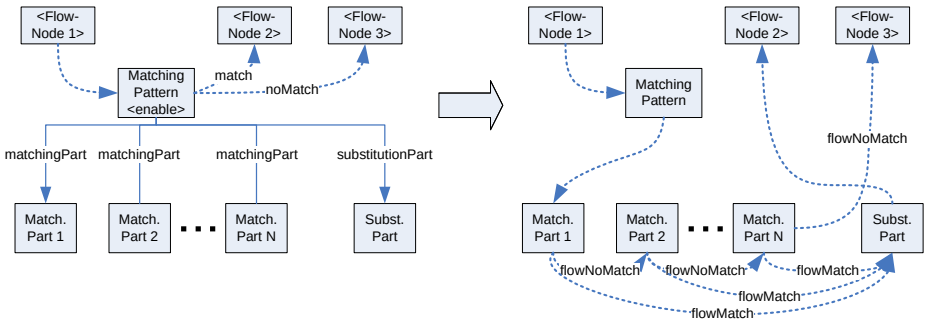
Matching pattern rules

For the matching pattern there are two rules, one for each type of condition operator. The matching pattern was annotated with the type of the condition operator in the filter element rule. This annotation is used to distinguish between the two rules.

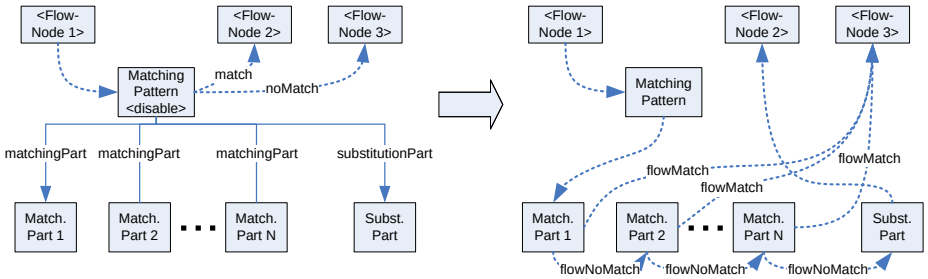
Figure 5.3f shows the *matching pattern-enable rule*. A matching pattern consists of one or more matching parts and a substitution part. The `enable` operator indicates that the matching pattern matches if there is one matching part that matches. Flow first goes to the first matching part. If a matching part matches, flow goes to the substitution part. From the substitution part, flow goes to the `match` exit of the matching pattern.

If a matching part does not match, flow goes to the next matching part. If the last matching part does not match, flow goes to the `noMatch` exit of the matching pattern.

The *matching pattern-disable rule* is shown in Figure 5.3g. The difference with the matching pattern-enable rule is the target of the `flowTrue` edges and the last `flowFalse` edge. The `disable` operator indicates that a matching pattern matches if all matching parts do not match. Therefore, if a matching part matches, flow



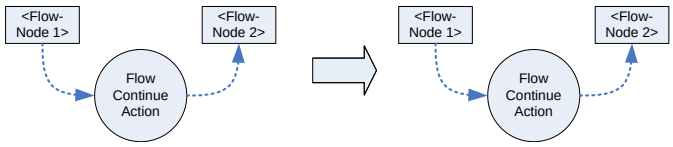
(f) The matching pattern-enable rule



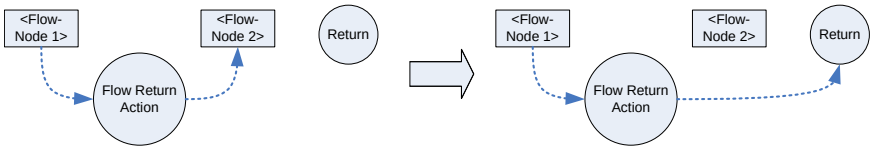
(g) The matching pattern-disable rule

Figure 5.3: The Transformation rules (Continued)

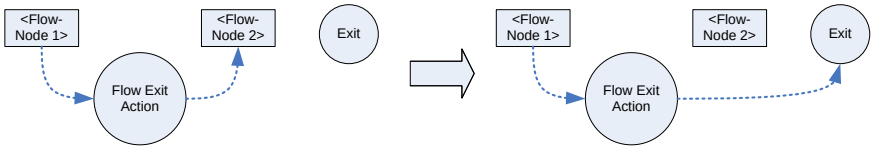
immediately goes to the `noMatch` exit node. If a matching part does not match, flow continues to the next matching part. If the last matching part does not match, all matching parts did not match, so flow continues to the substitution part. From the substitution part, flow goes to the `match` exit of the matching pattern.



(h) The flow-continue action rule



(i) The flow-return action rule



(j) The flow-exit action rule

Figure 5.3: The Transformation rules (Continued)

Action rules

There are three different action rules; the *flow-continue action rule*, the *flow-return action rule* and the *flow-exit action rule* (abnormal return). These action nodes are shown in respectively Figure 5.3h, Figure 5.3i and Figure 5.3j. Examples of actions that continue are the `Substitution` action and the `Before` action. An example of an action that returns is the `Dispatch` action. An example of an action that exits is the `Error` action.

For each new filter action that is added to the system, the flow behavior needs to be specified. As can be seen from the action rules, the flow behavior is either `continue`, `return` or `exit`.

If these transformation rules are applied to the AST in Figure 5.1 on page 53, the result is the flowchart shown in Figure 5.2 on page 54.

5.2 Simulating the Execution of a Message in the Filter Set

The previous step transformed the abstract syntax tree to a flowchart. A flowchart represents how messages might flow through the filter set. It does not say anything about how a specific message actually behaves in a filter set. To obtain this information, we have to simulate the execution of the filter set for that specific message. This is the next step in the reasoning process.

The flowchart is used in the simulation, because the flowchart already contains the flow semantics of the filter set. In this way, we do not have to implement the flow semantics in the simulation procedure. For example, if a name matching part matches in the simulation of an execution, the simulation procedure can use the outgoing `flowMatch` edge from the corresponding node in the flowchart to find the next part of the filter set to simulate.

5.2.1 Adding a Frame

To simulate the execution of a message in a filter set, a *frame element* is used, as can be seen in Figure 5.4. This frame element maintains a *program counter property*. This program counter property indicates the flow node corresponding to the element in the filter set that is currently being simulated. The frame element also maintains a *message property*. This message property contains the state of the message at that point in the simulation of the execution.

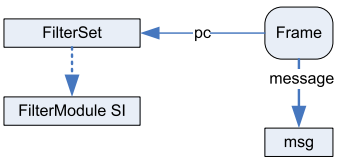


Figure 5.4: A frame element is used to simulate the execution.

At the start of the simulation the program counter property is initialized to the `FilterSet` node. The message property is initialized to the *entrance message*. The entrance message is the message that enters the filter set.

5.2.2 Execution Steps

A step in the execution of the filter set is simulated as a change of the program counter property. The program counter property is changed to the next flow node. If the message changes during an execution step, the message property is also updated. Figure 5.5 gives an example of an execution step. This figure shows that during the execution step the program counter edge of the frame is updated to the next flow element. No message substitution takes place, so the message remains the same.

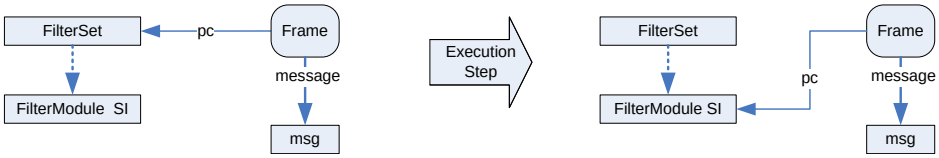


Figure 5.5: Doing an execution step

Most types of flow nodes have only one outgoing edge and do not substitute the message. The simulation of an execution step for these flow nodes is straightforward. It just consists of updating the program counter property to the next flow node, as shown in Figure 5.5. There are, however, a few types of flow nodes that have special behavior. Firstly, there are *branching nodes* like the `ConditionExpression` node. Branching nodes have more than one possible next flow node. Secondly, there are *substitution nodes*. Substitution nodes might change the message property. Finally, there are *action nodes*. Action nodes indicate that a certain filter action needs to be executed. How the execution of each of these special types of flow nodes is simulated, is explained next.

5.2.2.1 Branching Steps

There are three types of branching nodes: the `ConditionExpression` node, the `SignatureMatching` node and the `NameMatching` node. If a `ConditionExpression` node is encountered in the simulation, the condition expression is not evaluated. Why condition expressions are not evaluated is explained in Section 5.5.1. Because condition expressions are not evaluated, both outgoing edges can be taken, as can be seen in Figure 5.6. The constants `True` and `False` form an exception to this rule. If the condition expression is one of these constants, only the corresponding outgoing edge will be taken in the simulation.

If a `NameMatching` node is encountered in the simulation, the name matching can be evaluated, because the frame maintains the current message. In this case, only one execution step is possible. This execution step is either to the `flowMatch` exit node or to the `flowNoMatch` exit node, based on whether the current message matches or not.

If a `SignatureMatching` node is encountered, first it is checked whether the selector matches. If the selector does not match, the signature matching part always rejects. In this case, the only execution step possible is to the `flowNoMatch` exit node. If the selector

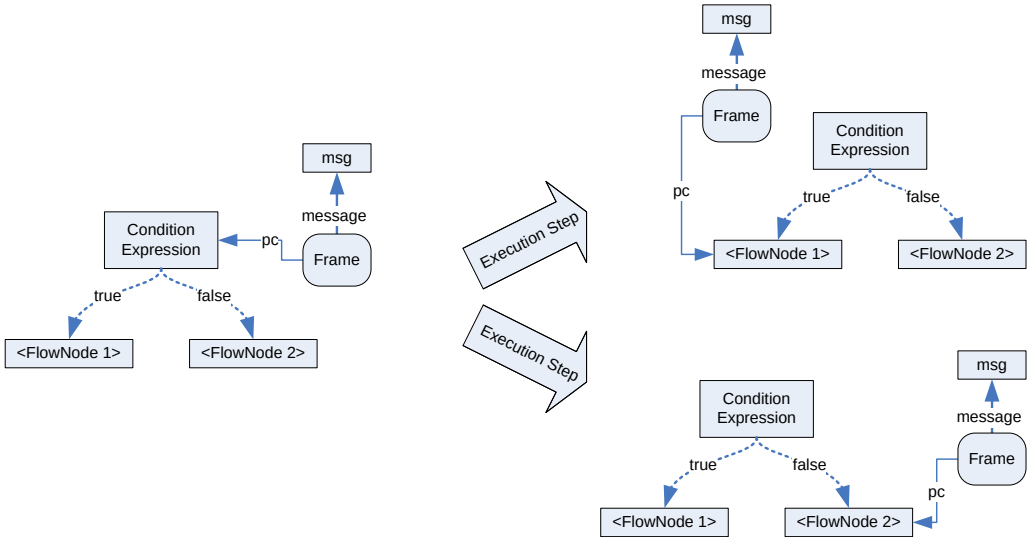


Figure 5.6: Branching execution steps

does match, the next step is checking whether the selector is in the signature of the matching target. This check can be done if signature information is already available. This leads always to one possible execution step: based on whether the selector is in the matching target, either to the `flowMatch` exit node or to the `flowNoMatch` exit node. But the signature information is generated using filter reasoning, as will be explained in Chapter 8. Therefore, signature information is not always available. If signature information is not yet available, the simulation procedure assumes that the signature matching can accept as well as reject. In this case, both possible execution steps are taken in the simulation.

5.2.2.2 Substitution Steps

According to the composition filter semantics, when execution encounters a substitution part, the current message is substituted with the message in the substitution part [2]. But it is actually the action that is executed after the substitution part that decides whether the message remains substituted or not. A `Before` action, for example, uses the substituted message to indicate the method containing the `Before` advice. After the `Before` action, the execution continues with the original message.

To implement this in the simulation procedure, a working copy of the message is created after the simulation of a substitution part. The substitution is applied to this working copy of the message. The working copy is stored in the `substitutionMessage` property of the frame element. This is shown in Figure 5.7. This figure shows the situation in which the substitution part does not contain a wildcard. If the target of the substitution part is the wildcard, then the target of the working copy is the target of the original message. If the selector of the substitution part is the wildcard, then the selector of the working copy is the selector of the original message.

The next flow node after a substitution part is always an action node. The action node might use the working copy of the message to change the original message. This

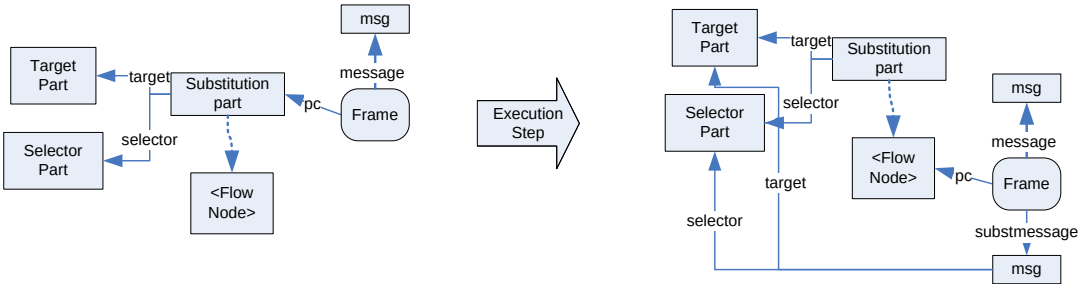


Figure 5.7: Handling the substitution part

is explained next.

5.2.2.3 Action Steps

The simulation of the execution of a filter set inevitably encounters *action nodes*. Because we are only interested in the behavior of a message in the filter set itself, we only simulate how the filter action changes the behavior of the message in the filter set. Because the flow semantics is already determined, we only simulate how a message is changed by a filter action. The part of the filter action semantics that is not important for the simulation of the filter set is just abstracted away into a single execution step. For example, if a `Before` action is encountered, the execution of the `Before` advice is not simulated. The simulation procedure only takes into account how the `Before` action changes the message, because this changes the flow behavior of the message in the filter set.

Most filter actions do not change the message. Examples of such filter actions are the `Wait` action, the `Before` action and the `Error` action. After such an action, the simulation procedure removes the `substitutionMessage` property from the frame element, because no next element uses this working copy anymore.

There are two filter actions that can change the message: the `Substitution` action and the `Meta` action. The `Substitution` action changes the original message to the working copy. This is simulated by setting the `message` property of the frame element to the `substitutionMessage` property. The `substitutionMessage` property is removed, because it is no longer needed.

The `Meta` action does not use the substitution part to change the message, but changes the message in its own advice code. Because this code is outside the filter set, we cannot determine exactly how a message is changed during a `Meta` action by just looking at the filter set specification. A `Meta` action might change a message into any other message or it might never change the message at all. Because we do not know how the `Meta` action changes the message, this forms a problematic uncertainty in filter reasoning. If only the information about the filter set is used, two options are available. We can either assume that a `Meta` action can change the message into any other message or we can assume that a `Meta` action does not change the message at all. The first assumption is too wide, leading to simulated executions that might never happen in practice. The second assumption is too narrow, ignoring executions that might happen in practice. Of both assumptions, there is no one better than the other. For some applications, the first

assumption is better that the second. For other applications the second is better than the first. Section 5.5.2 further elaborates on this problem. It discusses which assumption is preferred under which circumstances and investigates other ways to solve this problem.

When a new filter actions is added to the system, the substitution behavior needs to be specified. The possibilities are:

Original The message is not changed.
Substitution The message is changed to the working copy.
Any The message might be changed an any other message.

5.2.3 Maintaining State

The execution of a message in the filter set can now be simulated. To use the information obtained during the simulation, it needs to be stored in some suitable format. This section explains how this is done.

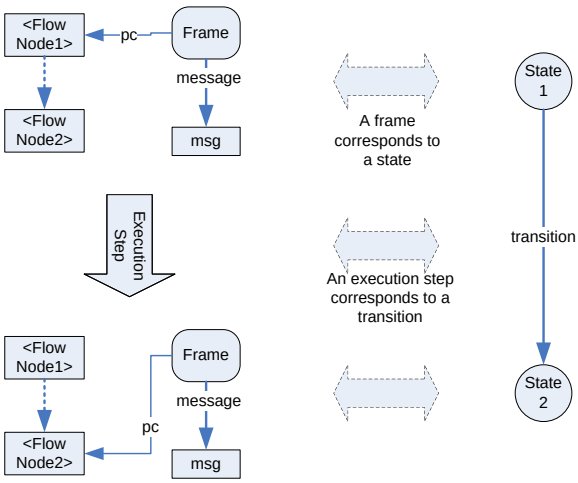


Figure 5.8: Creating a state space during the simulation.

Figure 5.8 shows how a state space is created during the simulation of a filter set execution. Only the properties of the frame element change during the simulation of an execution step. Therefore, the properties of the frame element represent at each point in the simulation the state of the execution. During the simulation of an execution step, the program counter property of the frame element is updated to the next flow node. The message property might also be changed. These changes in the properties of the frame element lead to a new state. Therefore, an execution step is a transition between two states.

If the properties of the frame element have the same value at two different points in the simulation, this is the same state. If an already existing state is encountered again in the simulation of a filter set execution, the simulation does not process further from that state, because this has already been done.

So, a complete simulation of all possible executions of a message in the filter set leads to a state space containing all states that can be reached and all transitions that

can be done between states. This state space forms the representation of the complete simulation of the filter set execution.

5.3 Simulating with each Possible Message

The previous section described how the execution of a filter set can be simulated for a single message. The result of this simulation is a state space representing the execution of the filter set for that single message. Beside a simulation of the filter set for a single message, we also want a simulation of the filter set for every possible message. This gives a complete state space of the filter set. A state is in this state space if and only if there is a certain message for which there is a certain execution that can reach the state. This state space is, for example, needed for consistency reasoning and behavioral reasoning. For these applications, the state spaces for individual messages are not sufficient.

The number of possible messages is infinite. Therefore, creating the complete state space is not a trivial task. Because of the infinity of the number of messages, the state spaces for each individual message cannot just be combined. This section explains how a complete simulation can be created, resulting in a complete state space.

5.3.1 Different Messages that Behave the Same

The key to creating a complete simulation for every possible message is finding sets of messages that have the same flow behavior in the filter set. If two messages have the same flow behavior, they also have equivalent state spaces. The set of messages that all have the same flow behavior forms an *equivalence class* of messages. If the set of all equivalence classes is finite, we can use one message from each equivalence class to create the complete state space.

5.3.1.1 Flow Behavior Equivalence

As explained in Section 5.2, the behavior of a message is comprised of all possible executions. All possible executions means all possible paths through the flowchart in the simulation. So, if two messages have the same set of possible paths through the flowchart, they also have equivalent flow behavior. The following definition formalizes *flow behavior equivalence*.

Definition 5.3.1 (Flow behavior equivalence) *Two messages M_1 and M_2 are* flow behaviorally equivalent *if and only if the set of execution paths of M_1 is the same as the set of execution paths of M_2:*

$$M_1 \equiv M_2 \Leftrightarrow executionPaths(M_1) = executionPaths(M_2) \tag{5.1}$$

Because we are only interested in how messages flow through the filter set, we only look at the flow behavior in the filter set. If two messages are flow behaviorally equivalent, they do not necessarily have the same externally visible behavior. For example, Listing 5.2 shows a filter set in which all messages have the same flow behavior. But the advice method executed by the `Before` action might behave differently for different messages. So, the flow behavior is the same, but the externally visible behavior might be different.

```
before : Before = { True => [*.*] inner.before };
```

Listing 5.2: Example of same flow behavior, but different external behavior

On the other hand, there are also filter sets for which two different messages have the same externally visible behavior, but have different flow behavior in the filter set. An example of this is shown in Listing 5.3. In this example, both message `p` and message `q` are dispatched to `inner.q`. So, the externally visible behavior is the same. But the flow behavior within the filter set is different; the first filter accepts for message `p`, while it rejects for message `q`.

```
subst : Substitution = { True => [*.p] *.q };
disp  : Dispatch     = { True => [*.q] inner.q };
```

Listing 5.3: Example of same external behavior, but different flow behavior

5.3.2 Identifying Equivalence Classes

The previous section defined flow behavior equivalence. This section explains how the set of all possible message can be divided into a finite set of equivalence classes.

The only nodes in a flowchart where two message can differ in behavior are the branching nodes. In the other nodes there is no choice, so all message behave the same. There are two types of branching nodes; the `ConditionExpression` node and the `MatchingPart` node. The `ConditionExpression` node also does not make two messages differ in behavior, because the truth value of a condition expression does not directly depend on the value of the message. The implementation of a condition variable might make use of the message, but this is not visible in the filter set. Therefore, it is assumed that conditions are independent of messages. So, the only type of node in which two message can differ in behavior is the `MatchingPart` node, both the `NameMatchingPart` node and `SignatureMatchingPart` node.

5.3.2.1 Simplified Message Concept

To make the following explanation easier to understand, we first assume that a message consists of just a single element and not of a target element and a selector element. From this it follows that we also do not have signature matching. We will later expand the explanation to the full message concept.

A matching part might consist of the wildcard operator. When this is the case, all messages accept and so all messages behave the same. A matching part can also consist of a unique message. When this is the case, only messages that are the same as the message in the matching part are accepted by that matching part. All other messages are rejected. So, that specific message behaves differently from all other messages. Such a message is called a *distinguishable message*.

Definition 5.3.2 (Distinguishable message) *A message is a distinguishable message of a filter set if there is a matching part that uniquely matches that message. The set of distinguishable messages is represented by* M_{dist}

Because the flow behavior of a distinguishable message at one specific matching part is different from the flow behavior of all other messages at that specific matching part, the flow behavior of a distinguishable message in the filter set is different from the flow

behavior of all other messages in the filter set. So, a distinguishable message forms an equivalence class consisting of a single message.

Not all messages are distinguishable. There are messages for which there is no matching part that uniquely matches that message. Such a message is called an *undistinguishable message.*

Definition 5.3.3 (Undistinguishable message) *An undistinguishable message is a message that is not distinguishable. From this it follows that there is no matching part that uniquely matches that message.*

We will now prove by contradiction that the set of undistinguishable messages forms one equivalence class. Suppose that the set of undistinguishable message can be divided into more than one equivalence class. We now take two of these equivalence classes: C_1 and C_2. Out of each of these equivalence classes we take a message: M_1 respectively M_2. Because M_1 and M_2 are from different equivalence classes, they differ in flow behavior. Previously, we have explained that the flow behavior of a message differs from another message if there is a matching part at which the flow behavior of that message differs from the flow behavior of the other message. From this it can be concluded that there is a matching part at which the flow behavior of M_1 is different from the flow behavior of M_2. Lets assume that M_1 accepts and M_2 rejects at that matching part. This matching part cannot be the wildcard, because then all message would behave the same. So, the matching part must match a specific message. Then M_1 can only accept at this matching part if it is that specific message. But then M_1 is a distinguishable message and not an undistinguishable message. Therefore, all undistinguishable messages behave the same in the filter set: they accept at wildcard matching parts and reject at the other matching parts.

Theorem 5.3.1 (Undistinguishable message equivalence) *All undistinguishable message have the same flow behavior in the filter set. Therefore, they are part of the same equivalence class. This equivalence class of undistinguishable messages is represented by the placeholder '_'.*

Summarizing, we have the following equivalence classes:

- Each distinguishable message forms its own equivalence class. The representative message for this equivalence class is the distinguishable message.
- All undistinguishable message form together one equivalence class. The representative message for this equivalence class is the placeholder '_'.

Because a filter set is a finite structure, there are always a finite number of distinguishable message. Therefore, the number of equivalence classes is also finite. So, the equivalence classes can be used to create the complete state space.

To get the complete state space, the state spaces for a representative of each equivalence class of messages are created. These state spaces are combined to form the complete state space.

It should be noted that our approach to divide the messages into equivalence classes might not result in true equivalence classes but in *pseudo-equivalence classes.* This means that there might be two equivalence classes in which the messages have the same flow behavior. So, these two equivalence classes should actually be one equivalence class. This happens, for example, if a matching part that matches a specific message is not

reachable. Because that specific message cannot reach the matching part, its behavior is not distinguished. Listing 5.4 gives an example of this issue. In this example, message `q` is marked as a distinguishable message, because the matching part in the second filter matches on this message. But because the first filter dispatches all messages, the matching part in the second filter is not reachable. Therefore, the flow behavior of message `q` is not different from the flow behavior of the other messages. For our algorithm this forms no problem. It only introduces redundancy and not the exclusion of states. Actually, the only way to detect this is to do filter reasoning.

```
disp1 : Dispatch = { True => [*.*] inner.* };
disp2 : Dispatch = { True => [*.q] inner.q };
```

Listing 5.4: Example leading to pseudo-equivalence classes

5.3.2.2 Full Message Concept

To extend the approach to the full message concept, we extend the definition of a distinguishable message to targets and selectors:

Definition 5.3.4 (Distinguishable target) *A target is a distinguishable target of a filter set if there is a matching part in which the target part equals the given target. The set of distinguishable targets is represented by* T_{dist}

Definition 5.3.5 (Distinguishable selector) *A selector is a distinguishable selector of a filter set if there is a matching part in which the selector part equals the given selector. The set of distinguishable selectors is represented by* S_{dist}

To get the set of representative messages for all equivalence classes, the cross product of the target set and the selector set is taken. The symbol '_' is used as a placeholder for an undistinguishable target as well as a placeholder for an undistinguishable selector:

$$messages = (T_{dist} \bigcup \{\_\}) \times (S_{dist} \bigcup \{\_\})$$

This results in a set containing the following messages:

- The *fully distinguishable messages*: the set of messages consisting of a distinguishable selector and a distinguishable target.
- The *partially undistinguishable messages*: The set of messages in which either the selector or target is distinguishable and the other is the undistinguishable placeholder.
- The *fully undistinguishable message*: The message for which both the target and the selector is the undistinguishable placeholder.

The simulation is done for all these messages and the state spaces are combined to form the complete state space.

5.3.3 State Space Example

If we apply the procedure to the example in Listing 5.1 on page 52, we get the state space in Figure 5.9. This figure shows the combination of four different state spaces for the message `inner.a`, `_.a`, `inner._` and `_._`.

Three parts of the state space are enlarged. The top part shows that the first transition from the start state chooses the specific message. This transition is targeted to the entrance state of the state space corresponding to that specific message. The middle part shows how the simulation branches after a condition expression state. This is the condition expression in the `Error` filter. The figure shows that if the `false` branch is taken, an `Error` action is executed and the simulation stops at an `Exit` state. If the `true` branch is taken, simulation continues in the filter set. The bottom part shows matching at a signature matching part for two messages. The left part shows that both branching steps are taken for message `_.a`, because the signatures have not been generated yet. The right part shows that the simulation only takes the `noMatch` branch for message `inner._`, because the selector does not match.

5.4 Computational Complexity

The previous sections explained the filter reasoning procedure. To be useful in practice, the filter reasoning procedure should be efficient. This section investigates the computational complexity of the filter reasoning algorithm.

5.4.1 From AST to Flowchart

Transforming the AST to a flowchart can be seen as adding flow edges to the AST. The transformation rules presented in Section 5.1 took a top-down approach to transform the AST into a flowchart. This approach started with the top-level `FilterSet` node and worked its way down the tree, incrementally transforming the AST into a flowchart. Table 5.1 shows for each rule the number of computation steps for one application of the rule, the total number of applications of the rule and the total computation steps for all applications of the rule. With a computation step is meant the addition or removal of a flow edge. Each entry in the table is explained. The table uses the following conventions:

- $\#F^{(FM)}$ means the number of filters in the specific filter module of that application of the rule.

Rule	One application	# applications	All applications
Filter set rule	$\#FM^{(FS)} + 4$	1	$\#FM + 4$
Conditional SI rule	5	$\#FM$	$5 \cdot \#FM$
Filter module rule	$\#F^{(FM)} + 2$	$\#FM$	$\#F + 2 \cdot \#FM$
Filter rule	$2 \cdot \#FE^{(F)} + 10$	$\#F$	$2 \cdot \#FE + 10 \cdot \#F$
Filter element rule	9	$\#FE$	$9 \cdot \#FE$
Matching pattern rule	$2 \cdot \#MP^{(MPtrn)} + 4$	$\#FE$	$2 \cdot \#MP + 4 \cdot \#FE$
Action rule	2	$4 \cdot \#F$	$8 \cdot \#F$

Table 5.1: Computational steps when transforming an AST to a flowchart

Filter set rule

The filter set rule does the following transformations:

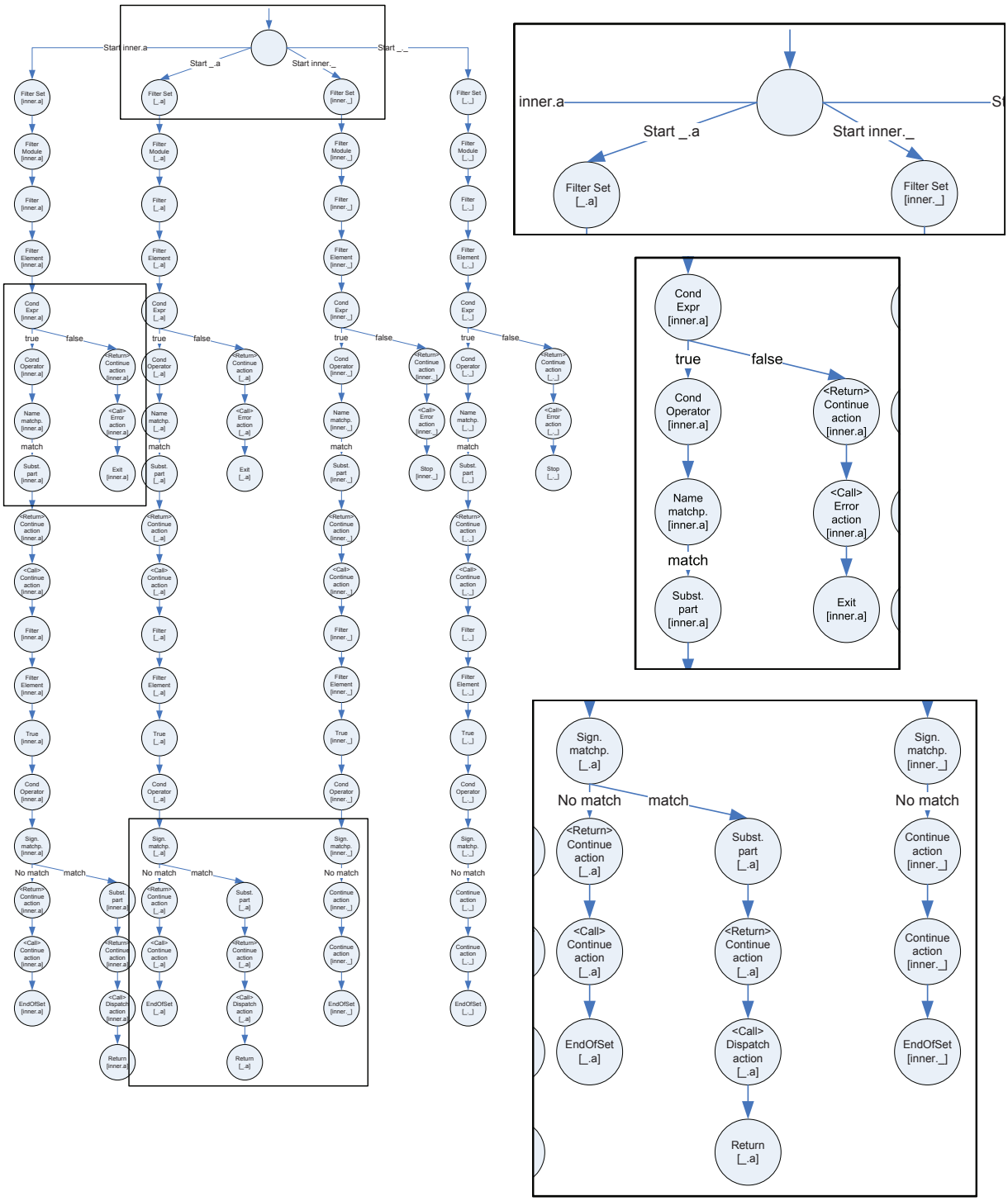


Figure 5.9: Example of a state space

- It adds a flow edge from the `FilterSet` node to the first `FilterModuleSI` node. This is 1 step.
- It adds flow edges between the `FilterModuleSI` nodes. This are $\#FM^{(FS)} - 1$ steps.
- It adds an `End` node, a `Stop` node and an `Exit` node. This costs 3 steps.
- It adds a flow edge between the last `FilterModule` node and the `End` node. This costs 1 step.

So, the total number of transformations performed in one application of the filter set rule is $\#FM^{(FS)} + 4$. The filter set rule is applied only one time, because there is only one filter set. Therefore, the total number of transformations is $\#FM + 4$, where $\#FM$ is the total number of filter modules in the AST.

Conditional superimposition rule

The conditional superimposition rule does the following transformations:

- It adds a flow edge from the `FilterModuleSI` node to the `ConditionExpression` node. This costs 1 step.
- It adds two flow edges from the `ConditionExpression` node. This costs 2 steps.
- It adds a flow edge from the `FilterModule` node. This costs 1 step.
- It removes the flow edge between the `FilterModuleSI` node and the $\langle FlowNode2 \rangle$ node. This costs 1 step.

So, the total number of transformations performed in one application of the conditional superimposition rule is 5. The conditional superimposition rule is applied as many times as there are filter modules, so $\#FM$ times. Therefore, the total number of transformations is $5 \cdot \#FM$.

Filter module rule

The filter module rule does the following transformations:

- It adds a flow edge from the `FilterModule` node to the first `Filter` node. This is 1 step.
- It adds flow edges between the `Filter` nodes. This are $\#F^{(FM)} - 1$ steps.
- It removes the flow edge between the `FilterModule` node and the $\langle FlowNode2 \rangle$ node. This is 1 step.
- It adds a flow edge between the last `Filter` node and the $\langle FlowNode2 \rangle$ node. This is 1 step.

So, the total number of transformations performed in one application of the filter module rule is $\#F^{(FM)} + 2$. The filter module rule is applied as many times as there are filter modules in the AST, so $\#FM$ times. Therefore, the total number of transformations is $\#F + 2 \cdot \#FM$

Filter rule

The filter rule does the following transformations:

- It adds 4 action nodes. This is 4 steps.

- It adds a flow edge from the `Filter` node to the first `FilterElement` node. This is 1 step.
- It adds an outgoing accept flow edge and an outgoing reject flow edge to every `FilterElement` node. This costs $2 \cdot \#FE^{(F)}$ steps.
- It removes the flow edge between the `Filter` node and the $\langle FlowNode2 \rangle$ node. This is 1 step.
- It adds an outgoing flow edge to each action node. This costs 4 steps.

So, the total number of transformations performed in one application of the filter rule is $2 \cdot \#FE^{(F)} + 10$. The filter rule is applied as many times as there are filters in the filter set, so $\#F$ times. Therefore, the total number of transformations is $2 \cdot \#FE + 10 \cdot \#F$.

Filter element rule

The filter element rule does the following transformations:

- It removes the two outgoing flow edges from the `FilterElement` node. This costs 2 steps.
- It adds a flow edge from the `FilterElement` node to the `ConditionExpression` node. This costs 1 step.
- It adds a one flow edge from the `ConditionExpression` node to the `ConditionOperator` node and one flow edge from the `ConditionExpression` node to the $\langle FlowNode3 \rangle$ node. This costs 2 steps.
- It adds a flow edge from the `ConditionOperator` node to the `MatchingPattern` node. This costs 1 step.
- It adds 2 outgoing flow edges from the `MatchingPattern` node. This costs 2 steps.
- It adds an annotation to the `MatchingPattern` node. This costs 1 steps.

So, the total number of transformations performed in one application of the filter element rule is 9. The filter element rule is applied as many times as there are filter elements in the filter set, so $\#FE$ times. Therefore, the total number of transformations is $9 \cdot \#FE$.

Matching pattern rule

The matching pattern rule does the following transformations:

- It removes the two outgoing flow edges from the `MatchingPattern` node. This costs 2 steps.
- It adds a flow edge from the `MatchingPattern` node to the first `MatchingPart` node. This costs 1 step.
- It adds two outgoing flow edges from each `MatchingPart` node. This costs $2 \cdot \#MP^{(Mptrn)}$ steps.
- It adds one outgoing flow edge to the `SubstitutionPart` node. This costs 1 step.

So, the total number of transformations performed in one application of the matching pattern rule is $2 \cdot \#MP^{(Mptrn)} + 4$. The matching pattern rule is applied as

many times as there are matching patterns in the filter set. This is equal to the number of filter elements. So, the rule is applied $\#FE$ times. Therefore, the total number of transformations is $2 \cdot \#MP + 4 \cdot \#FE$.

Action rule

Some action rules change the end of the outgoing flow edge. This costs 2 steps. There are four times as many action nodes as there are filter nodes. Therefore, the total number of transformations is $8 \cdot \#F$.

So totally we get:

$$\#Steps = 8 \cdot \#FM + 19 \cdot \#F + 15 \cdot \#FE + 2 \cdot \#MP + 4 < 19 \cdot \#nodes + 4$$

So, the time complexity of the algorithm to transform the AST into a flowchart is $O(\#nodes)$. Because the number of nodes in the AST is linear to the size of the filter set, the time complexity of the transformation is linear to the size of the filter set.

5.4.2 Simulating the Execution

We will now calculate the time complexity of the simulation. We will first calculate the maximum number of states in the state space. A state corresponds to the properties of the frame element. These properties are the program counter property, containing the current flow node, and the message property. So, the number of states is the number of nodes in the flowchart multiplied with the number of messages that can occur:

$$O(\#States) = O(\#FlowNodes \cdot \#Messages)$$

Where the number of messages is defined as the product of the number of possible targets with the number of possible selectors:

$$O(\#Messages) = O((\#DistinguishableTargets^{ext}+1)\cdot(\#DistinguishableSelectors^{ext}+1))$$

$DistinguishableTargets^{ext}$ includes not only the targets in the matching parts but also the targets in the substitution parts. The same applies to $DistinguishableSelectors^{ext}$. This is needed, because the target or the selector of a message might change through substitution to a target or selector from a substitution part.

Each state has at most two outgoing transitions. So the number of transitions is:

$$O(\#Transitions) = O(\#States)$$

In the simulation each state is processed once. A state can be reached through different paths. Therefore, when the simulation reaches a state, a check is done whether this state is already processed. Using hashtables this can be implemented efficiently, costing approximately constant time. If this check finds that the state already has been processed, it is not processed again. If we would not do this check this would lead to worser time complexity, because every execution path is completely processed. Because of branching, there are potentially exponentially many execution paths possible through

the filter set. Processing each complete execution path is not needed, because if two execution paths reach the same state, the execution after this state is the same for both execution paths. Therefore, the execution after that state only needs to be processed once.

So the time complexity to generate the execution model is:

$$O(c_1 \cdot \#States)$$

Where c_1 is a constant indicating the time it costs to process the two outgoing transitions and check whether the resulting states already have been processed. This is equivalent to:

$$\begin{aligned} &O(c_1 \cdot \#States) \\ &= O(\#States) \\ &= O(\#FlowNodes \cdot \#Messages) \end{aligned}$$

5.4.2.1 Relation Between the Number of Messages and the Number of Flow Nodes

The set of possible messages is created from the targets and selectors from the matching parts and substitution parts. Therefore, the number of these messages is related to the number of matching parts and substitution parts. So, the number of messages is related to the size of the flowchart. For the maximum number of messages the following holds:

$$O(\#Messages) = O(\#FlowNodes^2)$$

The power-two comes from the fact that both the number of targets and the number of selectors are linear in the size of the flowchart and the set of messages is the cross product between the set of targets and the set of selectors. So, the number of states in the state space is worst case:

$$O(\#States) = O(\#FlowNodes \cdot \#FlowNodes^2) = O(\#FlowNodes^3)$$

But it should be noted that it is a rare case in which this relation applies. It is only the case when in almost any matching part and substitution part a different target and selector is used. In reality, many matching parts and substitution parts use the same target and selector as other matching parts or substitution parts. In this case, the relationship between the number of states and the number of flownodes is closer to linear.

5.4.3 Total Time Complexity

The total time complexity is the combination of the time complexity of the transformation step and of the simulation step. The time complexity of the transformation step is $O(\#FlowNodes)$. The time complexity of the simulation is worst case $O(\#FlowNodes^3)$. So, the total time complexity is worst case $O(\#FlowNodes^3)$.

Because the number of flow nodes in the flowchart is linear to the size of the filter set, it can be concluded that the filter reasoning procedure has polynomial time complexity.

5.5 Discussion

This section discusses several problems and issues concerning the simulation of the execution of a filter set. It first discusses if condition expressions also need to be evaluated. Then it discusses how to cope with the `Meta` filter uncertainty. This is followed by the target matching problem. Finally, message lists are discussed.

5.5.1 Evaluating Condition Expressions

The message-action tree approach explained in Section 4.4.2 also evaluates condition expressions in the reasoning algorithm. It thereby assumes that condition variables remain constant during the execution of the filter set. The proposed message flow simulation approach does not do this. This section explains advantages and disadvantages of not evaluating condition expressions. It also investigates the validity of the assumption that condition variables remain constant during the execution of the filter set.

Not evaluating condition expressions means that the simulation assumes that a condition expression can always be true as well as false. The problem with this assumption is that it does not take into account the influence of the truth value of earlier condition expressions on the truth value of a condition expression later in the execution path. This might lead to execution paths that in practice can never occur.

```
error : Error    = { C => [*.*] };
disp  : Dispatch = { C => [*.a] *.b, [*.*] }
```

Listing 5.5: Not taking conditions into account leads to more execution paths

Take for example the filter set in Listing 5.5. The message-action tree approach finds for this filter set that message `a` either gives an error or is dispatched to method `b`. The message flow simulation approach, on the other hand, finds for this filter set that message `a` either gives an error, is dispatched to method `b` or is dispatched to method `a`. The difference is caused by the fact that if message `a` reaches the `Dispatch` filter, the condition expression `C` always has to be `true`. If `C` was not `true`, an error would have been given in the `Error` filter. The message-action tree approach takes this into account when reasoning about the first filter element of the `Dispatch` filter. It finds that the condition expression `C` is also always `true`. So, if the `Dispatch` filter is reached, the first filter element of that filter always matches for message `a`. This causes message `a` to be always dispatched to `b`. The message flow simulation approach does not take into account that `C` is `true` when the `Dispatch` filter is reached. It assumes that the condition expression in the first filter element of the `Dispatch` filter can be `true` as well as `false`. So, message `a` can be dispatched to `b` as well as to `a`.

```
error1 : Error    = { C1 | C2 => [*.*] };
error2 : Error    = { C1 | C3 => [*.*] };
disp   : Dispatch = { C1 | (C2 & C3) => [*.a] *.b, [*.*] }
```

Listing 5.6: A more elaborate example of how not taking conditions into account leads to more execution paths

Listing 5.6 gives another example, leading to the same outcome for message `a` as in the previous example. This example is, however, more elaborate. The fact that the

third condition expression is always `true` is implied by the combination of the first two condition expressions being `true`, instead of just one of them.

So, in general we have the following situation: if a filter element with condition expression ψ is reached in an execution path in which the condition expressions $\phi_1, \phi_2, \ldots, \phi_n$ were true (false condition expressions are negated), then we have to check whether one of the following equations holds:

$$\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$$

Or

$$\phi_1, \phi_2, \ldots, \phi_n \vDash \neg\psi$$

The operator $\vDash$ stands for *semantic entailment*: If the formula's on the left are `true`, the formula on the right is also `true`.

If the first equation holds, the condition expression ψ is always `true` in the given execution path. If the second equation holds, the condition expression ψ is always `false` in the given execution path. If none of the equations hold, the truth value of ψ does not dependent on $\phi_1, \phi_2, \ldots, \phi_n$. So, the condition expression can be `true` as well as `false` in the given execution path.

So how hard is checking this? Checking whether

$$\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$$

holds is equivalent to checking whether

$$\vDash \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n \rightarrow \psi$$

holds. But this is equivalent to checking whether the formula on the right is a *tautology*. This is a CO-NP complete problem [50].

So, evaluating condition expressions always makes the algorithm worst case exponential in the number of condition variables. This conflicts with the requirement of being an efficient algorithm. The message-action tree approach implemented the checking of condition expressions by doing the analysis for every valuation of the condition variables. This made the time-complexity of the algorithm always power 2 exponential in the number of condition variables.

5.5.1.1 Do we Really Need to Take Conditions into Account?

Taking conditions into account was based on the assumption that the valuation of the condition variables remained constant during the execution of the filter set. This might, however, not be the case. For example, other threads might change the state of the system, thereby influencing the valuation of the condition expressions. Even the execution of a filter action in the filter set itself might be responsible for changing the valuation of the condition variables. To prevent this, we could take a snapshot of the valuation of all condition variables before starting the execution of the filter set and use this snapshot during the execution. But this may not be desirable. We might want the flexibility of one filter action to be able to influence the execution of the following filters in the filter set and conditions seem a nice way to do this. Also, certain filter actions require that conditions can be changed by other threads. The `Wait` action is an example of such a filter action.

5.5.2 The Meta Filter Uncertainty

The `Meta` action has the ability to change the message and to manipulate the flow of the message. This happens in the implementation code of the `Meta` action. Therefore, the reasoning algorithm cannot predict the behavior of the `Meta` action by just looking at the filter set. This introduces uncertainty into the algorithm.

5.5.2.1 Message Substitution Uncertainty

A `Meta` action has the ability to change the message. It can change the message in any other message. This is done in the implementing method and not in the filter set. Therefore, it cannot be known how a `Meta` action changes the message by just looking at the filter set. This introduces uncertainty into the reasoning algorithm. If a `Meta` action is encountered during the simulation, the simulation procedure cannot predict whether the `Meta` action changes the message and how it changes the message. There are four options to cope with this uncertainty:

- We can assume that the message is not changed at all. A result of this is that certain possible execution paths are not present in the execution model. This may lead to problems with the detection of conflicts in consistency reasoning and behavioral reasoning. Certain conflicts might be detected that never occur in practice, while other conflicts that might occur are not detected.

- We can assume that the message might be changed in any other message. This may result in many execution paths in the execution model that never occur in practice if the `Meta` action does not change the message into any other message. This leads to problems with signature generation and type checking. For example, if a `Dispatch` action occurs after a `Meta` action, the message might be dispatched to any method, because we assume that the `Meta` action might have changed the message to any other message. In many cases these methods do not exist. This results in many type errors being given.

  Consistency reasoning and behavioral reasoning have the same problems as with the first option. But now the problems are caused by execution paths that can never occur in practice.

- We can use annotations on the `Meta` method, describing how it might change the message, and use these annotations in reasoning. Problem with this is that the annotation might not correspond to the actual implementation.

- We can do a semantic analysis of the implementation code to find out how the `Meta` action changes the message. This, however, is a difficult task, just starting to be explored [53]

Preventing Meta from Changing the Message We can prevent the uncertainty if we can strictly define the message substitution behavior of the `Meta` action in the filter set. This can be done by using a `Substitution` filter after the `Meta` filter. The `Substitution` filter can change the message in the desirable way based on condition variables that are influenced by the `Meta` action. If we also prevent that `Meta` actions can

directly change a message, the behavior is fully defined in the filter set specification and there is no uncertainty anymore. An example of this approach can be seen in Listing 5.7. This listing shows that the `Meta` action can indirectly change message `a` into message `b`, `c` or `d` by setting respectively `C1`, `C2` or `C3`. To be able to do this we need the assumption that condition variables can change during the execution of the filter set.

```
meta  : Meta         = { [*.a] *.metamethod };
subst : Substitution = { C1 => [*.a] *.b, C2 => [*.a] *.c,
                         C3 => [*.a] *.d }
```

Listing 5.7: An example of a `Meta` action changing a message by using a `Substitution` filter

5.5.2.2 Flow Behavior Uncertainty

A `Meta` action can also influence the flow in the filter set. The reasoning procedure assumes that flow continues, but this might not be the case. Beside a `Proceed`, a `Meta` action can, for example, also do a `Reply`. In this case, the evaluation of the filter set is not continued.

Just as with the uncertainty in the message change behavior, the uncertainty in the flow behavior can also be coped with by adding annotations to the `Meta` advice or by a semantic analysis of the implementation. But it might also be prevented by replacing it with an explicit `Reply` filter, for example.

5.5.3 Target Matching

Target matching in a filter set execution happens on an instance level. A target of a message matches the target in a name matching part if they represent the same object. In this way, it might happen that the target of a message is substituted to one `external` and later matches another external, because both externals contain the same object. Listing 5.8 shows this in an example. External `t1` and external `t2` both contain the same object. The first filter substitutes the target of the message to `t1`. The second filter tries to match the target on `t2`. This results in a match, because both externals contain the same object.

```
concern A {
  filtermodule FMA {
    externals
      t1 : T = T.instance();
      t2 : T = T.instance();
    inputfilters
      subst : Substitution = { [*.*] t1.* };
      disp  : Dispatch     = { [t2.*] }
  }
```

Listing 5.8: An example of how instance matching leads to the acceptance of a matching part

Instance matching of the targets leads to problems for filter reasoning. No information about the specific instances of externals and internals is known at compile time. Therefore, the reasoning algorithm cannot match the targets on an instance level. Instead, it matches the targets on a name level. In the given example this leads to the

rejection of the `Dispatch` filter, because the message has target `t1` and the matching part tries to match on target `t2`.

5.5.4 Message Lists

A new feature in composition filters are the *message lists*. With the message list approach the 'message' in the filter set execution might be a complete list of messages instead of a single message. Matching and substitution can be evaluated on the complete message list or on individual messages in the list [34].

The problem with this new feature is that the state space might become exponential in size compared to the size of the filter set. Without message lists, the number of states in the state space was:

$$O(\#states) = O(\#flownodes \cdot \#messages)$$

But with the message list feature each individual state has not a single message anymore but a message list. Therefore, the number of states becomes:

$$O(\#states) = O(\#flownodes \cdot \#messagelists)$$

Where the number of message lists is worst case:

$$O(\#messagelists) = O(2^{\#messages})$$

Because each message can be in the list.

```
p1   : Append  = { C1 => [*.*] Logger.log };
p2   : Append  = { C2 => [*.*] Timer.time };
p3   : Append  = { C3 => [*.*] Tracer.trace };
disp : Dispatch = { [*.*] }
```

Listing 5.9: An example of a message list filter set that leads to an exponential number of states

Listing 5.9 shows an example of a filter set that leads to exponentially many states. This filter set has three `Append` filters that all conditionally append a certain message to the message list. Figure 5.10 shows a schematic representation of the execution model: each state represents one filter. The message list and the specific filter are shown in each state.

This figure shows that the execution model branches with each next `Append` filter, because the `Append` filters are conditional. Because the message list is either appended with a message or not, this leads to different message lists in each branch. Therefore, the branches do not come together anymore and the state space is exponential.

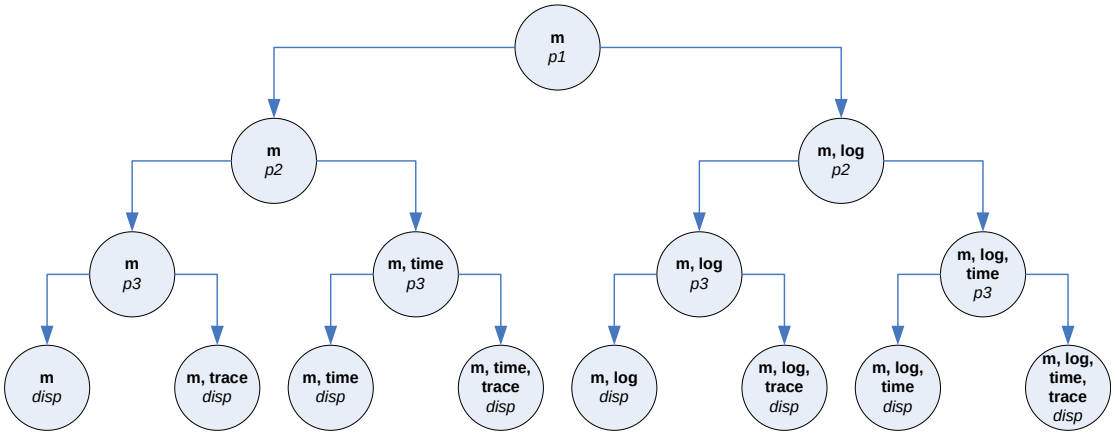


Figure 5.10: Execution model with an exponential number of states.

Chapter 6

The Filter Reasoning Engine

"Irrationally held truths may be more harmful than reasoned errors."
Thomas H. Huxley
English biologist (1825 - 1895)

This chapter describes how filter reasoning is implemented in Compose⋆. First, an overview of the *Filter Reasoning Engine* is given. Next, each component in the Filter Reasoning Engine is explained. Finally, some implementation issues are discussed.

6.1 Filter Reasoning Engine Overview

The *Filter Reasoning Engine* (*FIRE*) consists of three main components. This section gives an overview of these main components. The upcoming sections explain each component in more detail. Figure 6.1 shows the main components in the Filter Reasoning Engine. These main components are the *FIRE Preprocessor*, the *FIRE Model* and the *FIRE Tools*.

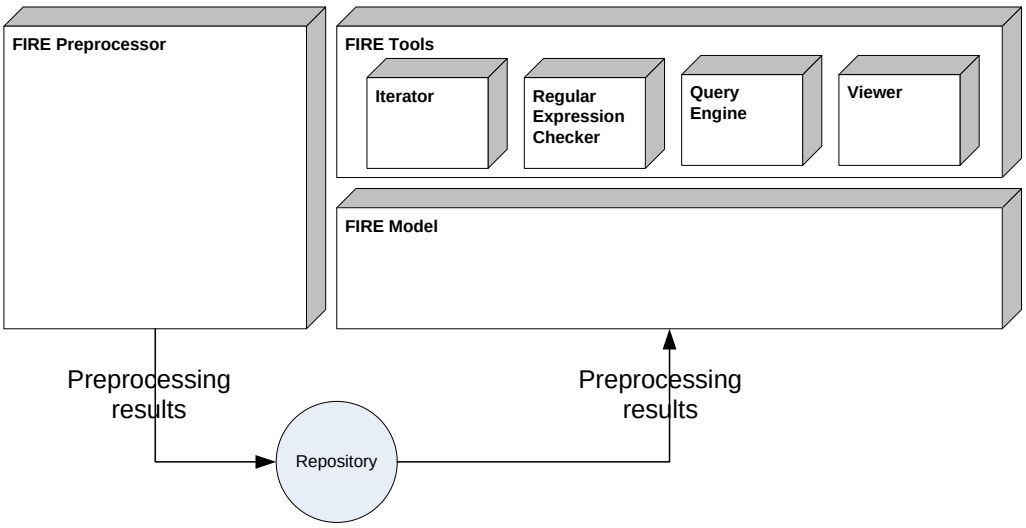


Figure 6.1: Components in the Filter Reasoning Engine

FIRE Preprocessor

The *FIRE Preprocessor* component generates the reasoning information. It needs to be executed before any other module can access the filter reasoning results. For each filter module in the repository, it transforms the abstract syntax tree of the filter module to a flowchart and it simulates the execution of the filter module. The resulting flowchart and execution model are stored in the repository.

The transformation and simulation are implemented using graph transformations and the tool GROOVE [43]. Section 6.2 explains why graphs transformations are used and how they are used to implement the filter reasoning procedure.

Section 6.3 describes the complete preprocessing process and explains why this is done on each filter module instead of each filter set.

The FIRE Preprocessor is run at one moment in the compilation process. It should not be used by other modules.

FIRE Model

The *FIRE Model* component provides an interface for other modules to access the filter reasoning results. Through this interface other modules can access the flowchart and execution models of a concern. The FIRE Model component generates this filter reasoning information using the filter reasoning information about the separate filter modules from the repository. How this information is combined is explained in Section 6.4. The FIRE Model interface is explained in Section 6.5.

FIRE Tools

The *FIRE Tools* component provides a number of tools that work on top of the FIRE Model. The tools component contains the following tools:

- Iterators to iterate over the models.
- A query engine to query certain states in an execution model.
- A regular expression checker to match regular expressions on an execution model.
- A viewer to visualize a flowchart or an execution model.

Section 6.6 describes every tool in the FIRE Tools component.

6.2 Using Graphs and Graph Transformations

Graph transformations are a powerful technique to transform one graph like structure into another graph like structure. An example of a tool that implements graph transformations is the tool GROOVE [43]. Because both the abstract syntax tree of a filter set as the corresponding flowchart are graph like structures, graph transformation tools seem a good candidate to implement the transformation rules defined in Section 5.1. Graph transformations can also be used to implement the simulation of the execution, as described in Section 5.2. This section describes how the transformation from AST to flowchart and the simulation of the execution is implemented using graph transformations.

6.2.1 Transforming the AST to a Flowchart

To transform the AST to a flowchart we start with a graph representation of the AST. On this graph a GROOVE production system is applied. This production system contains rules that resemble the rules as explained in Section 5.1. The difference is that in the GROOVE production system the rules are more fine-grained; in GROOVE it is impossible to define a rule that matches a pattern that contains an undefined number of occurrences of a certain subpattern, as was the case in the rules in the theory section. For example, matching and transforming all filters (subpattern) in a filter module (pattern) is not possible in one rule. To do this in GROOVE, the rule needs to be split into subrules which match and transform the subpatterns one by one.

The production system transforms the graph representation of the AST to a graph representation of the flowchart by incrementally applying the transformation rules to the graph until no more rules can be applied. It is possible that in certain states more than one rule can be applied. In this case, it does not matter which rule is applied first. Eventually, the same end state is reached. Therefore, we can use GROOVE's linear traversal algorithm for applying the rules. The linear traversal algorithm applies the rules in one order. This is sufficient, because we are only interested in the final state. Applying the rules in another order leads only to different intermediate states.

6.2.2 Simulating the Flowchart

Upon the resulting graph representation of the flowchart another GROOVE production system is applied that does the simulation. This production system first contains a rule that adds nodes and edges to the graph, representing the frame element with its program counter edge and its message element. This rule is always applied first on the graph representation of the flowchart. The other rules in the production system all represent doing an execution step, as explained in Section 5.2. There is one rule for each specific case. Such a rule updates the program counter edge to the next flow node and changed the message element, if necessary.

When GROOVE applies a production system, it maintains a state space. If a production rule is applied to a graph, this leads to a new graph. GROOVE maintains these graphs as states in a state space. Both the graph before applying the production rule as the graph after applying the production rule correspond to a state in this state space. The application of the production rule corresponds to a transition between two states.

GROOVE does not immediately make a new state after a production rule is applied. It first checks whether there is already an existing state that corresponds to an equivalent graph. GROOVE uses graph isomorphism to check whether two graphs are equivalent. This checking ensures that there are not two states in the state space that actually represent the same state.

The only nodes and edges that change if a production rule is applied in this production system are the frame's program counter edge and message element. Therefore, a state in the GROOVE state space corresponds to the state of the frame. The application of a rule, which corresponds to an execution step, leads to a transition in the state space. So, the state space that GROOVE generates corresponds to the conceptual state space that represents the execution of the filter set, as explained in Section 5.2. To generate the entire state space, GROOVE's full traversal algorithm is applied. This traversal

algorithm applies in each state every possible production rule, as opposed to the linear traversal algorithm, which applies only one possible production rule. In this way, the complete execution model is generated.

6.2.3 Advantages and Disadvantages of using GROOVE

There are both advantages as disadvantages of using Groove.

6.2.3.1 Advantages

- GROOVE facilitates the definition of the transformation system in a way that closely resembles the conceptual rules. When the conceptual language rules change, these changes can be easily incorporated in the GROOVE production system: only the corresponding rules need to be changed, while leaving the other rules unaffected.
- The state space GROOVE generates is equal to the conceptual state space. The generation of the state space in GROOVE is built-in functionality. This does not have to be implemented anymore.

6.2.3.2 Disadvantages

- GROOVE uses graph isomorphism to check whether a new state, after applying a production rule, already exists. The computational complexity of these isomorphism checks is worst case exponential in the size of the graphs. The graphs are linear in size of the filter set, so this might degrade the performance of the algorithm. This problem is explained in depth in Section 6.7.2.
- The information is represented in GROOVE as nodes and edges. This makes the information not easily accessible by other modules. For example, if we want to know the message in a certain state, we first have to find the frame node in the corresponding graph. Then we have to find the outgoing message edge from the frame node and check to which message node this is directed. This is clearly too complex to be used often by other modules. Therefore, the information first needs to be extracted from the GROOVE models into a model that is more usable and is specific to this problem. For example, a state in this model directly contains the message object and the flow node to which the program counter edge is pointing, instead of a graph in which this information is embedded.

6.3 The Filter Reasoning Engine Preprocessor

The previous section explained that GROOVE and graph transformations are used to implement the filter reasoning procedure. It also noted that a disadvantage of GROOVE is that obtaining the information from its graphs is complex. This section explains the *FIRE Preprocessor*. The FIRE preprocessor uses GROOVE to implement the filter reasoning procedure. It also extracts the information from the resulting GROOVE models into a domain specific model.

6.3.1 Overview of the process

Generating and extracting the information is done in the FIRE preprocessing process. Figure 6.2 shows all FIRE preprocessing steps. The dashed lines indicate control flow. The solid lines indicate data flow.

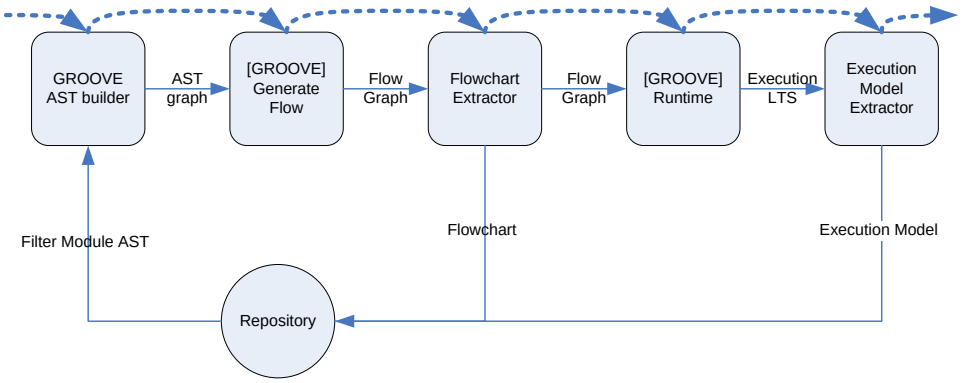


Figure 6.2: The FIRE preprocessing process

GROOVE AST Builder

The *GROOVE AST Builder* creates an *ASTgraph*: a graph representation of the AST of a filter module. It also adds to each graph node a link to the corresponding `RepositoryEntity` object (the original element from the AST). This is needed to maintain the correspondence between the nodes and the `RepositoryEntity` object during the transformation to the *flowgraph* (a graph representation of the flowchart). By maintaining this information, we know for each node in the flowgraph to which `RepositoryEntity` object it corresponds.

The link to the `RepositoryEntity` element is not directly visible in the graph. This is done by making a subclass of GROOVE's `DefaultNode` class. This subclass is called `AnnotatedNode`. Other objects can be attached as annotation to objects of the `AnnotatedNode` class. In a GROOVE production system, `AnnotatedNode` has the same behaviour as the `DefaultNode`. Because GROOVE maintains the node objects during the transformation from AST to flowchart, the corresponding annotations are also maintained. So, by adding the `RepositoryEntity` objects to the corresponding nodes in the ASTgraph, we can link the flow nodes back to the corresponding `RepositoryEntity` objects after the transformation to the flowgraph.

Generate Flow (GROOVE)

In this step, the *Generate Flow* GROOVE production system is applied to the ASTgraph. This leads to a *flowgraph*: a graph representation of the flowchart.

Flowchart Extractor

The *Flowchart Extractor* extracts the information from the flowgraph into a more usable domain specific flowchart and stores this flowchart in the repository. The domain specific flowchart is accessible to other modules through FIRE's interface. It makes the information easier accessible. For example, the corresponding

`RepositoryEntity` object is directly accessible from a `FlowNode` object, instead of being hidden in some added annotation. Section 6.5 explains this model in depth.

Runtime (GROOVE)
The *Runtime* GROOVE production system implements the simulation procedure. It thereby uses the generated flowgraph. The result of this step is a GROOVE *state space*.

Execution Model Extractor
The *Execution Model Extractor* extracts the information from the GROOVE state space into a more usable domain specific *execution model*. This execution model is stored in the repository. Just as with the flowchart, the execution model provides easier access to the information. For example, the message in a state is directly accessible from the state object, instead of being embedded in a graph structure.

6.3.2 Class Structure

Figure 6.3 shows the class structure of the FIRE Preprocessor. It is contained in the package `Composestar.Core.FIRE2.Preprocessing`. Each class is explained next.

Preprocessor
`Preprocessor` is the main class. It iterates over all filter modules, directs all tasks in the preprocessing process and stores the results in the repository.

GrooveASTBuilder, FlowChartExtractor, ExecutionModelExtractor
These classes implement the corresponding tasks in the preprocessing process.

FirePreprocessingResult
This class is the container class to store the results in the repository. It contains the flowchart and the execution model of a filter module for both the input filters as the output filters. The preprocessing results are stored in the dynamic map of the filter module.

AnnotatedNode, AnnotatedEdge
`AnnotatedEdge` and `AnnotatedNode` are subclasses of GROOVE's `DefaultNode` and `DefaultEdge` classes. They contain methods to add annotations. These classes are used by `GrooveASTBuilder` to create the ASTgraph. `GrooveASTBuilder` adds the corresponding `RepositoryEntity` objects as annotation to the nodes in the ASTgraph.

JarGPSGrammar, GPSGrammar
The class `JarGPSGrammar` is a subclass of GROOVE's `GPSGrammar` class. It is used to load a GROOVE grammar that is contained as a resource in a jar-file.

6.3.3 Preprocessing each filter module?

The FIRE preprocessing process is applied on each single filter module instead of on a filter set. This is done for the following reasons:

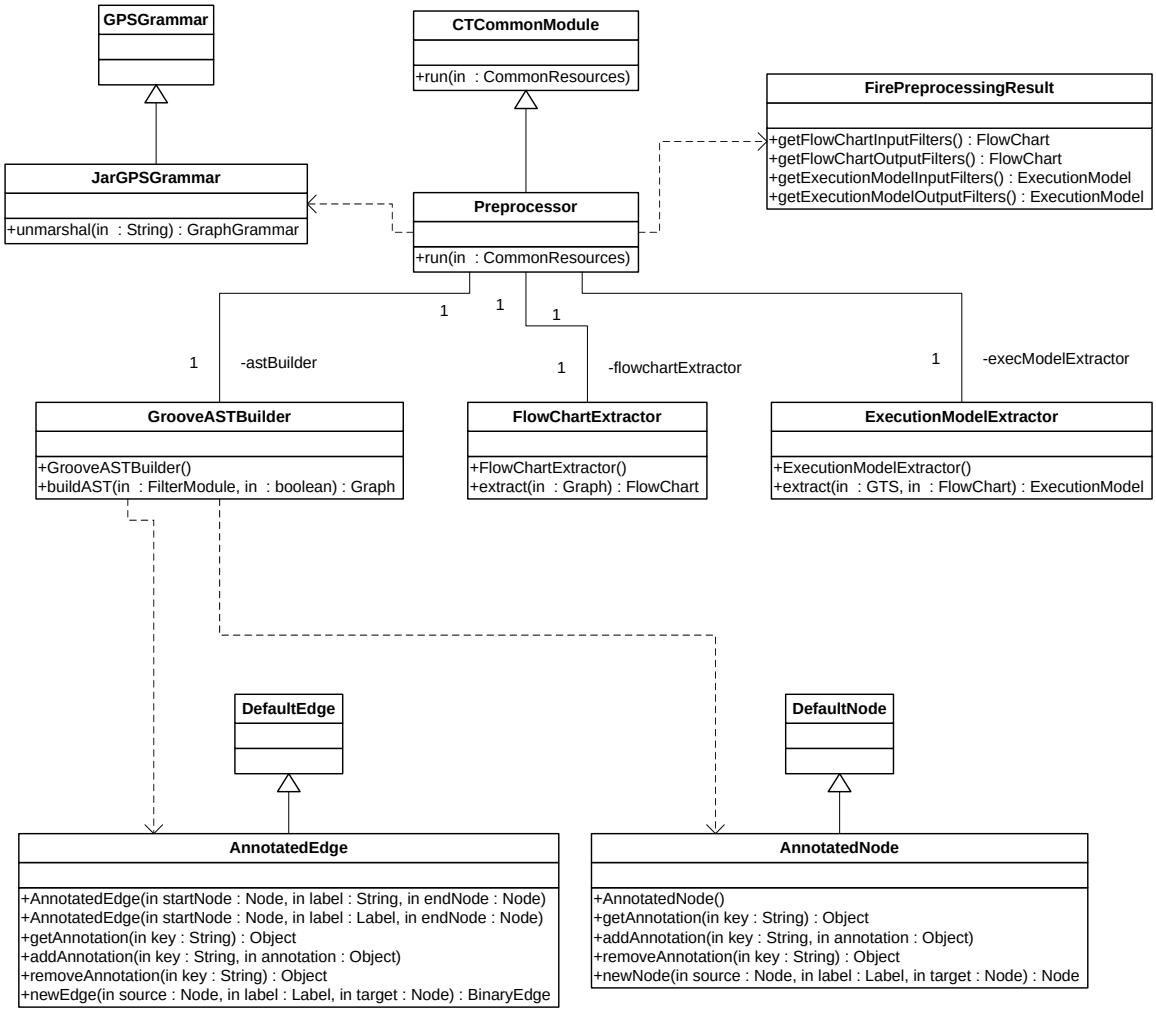


Figure 6.3: Class structure of the FIRE Preprocessor.

- The flowcharts and execution models of different filter modules can be easily and efficiently combined into the flowchart and execution model of a filter set. How this can be done is explained in the next section.
- Few filter modules might be used in many different filter sets. So applying the process on each filter module facilitates the reuse of the results in each filter set the filter module is used in.
- As explained earlier, GROOVE is inefficient compared to the theoretical efficiency of the reasoning procedure. Applying GROOVE on a filter module instead of a filter set leads to less states and smaller graphs, so a faster operation.

Looking at these arguments one might argue about why not going further and do the processing on each filter instead of each filter module, or even further on each filter element instead of each filter module or filter. This is not done because of the following reasons:

- Filters are not reused individually in different filter modules. Therefore, the filter reasoning results for a filter are only used in one filter module. So, filters do not have the reuse advantage that filter modules have.

- Dividing the processing into smaller pieces than a filter set means that the flow structure between the pieces needs to be generated when combining the pieces. The flow structure between filter modules is straightforward. It just consists of a sequential flow: when a message leaves one filter module it enters the next. Note that certain filter modules might be conditional. The flow structure between filters can be more complex. Although not implemented in Compose⋆ yet, there are different composition operators possible between filters, each leading to a different flow structure. Using filters as the processing pieces means that we cannot define the rules for the flow structure between the filters in the GROOVE production systems, but have to code them in the combining logic. This reduces the advantage that the GROOVE production system is easy to modify.

So, to reduce the GROOVE overhead, while maintaining the advantages of using GROOVE, it is best to do the processing on each filter module.

6.4 Combining the Information of Different Filter Modules in a Filter Set

To obtain the flowchart and execution model of a filter set, the flowcharts and execution models of the individual filter modules in the filter set need to be combined. This section explains how this is done.

6.4.1 Combining Different Flowcharts

The flowcharts of each individual filter module are obtained from the repository. Figure 6.4 shows a schematic representation of a flowchart of a filter module.

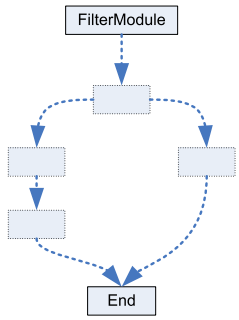


Figure 6.4: The flowchart of a filter module.

This figure shows that the flowchart has one entrance node, the `FilterModule` node. The flowchart also has one exit node, the `End` node. This node indicates the end of the flow in the current filter module and the continuation to the next filter module. Note that a filter module has also an `Exit` node and a `Return` node to mark the end of the

flow in the filter module. However, if these nodes are reached, flow does not continue to the next filter module. Therefore, they are not shown in the figure.

Step 1: Adding Conditional Superimposition Certain filter modules might be conditionally superimposed on a concern. The first step in the combination process is extending the flowchart of a filter module with the filter module condition. This is only done for filter modules that are conditionally superimposed. Figure 6.5 shows how the flowchart of the filter module is extended with the filter module condition.

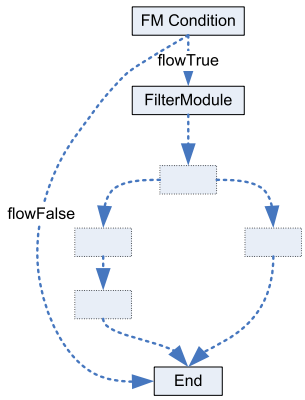


Figure 6.5: First step: adding the filter module condition.

The figure shows that the `FilterModuleCondition` node becomes the entrance node of the extended flowchart. The `FilterModuleCondition` node has two outgoing edges. The `flowTrue` edge is targeted at the `FilterModule` node: if the condition is `true`, the filter module is executed. The `flowFalse` edge is targeted at the `End` node: if the condition is `false`, the filter module is not executed and flow continues to the next filter module.

Step 2: Combining the Extended Flowcharts After the flowcharts of the filter modules have been extended with the filter module conditions, they can be combined to form the flowchart of the filter set. Flow between filter modules is just sequential. So different filter modules can be combined by adding a flow edge between one filter module's end node and the next filter module's start node. Figure 6.6 shows an example of three filter module combined in a filter set. The entrance node of the first flowchart becomes the entrance node of the combined flowchart. The end node of the last flowchart becomes the end node of the combined flowchart. Note that the second filter module is not conditionally superimposed.

6.4.2 Combining Different Execution Models

Combining the different execution models is more complex than combining the different flowcharts. Therefore, we explain this process using an example. This example contains three filter modules. Targets are ignored in this example, to make it more easy to understand. Listing 6.1 shows the three filter modules.

```
1 filtermodule FM1 {
```

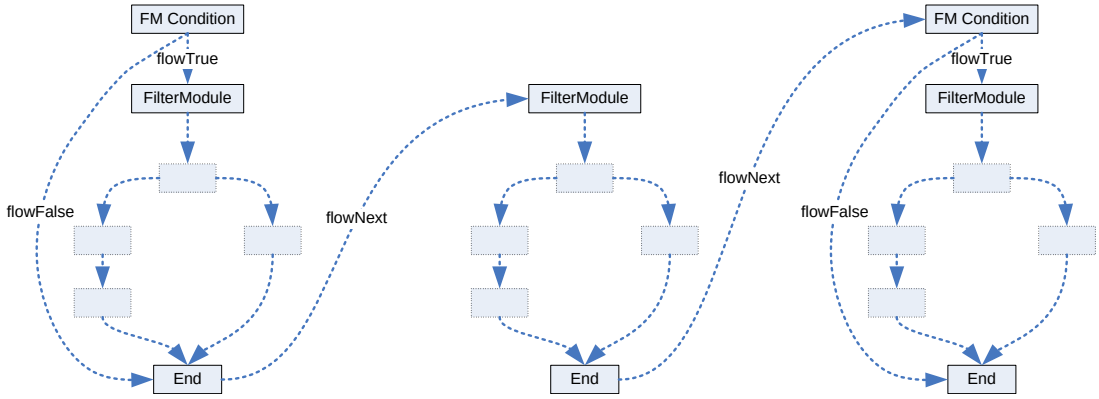


Figure 6.6: Combining two flowcharts

```
  inputfilters
    error : Error        = { True ~> [a] };
    subst : Substitution = { C => [b] c }
}

filtermodule FM2 {
  inputfilters
    disp1 : Dispatch = { [c] e };
    log   : Logging  = { [a] };
    time  : Timing   = { True ~> [a] }
}

filtermodule FM3 {
  inputfilters
    disp2 : Dispatch = { [*] }
}
```

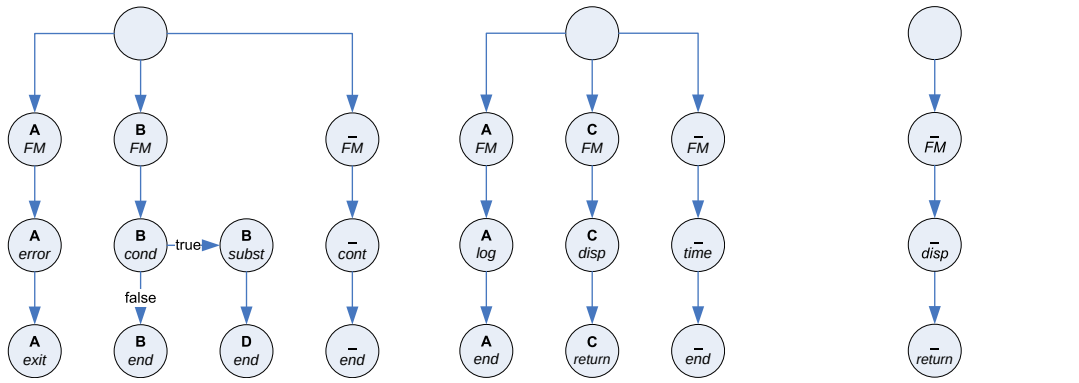
Listing 6.1: The filter modules

The first filter module contains one `Error` filter, which gives an error for all messages `a`. The second filter is a substitution filter that substitutes `b` to `c`, if condition `C` is true. The second filter module is conditionally superimposed. It dispatches all messages `c` to `e`, logs all messages `a` and times all messages different from `a`. The third filter module dispatches all messages.

Figure 6.7 shows schematic representations of the execution models. Only important states are shown. Each state is labeled with the message in that state and with the type of the state. The following procedure combines the execution models of each filter module to the execution model of the filter set.

Step 1: Adding Filter Module Conditions The first step in the combination process is adding the filter module conditions. In our example, only the second filter module is conditional.

Figure 6.8 shows how the filter module condition is added to the execution model of the second filter module. This figure shows that before each `FilterModule` state a new `FilterModuleCondition` state (abbreviated as 'cond') is added. The `true` transition from the `FilterModuleCondition` state is targeted at the `FilterModule` state. The `false`



(a) First execution model (b) Second execution model (c) Third execution model

Figure 6.7: The execution models corresponding to the separate filter modules.

transition from the `FilterModuleCondition` state is targeted at the `End` state. If there is no `End` state for that specific message, the `End` state is added. This is the case for message `c` in the example.

Step 2: Creating the Entrance Set To combine the execution models of different filter modules in a filter set, we first need to get the complete set of entrance messages of the combined execution model. This set is created by combining the distinguishable targets and the distinguishable selectors, as explained in Section 5.3:

$$messages = (T_{dist} \bigcup \{\_\}) \times (S_{dist} \bigcup \{\_\})$$

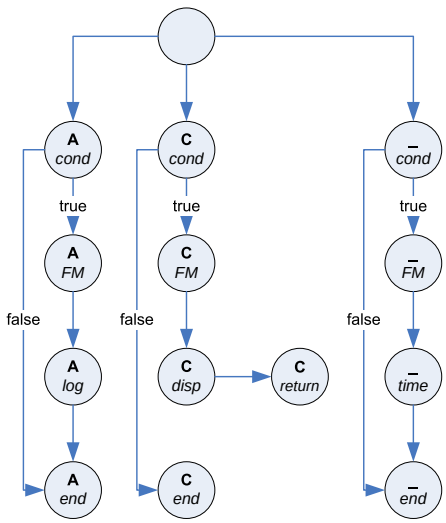


Figure 6.8: Adding filter module conditions to the execution model.

Where the set of all distinguishable targets is the union of the set of distinguishable targets of each filter module:

$$T_{dist} = \bigcup_{fm \in fs} T_{dist}(fm)$$

The same for the distinguishable selectors:

$$S_{dist} = \bigcup_{fm \in fs} S_{dist}(fm)$$

In the example we ignore the targets. So, the set of distinguishable selectors for each filter module in the example is shown in Table 6.1.

Filter module	distinguishable set
FM1	$\{a, b\}$
FM2	$\{a, c\}$
FM3	$\{\}$
Entrance set	$\{a, b, c, \_\}$

Table 6.1: The distinguishable selectors of each filter module

This set of messages is the entrance set of the combined execution model. To create this combined execution model we start with the execution model of the first filter module and apply Step 3.

Step 3: Expanding the Execution Model with the Entrance Set The entrance set is used to expand the execution model of the filter module with additional execution traces. The following procedure is applied:

- For messages that already have a corresponding entrance state, this state is maintained as the entrance state for that message.
- For new messages that do not have a corresponding entrance state, the closest matching undistinguishable entrance state is taken. For example for the new message `A.p`, we take the entrance state with message `A._` or `_.p`, if available (note that when both are available, `A.p` was also available). If they are not available, the entrance state with message `_._` is taken (the completely undistinguishable entrance state).

  From this closest matching undistinguishable entrance state, a clone of the complete trace is made. First, the entrance state is cloned. The message property of the clone is set to the new message. All outgoing transitions are also cloned. If they are targeted at a state that has the same undistinguishable message, this state is also cloned, etc. This algorithm stops when a state is reached that has no outgoing transitions or when a state is reach of which the message property is not the undistinguishable message.
- Entrance states for which there is no message in the entrance set are removed. All traces from this entrance state are also removed.

So, for the new entrance message the execution traces of a (partially) undistinguishable message are cloned.

Figure 6.9 shows the extended execution model of the first filter module. For the

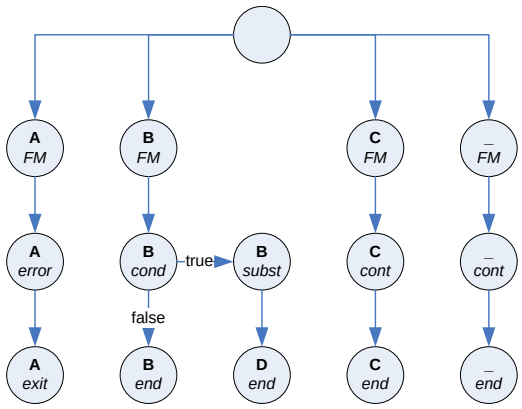


Figure 6.9: Extended execution model of the first filter module.

first execution model of the example only the message `c` needs to be added. A clone is made of the execution paths of message `_`.

One might argue that expanding the execution model is not needed. Instead of cloning the undistinguishable trace, we can attach multiple messages to that trace, in this way indicating that those messages also go through that trace. In this way, no new states need to be created. Why this is not done is discussed in Section 6.7.1.

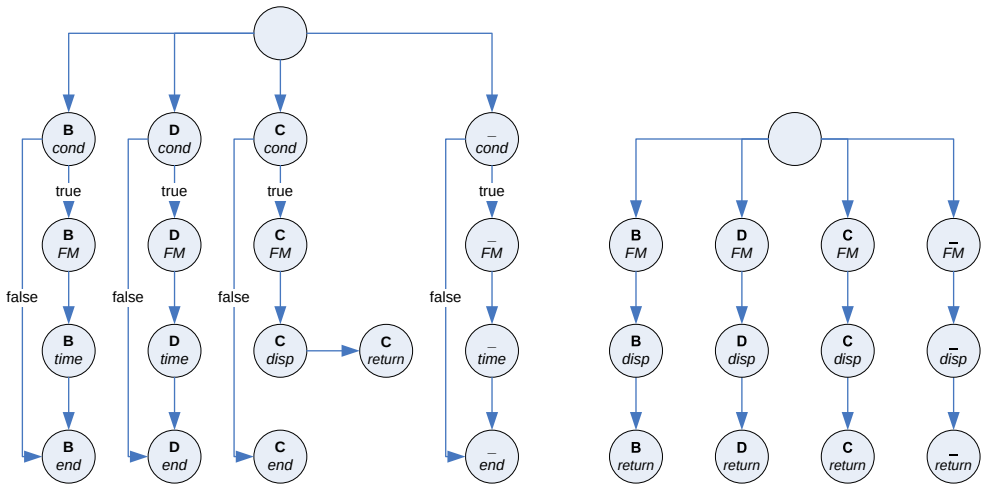
Step 4: Creating the exit set After the extended execution model has been generated, the exit set of messages is constructed. This is the set of all messages for which there is an `End` state. These messages continue to the next filter module. The messages that reach an `Exit` state or a `Return` state are not used, because they do not continue to the next filter module. The exit set becomes the entrance set for the next filter module.

For the first filter module in the example, the exit set is $\{b, c, d, \_\}$.

Step 5: Repeat step 3 and 4 Steps 3 and 4 are repeated for the next filter modules. Figure 6.10 shows the results of this expansion. In the second filter module, the `a` trace is removed, because message `a` does not enter the filter module. A clone of the `_` trace is made for message `b`. The exit set of the second filter module is $b, c, d, \_$.

This exit set is used to expand the third filter module. The figure shows that the `_` trace is cloned for message `b`, `c` and `d`. There is no need to calculate the exit set, because there is no next filter module. The next step is combining the expanded execution models.

Step 6: Combining the Expanded Execution Models Figure 6.11 shows the combined execution model. This combined model is created by adding a transition from each end state in one execution model to the corresponding entrance state in the next execution model.



(a) Second execution model expanded (b) Third execution model expanded

Figure 6.10: The execution models corresponding to the separate filter modules.

6.5 The Filter Reasoning Model

This section describes the interface of the FIRE Model. The FIRE Model is contained in the package `Composestar.Core.FIRE2.Model`.

6.5.1 The FIRE Model Structure

The FIRE Model component provides access to the information generated during FIRE's preprocessing phase. Remember that we do not have access to the original GROOVE models, but to an easier to use representation. The relation between this representation and the original GROOVE models is bijective, so no information is lost. Figure 6.12 shows the classes in the FIRE model component.

FireModel

The `FireModel` class is the main class to access the filter reasoning information. It has the following tasks:

- To provide access to the flowchart.
- To provide access to the different execution models: the complete execution model of the concern or the execution model for a specific entrance message.
- To combine the flowcharts and execution models of the different filter modules in the filter set (explained in the previous section).
- To evaluate signature matching, when signature information has been generated. Because the signature generation engine uses the filter reasoning engine, signature information was not available yet to the filter reasoning preprocessor. Therefore, the preprocessor could not evaluate signature matching parts. When signature information has been generated, `FireModel` takes care of evaluating the signature matching parts.

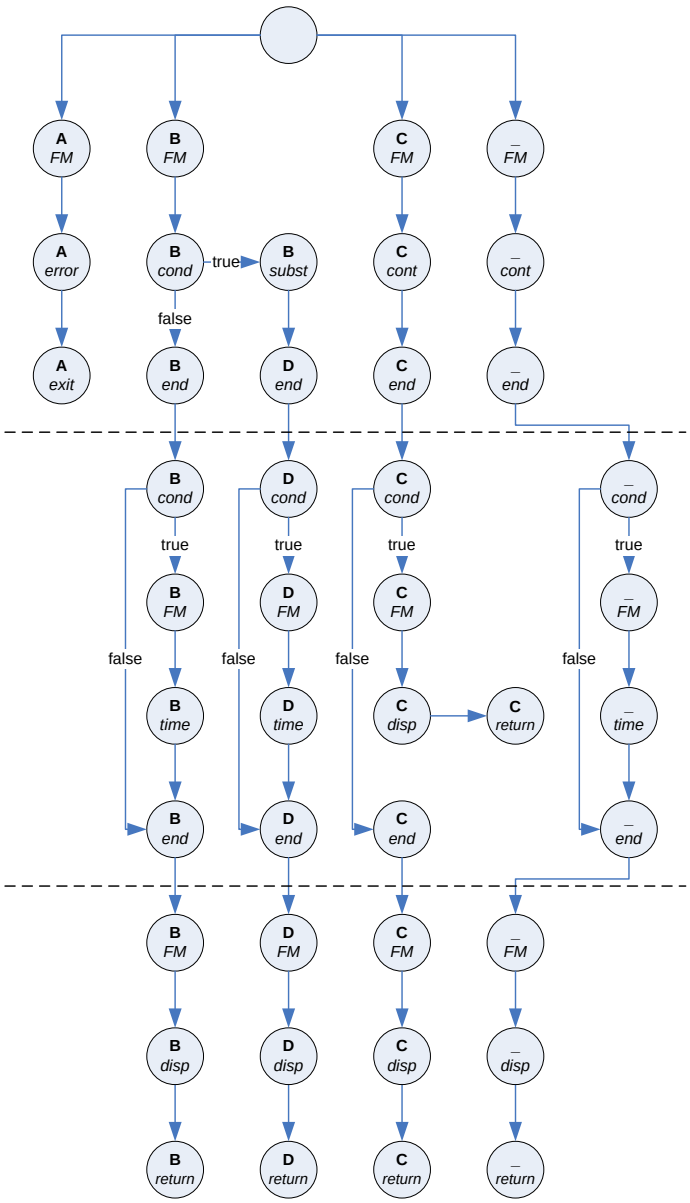


Figure 6.11: Combined execution model.

FlowChart, ExecutionModel

The class `FlowChart` represents a flowchart. The class `ExecutionModel` represent an execution model. Both classes form the main class of a more detailed class structure. These class structures are explained next.

6.5.2 The Flowchart Structure

Figure 6.13 shows `FlowChart` structure. It contains the following classes:

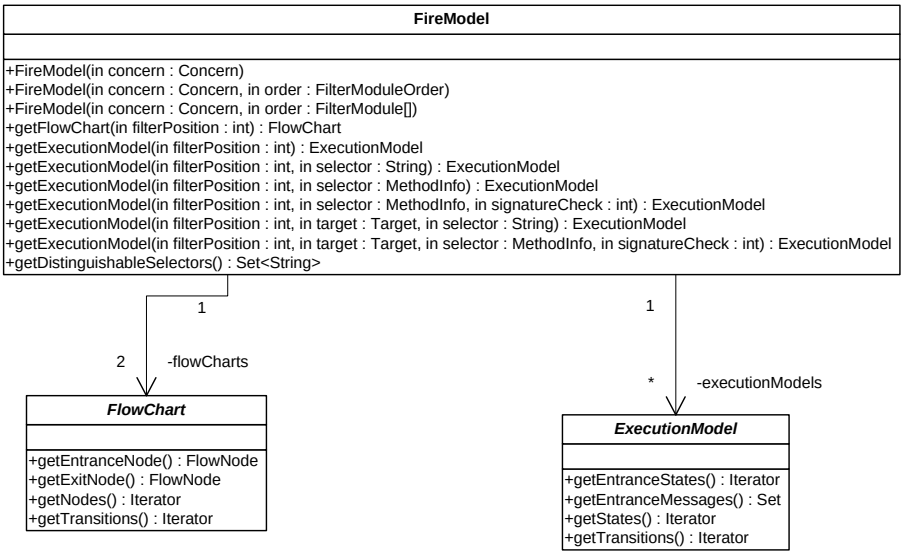


Figure 6.12: The `FireModel` component.

FlowChart

The class `FlowChart` represents a flowchart. A flowchart contains a number of flow nodes and flow edges. It also has one entrance node and one exit node. This exit node is the `End` node. After this node, flow continues to the next filter module. Note that the `End` node might not be reachable.

FlowNode

The class `FlowNode` represents a flow node. A `FlowNode` has the following properties:

- A number of outgoing `FlowTransitions`.
- A number of labels. `FlowNode` can have multiple labels. These labels represent the hierarchy of groups to which the flow node belongs. For example, a `Dispatch` action `FlowNode` has the labels `FlowNode`, `ProcedureNode`, `ActionNode` and `DispatchActionNode`. This hierarchical naming makes it possible to select a certain group of nodes, all `ActionNode` nodes for example.
- The corresponding `RepositoryEntity`

FlowTransition

The class `FlowTransition` represents a flow transition. It contains the following properties:

- The start node and the end node of the transition.
- The type of the transition: `flowNext`, `flowTrue` or `flowFalse`.

RepositoryEntity

The class `RepositoryEntity` is the standard repository entity class from the package `Composestar.Core.RepositoryImplementation`.

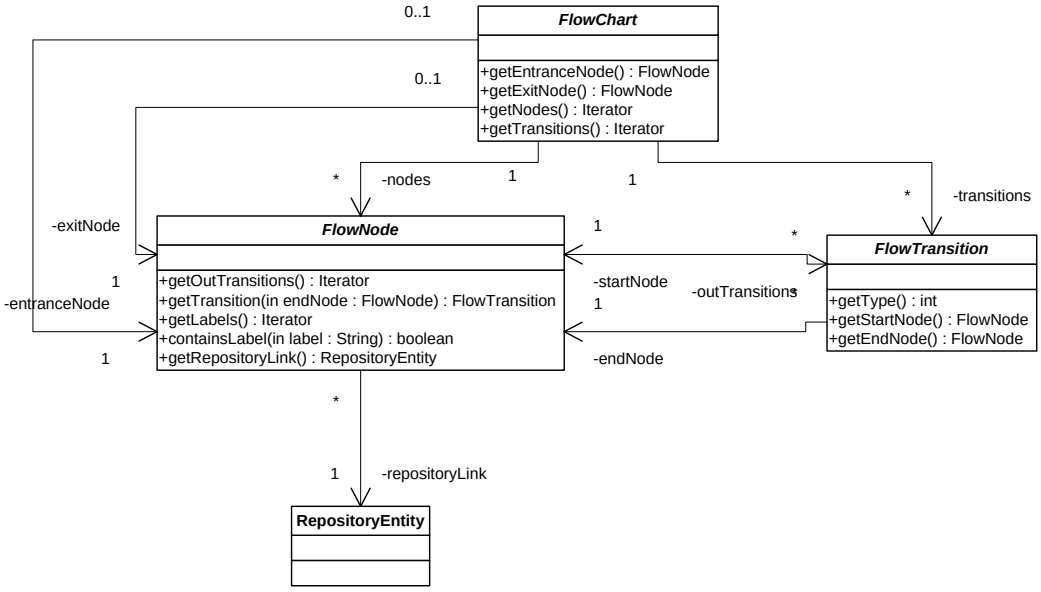


Figure 6.13: The `Flowchart` structure.

6.5.3 The Execution Model Structure

Figure 6.14 shows the `ExecutionModel` class structure.

ExecutionModel

The class `ExecutionModel` represents an execution model. It contains the following properties:

- All states in the execution model.
- All transition in the execution model.
- The entrance states of the execution model.
- The entrance messages. These messages are the messages corresponding to the entrance states.

ExecutionState

The class `ExecutionState` represents an execution state. It contains the following properties:

- The type of the state: `entrance`, `exit` or `normal`.
- The corresponding `FlowNode`.
- The message and the substitution message.
- The outgoing transitions from the state.

ExecutionTransition

The class `ExecutionTransition` represents an execution transition. It contains the following properties:

- The start state and the end state of the transition.
- The corresponding `FlowTransition`.

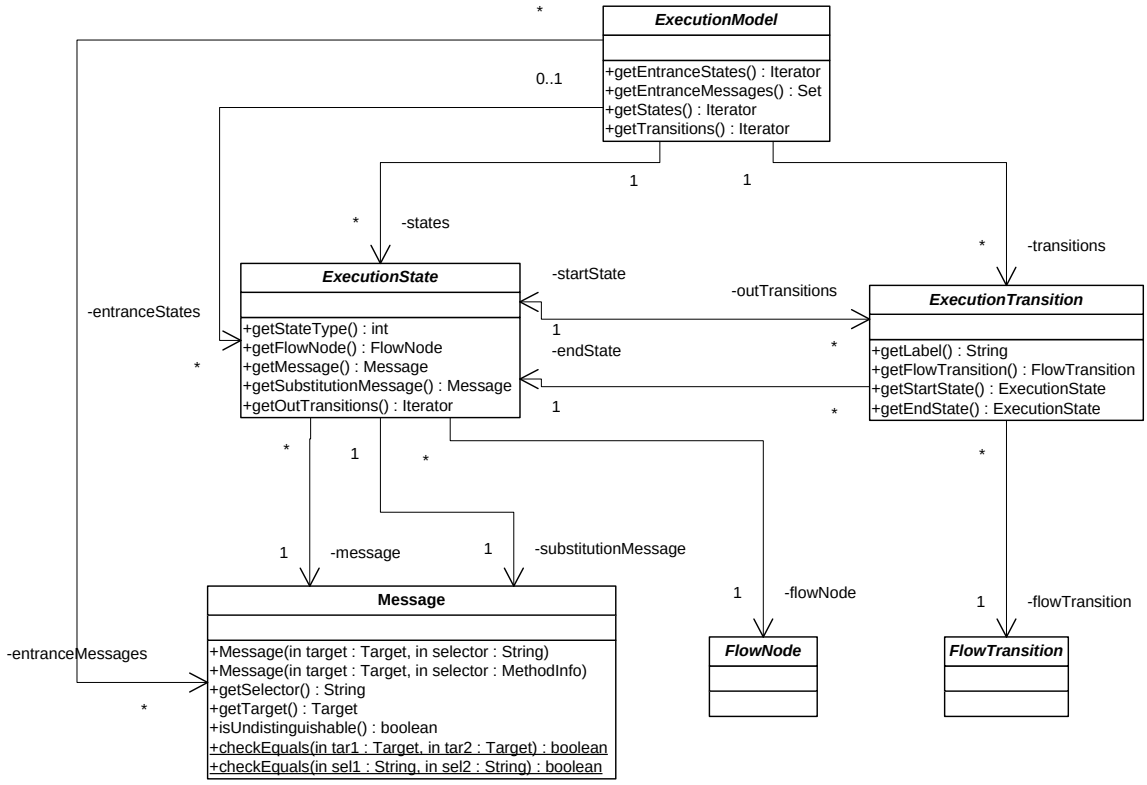


Figure 6.14: The `ExecutionModel` structure.

- The label of the execution transition. This is equal to the name of the GROOVE rule applied in the simulation.

Message

The class `Message` represents a message. It contains a target and a selector.

FlowNode, FlowTransition

These classes represent flow nodes and flow transitions. They have been explained during the explanation of the flow model structure.

6.6 Filter Reasoning Engine Tools

On top of the FIRE Model are some tools that make using the models easier. These tools are:

- *Iterators*: Several iterators are provided to iterate over the nodes in a flowchart or states in an execution model in a specific order.
- *Regular Expression Checker*: A regular expression checker is provided to match regular expressions on all possible executions of the filter set.
- *Query Engine*: Two query engines are provided to obtain all states from an execution model that have a certain property. The *basic query engine* only finds states

in which a certain *predicate* is true. The *CTL query engine* finds states in which a *computational temporal logic* formula is true.

- *Viewer*: A viewer is supplied to visualize a flowchart or execution model. This can be used as an aid during the development of a module that uses filter reasoning.

6.6.1 Iterators

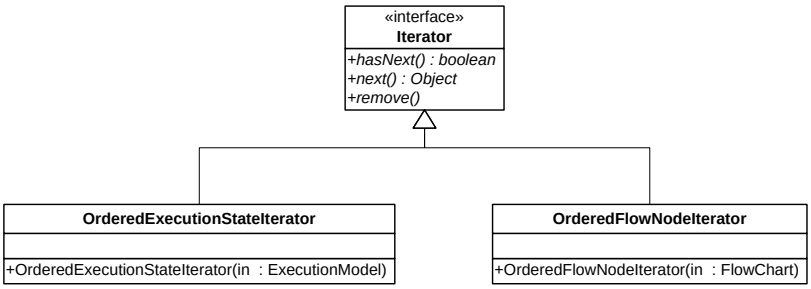


Figure 6.15: The `Iterator` components.

Figure 6.15 shows the iterator tools. These tools are in the package `Composestar.Core.FIRE2.Tools.Iterator`. An iterator is supplied for both flowcharts as for execution models. This iterator iterates in such a way that a node or state is not visited until all predecessor nodes or states are visited. Because there are no cycles in a flowchart or an execution model, this is possible.

Iterator
This is the standard Java `Iterator` interface.

OrderedExecutionStateIterator
An iterator to iterate over the states in an execution model.

OrderedFlowNodeIterator
An iterator to iterate over the nodes in a flowchart.

If ordered iteration is not needed, the default iterators from the class `FlowChart` or `ExecutionModel` should be used. They are more efficient.

6.6.2 Regular Expression Checker

This tool provides functionality to check whether there is an execution path in a given execution model that matches a certain regular expression. To do this, it provides the ability to transparently add labels to the transitions in the execution model (other modules cannot see these labels). The regular expression is matched on these added labels. This is for example useful for behavioral reasoning. For more information about the implementation of this module and about using this model, see Chapter 9.

6.6.3 Query Engine

The *query engine* can be used to find certain states in the execution model. Two query engines are supplied: a *basic query engine* to find states that match a *predicate* and a *ctl*

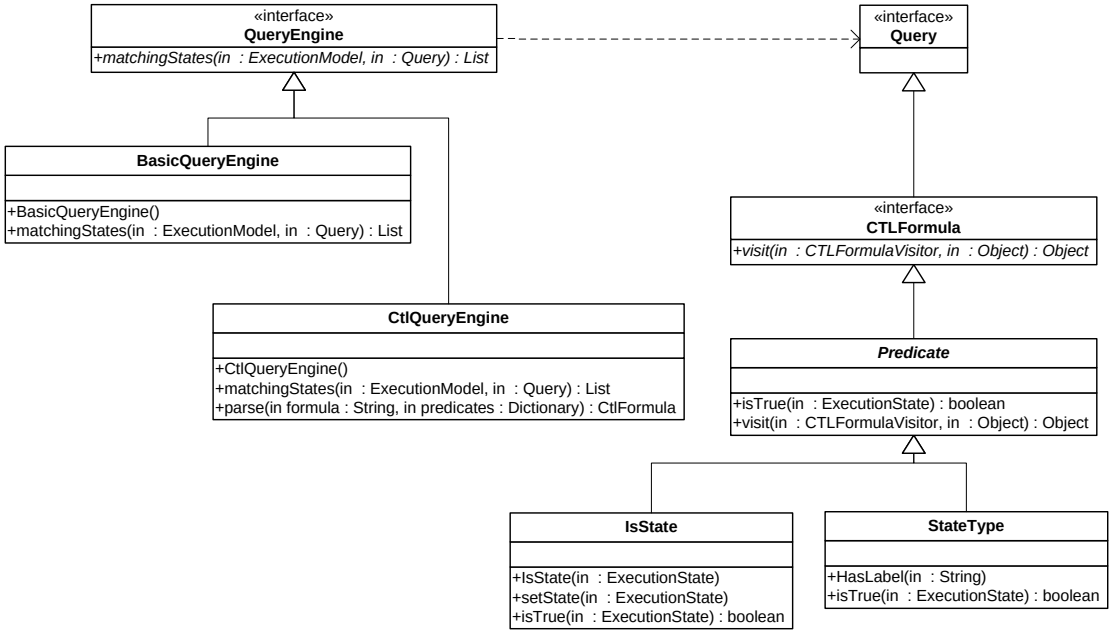


Figure 6.16: The `QueryEngine` components.

query engine to find states that match a *computational tree logic expression*. Figure 6.16 shows the class structure of the Query Engine. It is contained in the package `Composestar.Core.FIRE2.Tools.QueryEngine`.

QueryEngine
: `QueryEngine` is the interface that all query engines should implement.

Query
: `Query` represents a query to find states. Specific implementations of this interface are given for the basic query engine and the ctl query engine.

BasicQueryEngine
: The basic query engine. This query engine finds states that match a given predicate.

CTLQueryEngine
: The ctl query engine. The internal details of this query engine is explained later in this section.

CTLFormula
: `CTLFormula` represents a ctl formula. This is a `Query` for the `CTLQueryEngine`.

Predicate
: Implementations of the `Predicate` interface represent a predicate to match a state. Predicates can be used for both the basic query engine as for the ctl query engine.

IsState
: `IsState` is an implementation of the `Predicate` interface to match a specific state.

StateType
: `StateType` is an implementation of the `Predicate` interface to match states with a specific type. A state matches if its corresponding flow node has the given type as label. This predicate can for example be used to find all `DispatchAction` states.

6.6.3.1 Internal Structure of the CTL Query Engine

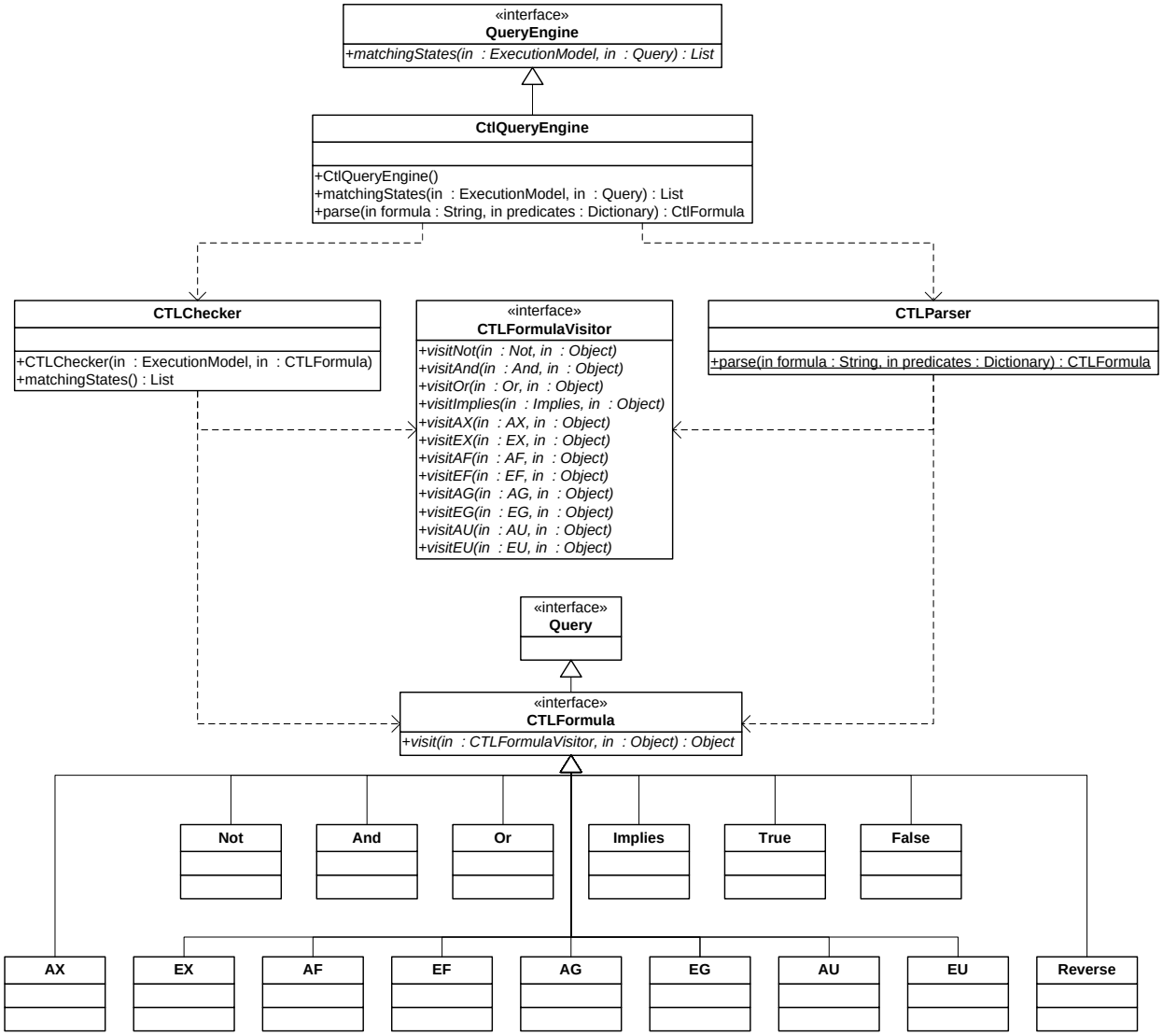


Figure 6.17: The internal structure of the `CTLQueryEngine`.

Figure 6.17 shows the internal structure of the ctl query engine.

QueryEngine, Query
: These are the general `QueryEngine` and `Query` interfaces.

CTLQueryEngine
: The `CTLQueryEngine` to find states that match a `CTLFormula`.

CTLParser
: The `CTLQueryEngine` uses the `CTLParser` class to parse a `String` representation of a ctl formula.

CTLChecker
: The `CTLChecker` class implements the ctl checking algorithm. It is used by the

`CTLQueryEngine` class to find the matching states.

CTLFormula + implementations

The `CTLFormula` class represents a ctl formula. It has implementations for each type of operator. These implementations for each type of operator contains methods to obtain the operands. These methods are not shown.

6.6.4 Viewer

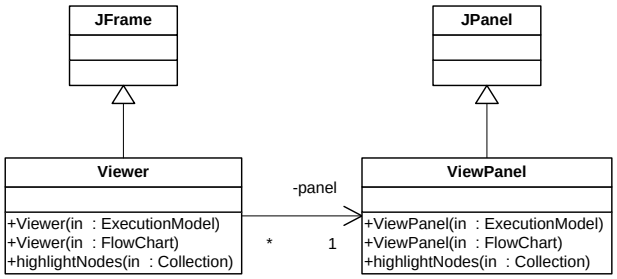


Figure 6.18: The `Viewer` components.

Figure 6.18 shows the class structure of the Viewer tool. The Viewer tool is in the package `Composestar.Core.FIRE2.Tools.Viewer`. The Viewer tool provides GUI classes to visualize a flowchart or an execution model. This can be helpful during implementation and debugging of a module that uses FIRE, in order to get an idea of what a certain flowchart or execution model looks like.

Viewer

Creates a new frame to show the given execution model or flowchart

ViewPanel

Creates a panel to show the given execution model or flowchart

JFrame, JPanel

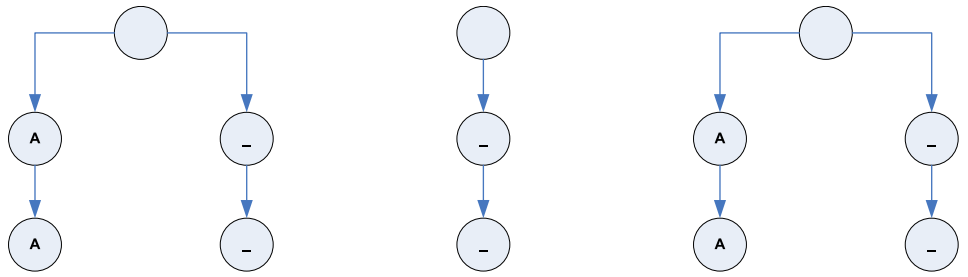
These are classes from the Java Swing package.

6.7 Implementation issues

This section discusses a number of implementation issues. It first discusses whether execution model expansion is really needed in the combination algorithm. Then it discusses GROOVE's inefficiency.

6.7.1 Is Execution Model Expansion Needed?

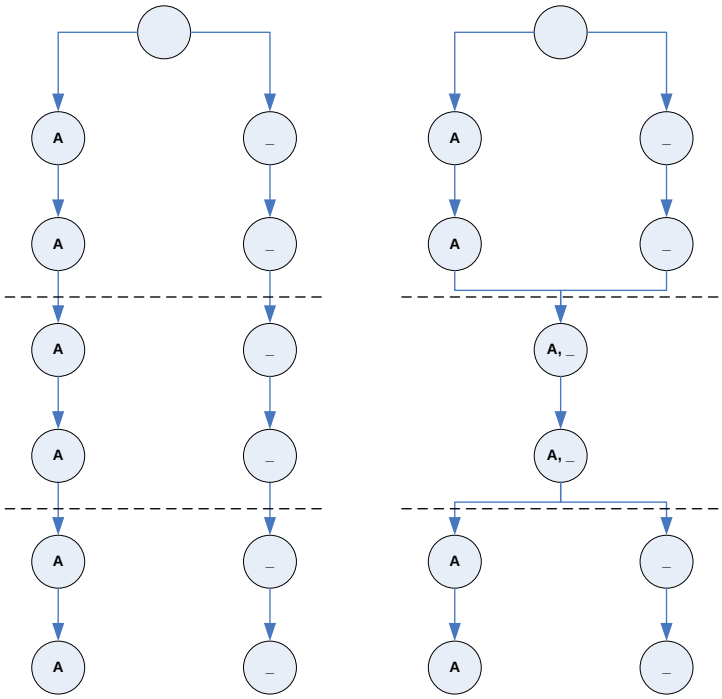
The combination algorithm expands the execution models of the filter modules using the entrance set. The undistinguishable trace is cloned for entrance messages that do not have a corresponding trace. The question raises why this is needed; the messages for which the undistinguishable trace is cloned all behave the same in the execution model. These messages can be attached as a set to the states in the undistinguishable trace. In this way, the states do not have to be cloned. We will illustrate this with an example.



(a) First execution model (b) Second execution model (c) Third execution model

Figure 6.19: The execution models corresponding to the separate filter modules.

Figure 6.19 shows the execution models of three filter modules that are combined in a filter set. Figure 6.20 shows the combined execution models, both with expansion as without expansion. In the execution model without expansion, the entrance message `A` is added in the set of the undistinguishable trace in the second execution model.



(a) With expansion (b) Without expansion

Figure 6.20: The combined execution model with and without expansion.

The problem is that, although message `A` behaves the same as the undistinguishable message in the second filter module, it does not behave the same in the entire filter set. Therefore, the execution model without expansion is not a true execution model anymore. This means that if a state is reachable from another state in the structure,

this might not be the case in practice. For example, the graph structure in Figure 6.20b shows that the last `A` state (third execution model) is reachable from the first `_` state (first execution model). But this can never happen in practice. Actually, the structure shows four different execution paths, while in practice there are only two. This leads to problems with certain application of filter reasoning that depend on the fact that the paths through the execution model represent all possible executions.

6.7.2 GROOVE's inefficiency

We used GROOVE to implement the transformation and simulation rules. The problem with GROOVE is that it uses graph matching to apply the transformation rules and isomorphism checks to check whether two states are equal. These techniques are in general less efficient than the theoretical complexity of our problem, as was explained in Section 5.4. For example, rule matching is in general NP-complete in the size of the rule. This could have been more efficient if the rule matching algorithm was specific for our application domain. For example, for our simulation rule system the rule matching algorithm only has to check the target of the frame's program counter edge to determine the rule to apply. In this way, the algorithm can approach its theoretical complexity. Rensink describes that allowing these domain specific rule matching algorithms are possible future improvements of GROOVE. [44]

The time complexity of checking whether two graphs are isomorphic is not known yet. The general believe is that it lies strictly between P and NP, but no proof has been found yet. GROOVE uses some approximation techniques to make it more efficient in certain cases, but this does not come close to the theoretical complexity. [44]

6.7.2.1 GROOVE Time Measurements

We did some time measurements to find out how well GROOVE is performing on our problem. We measured the time GROOVE took to do the reasoning for filter sets varying in size from one filter to twenty filters. Each filter consisted of two filter elements. All filters used the same set of targets and selectors so that the number of messages was the same for every filter set. This ensures that the time complexity is linear in the size of the filter set. Each filter set is tested five times. Table 6.2 shows the average times for both the transformation from AST to flowchart as to do the simulation.

In Figure 6.21 and Figure 6.22 these measurements are plotted on a linear and logarithmic scale. From the linear scale it can be seen clearly that the reasoning time is not growing linearly with the size of the filter set. From the logarithmic scale it becomes clear that the reasoning time is not growing exponentially (it is not a straight line).

From these results we can conclude that the GROOVE implementation is less efficient than the theoretical time complexity of the algorithm. It looks that it is multiple degree polynomial, instead of linear. This might degrade performance with larger filter sets. Note that a reasonable size of a filter module is four filters. In this case, the time is still reasonable.

size	transformation (ms)	simulation (ms)
1	115	175
2	200	418
3	403	762
4	625	1437
5	1009	2362
6	1521	3696
7	2140	5806
8	2990	9371
9	4234	14149
10	6003	19956
11	8465	25787
12	11096	32669
13	13240	43722
14	15802	55266
15	20500	67918
16	24203	84837
17	27809	105634
18	33253	131146
19	38619	165068
20	46243	203200

Table 6.2: Time GROOVE took to do the transformation and simulation for varying sizes of filter sets

6.7.2.2 FIRE Model Combination Time Measurements

We also measured the time of the combination algorithm in the FIRE Model component. This time was measured using a filter module consisting of 4 filters. These filters are

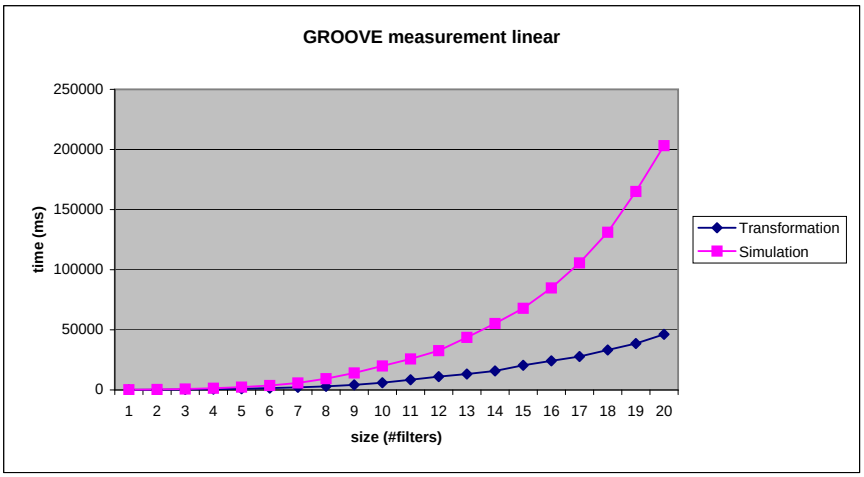


Figure 6.21: Time GROOVE took to do the transformation and simulation for varying sizes of filter sets on a linear scale.

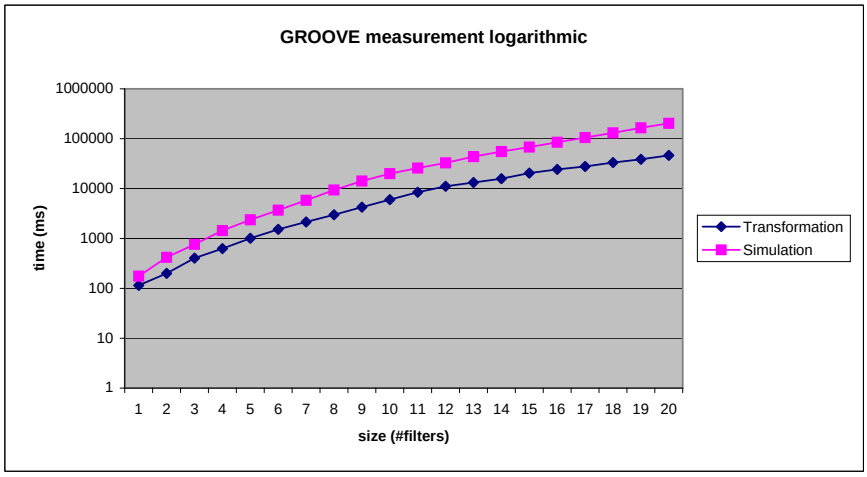


Figure 6.22: Time GROOVE took to do the transformation and simulation for varying sizes of filter sets on a logarithmic scale.

the same as the filters in the GROOVE time measurement. We measured the time for filter sets varying in size from 1 to 30 of these filter modules. Each size is tested 5 times and the time is averaged. During each of these tests, the combination was done 20 times and the total time of this 20 applications is measured, to get better significance. The time of this 20 applications is again averaged. Table 6.3 shows the time the combination algorithm took for each size of filter set.

size	combining (ms)	size	combining (ms)
1	2,65	16	18,25
2	1,05	17	19,5
3	1,85	18	17,95
4	2,35	19	21,2
5	3,25	20	25,15
6	4,05	21	30,15
7	4,5	22	28,1
8	10,15	23	34,05
9	6,2	24	35,45
10	7,65	25	41,7
11	8,9	26	37,95
12	9,5	27	49,65
13	11,25	28	41,85
14	12,95	29	52
15	17,8	30	64,05

Table 6.3: Time the FIRE Model took to combine the results of different filter modules in a filter set

Figure 6.23 shows these numbers plotted on a linear scale. Note that using this combination algorithm significantly improves the efficiency of the implementation. The

combination of 30 filter modules, equaling 120 filters, took only 64ms.

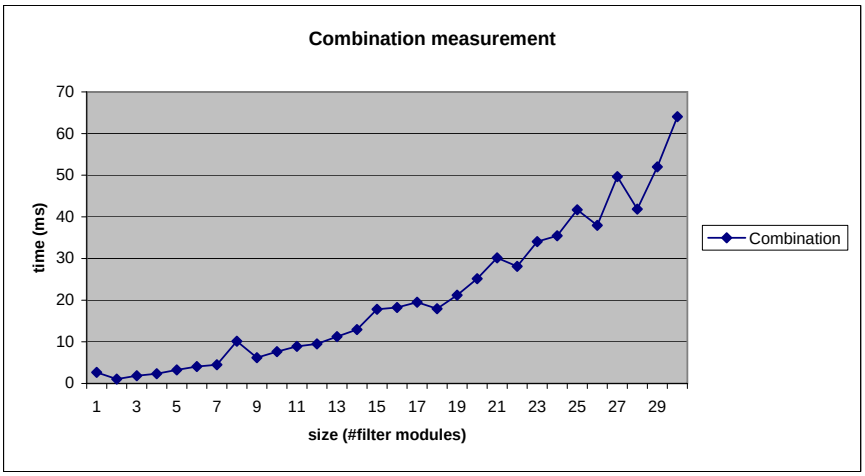


Figure 6.23: Time the FIRE Model took to combine the results of different filter modules in a filter set.

Part III

Message Flow Analysis Applied

Chapter 7

Consistency Reasoning

"Consistency is the last refuge of the unimaginative."
Oscar Wilde
Irish dramatist, novelist, & poet (1854 - 1900)

Filter elements can be composed into filters, filters into filter modules and filter modules into filter sets. The behavior of such a composition might cause certain components within the composition to be unreachable. For example, a certain filter might become unreachable if a previous `Dispatch` filter always accepts. Listing 7.1 shows an example of such a case. In this listing the first filter is an `Error` filter that gives an error on all messages except messages with selector `a`. The second filter is a `Dispatch` filter that dispatches messages with selector `a` to `inner`. The third filter is a `Dispatch` filter that dispatches messages with selector `b` to `inner`. Because the first filter only lets messages with selector `a` through and the second filter dispatches all messages with selector `a`, the third filter is never reached.

```
error : Error    = { True => [*.a] };
disp1 : Dispatch = { True => [*.a] inner.a };
disp2 : Dispatch = { True => [*.b] inner.b }
```

Listing 7.1: Consistency conflicts example

Unreachable components might indicate programming errors and therefore should be identified. This is the task of consistency reasoning. Consistency reasoning should identify the unreachable components in a filter set and bind conclusions to these unreachable components in the form of conflicts. It should also give information about the causes that leaded to the unreachability of these components. Examples of conflicts detected during consistency reasoning are unreachable filters, redundant filters (filters that do nothing but only continue), filter elements that always accept and matching parts that never match.

This chapter explores consistency reasoning. The first section explains the theory and concepts behind consistency reasoning; how are unreachable components identified, what conflicts arise from these unreachable components, and how do these conflicts relate to each other. The second section describes how consistency reasoning is implemented in the Compose⋆ project.

7.1 Theory and Concepts

This section explains the theory and concepts behind consistency reasoning. Consistency reasoning tries to find conflicts by identifying unreachable components in a filter set. This section first describes how unreachable components can be identified by using filter reasoning. After identifying the unreachable components, the next step is drawing conclusions to these unreachable components in the form of conflicts. The second part of this section explains which conflicts are indicated by which unreachable components. The second part also describes a cause and effect relationship between these conflicts. After explaining what consistency reasoning is and how it can be done, we also want to know the costs of using the algorithm. The last part of this section addresses this by looking at the computational complexity of the consistency reasoning algorithm.

7.1.1 Identifying Unreachable Components

The first step in consistency reasoning is identifying the unreachable components in the filter set. We will use filter reasoning to achieve this.

Filter reasoning provides us with a flowchart. The nodes in this flowchart correspond to the different components in the filter set. So, identifying unreachable components in the filter set is equivalent to identifying the unreachable nodes in the flowchart. Using the flowchart has the advantage that we can also examine the reachability of the flow edges. This can give us insight into the causes of the unreachability of the flow nodes. For example, the cause of an unreachable filter might be a matching part in a previous `Dispatch` filter that always matches (the `flowNoMatch` edge is never reached).

To find unreachable nodes and edges in the flowchart, the full execution model of the filter set can be used. The full execution model is the execution model corresponding to the simulation of the filter set with all possible messages, as was explained in Section 5.3. Each state in the execution model corresponds to a node in the flowchart. Also each transition in the execution model corresponds to an edge in the flowchart. This can be seen in Figure 7.1.

As was explained in Chapter 5, the execution model is a simulation of the filter set execution. A state is in the execution model if and only if there is an execution that can reach that state. Also, a transition between two states is in the execution model if and only if there is an execution that can do this transition. Because states correspond to nodes in the flowchart and transitions correspond to edges in the flowchart, a node is reachable if and only if there is a corresponding state in the execution model. Also, an edge is reachable if and only if there is a corresponding transition in the execution model.

So, all reachable components in the flowchart can now be found by iterating over all states and transitions in the execution model and mark the corresponding nodes and edges in the flowchart as reachable. Nodes and edges unmarked after the iteration are unreachable.

If we apply this algorithm to the example in Figure 7.1, we see that flow node `E` and flow edges 4 and 7 are unreachable.

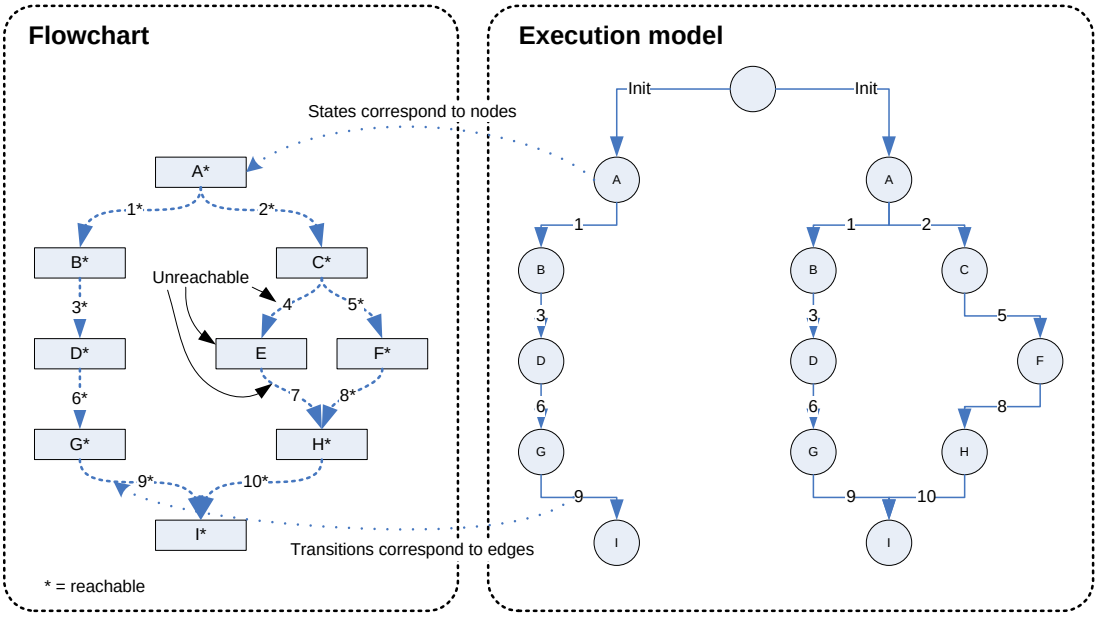


Figure 7.1: Finding unreachable nodes and edges by relating the states and transitions in the execution model back to the flowchart

7.1.2 From Unreachable Component to Conflict

After identifying the unreachable components, the next step is drawing conclusions from these unreachable components in the form of conflicts. This section looks systematically at each type of flow node and flow edge and indicates what conflicts arise if they are unreachable. Most conflicts do not happen in isolation. They are caused by other conflicts. This cause and effect relationship between conflicts is also investigated.

7.1.2.1 How Unreachable Components Indicate Conflicts

The unreachability of certain nodes and edges indicate conflicts. For example, an unreachable `filter` node indicates an unreachable filter. An unreachable `flowMatch` edge indicates a matching part that never matches. Not all types of flow nodes and flow edges indicate conflicts, while other flow nodes and flow edges indicate multiple conflicts. Table 7.1 shows for each type of flow node and flow edge what kind of conflicts are indicated if they are unreachable. Some conflicts are also bound by some other constraint, for example the type of the condition operator. Conflicts are written between angle brackets, to make them stand out of explaining text and constraints. Each entry in the table is explained next.

flowNext edge

If the `flowNext` edge is unreachable, the start node of the edge is also unreachable. The unreachability of the start node is a conflict, not the unreachability of the `flowNext` edge.

flowFalse edge

The `flowFalse` edge from an `ConditionExpression` node is unreachable if the con-

Flow node/edge	Conflicts
`flowNext` edge	No conflict
`flowFalse` edge	No conflict
`flowTrue` edge	⟨*Condition expression is the constant false*⟩ ⟨*Filter elem. always rej.*⟩ (⟨*redundant filter element*⟩)
`flowMatch` edge	⟨*Matching part never matches*⟩ If all matching parts never match: ⟨*Matching pattern always rejects*⟩, if `enable` oper. ⟨*Matching pattern always accepts*⟩, if `disable` oper. ⟨*The filter element always rejects*⟩, if `enable` oper. ⟨*The filter element always accepts*⟩, if `disable` oper and cond. expr. = `true`
`flowNoMatch` edge	⟨*Matching part always matches*⟩. ⟨*Matching pattern always accepts*⟩, if `enable` oper. ⟨*Matching pattern always rejects*⟩, if `disable` oper. ⟨*The filter element always accepts*⟩, if `enable` oper. and cond. expr. = `true`. ⟨*The filter element always rejects*⟩, if `disable` oper.
`FilterSet` node	Cannot be unreachable
`FilterModule` node	⟨*Unreachable filter module*⟩
`FilterModuleCondition` node	Subordinate conflict
`Filter` node	⟨*Unreachable filter*⟩
`FilterElement` node	⟨*Unreachable filter element*⟩
`ConditionExpression` node	Subordinate conflict
`ConditionOperator` node	Subordinate conflict
`MatchingPattern` node	Subordinate conflict
`MatchingPart` node	⟨*Unreachable matching part*⟩
`SubstitutionPart` node	Subordinate conflict
`FilterAction` node	If rej. action and filter reachable, ⟨*filter always accepts*⟩ If acc. action and filter reachable, ⟨*filter always rejects*⟩ Only `Continue` actions reachable: ⟨*Redundant filter*⟩

Table 7.1: Conflicts indicated by unreachable nodes and edges.

dition expression is the constant `true`. This is a common situation, because not all filter element are conditional. Therefore, the unreachability of the `flowFalse` edge is no conflict.

Note that another cause of the unreachability of the `flowFalse` edge is the unreach-

ability of the `ConditionExpression` node. Also in this case, the unreachability of the `flowFalse` edge does not indicate a conflict.

flowTrue edge

The unreachability of the `flowTrue` edge is a conflict. The `flowTrue` edge is unreachable of the condition expression is the constant `false`. But this is not useful; if the condition expression is the constant `false`, the filter element always rejects. This means that the filter element is useless. Therefore, the unreachability of the `flowTrue` edge indicates the conflict 'the condition expression is the constant `false`'. This conflict also causes the conflicts 'filter element always rejects' and 'redundant filter element'.

Note that another cause of the unreachability of the `flowTrue` edge is the unreachability of the `ConditionExpression` node. In this case, the unreachability of the `flowTrue` edge is a subordinate conflict.

flowMatch edge

If the `flowMatch` edge from a `MatchingPart` node is unreachable, the matching part never matches.

If the condition operator is the `enable` condition operator and all matching parts never match, the matching pattern always rejects. This causes the filter element to always reject (redundant filter element).

If the condition operator is the `disable` operator and all matching parts never match, the matching pattern always accepts. If the condition expression is the constant `true`, the filter element always accepts.

Note that another cause of the unreachability of the `flowMatch` edge is the unreachability of the `MatchingPart` node. In this case, the unreachability of the `flowMatch` edge is a subordinate conflict.

flowNoMatch edge

If the `flowNoMatch` edge from a `MatchingPart` node is unreachable, the matching part always matches.

If the condition operator is the `enable` condition operator, the always matching matching part causes the matching pattern to always accept. If the condition expression is the constant `true`, the filter element always accepts.

If the condition operator is the `disable` condition operator, the always accepting matching part causes matching pattern to always reject. This causes the filter element to always reject (redundant filter element).

Note that another cause of the unreachability of the `flowNoMatch` edge is the unreachability of the `MatchingPart` node. In this case, the unreachability of the `flowNoMatch` edge is a subordinate conflict.

FilterSet node

The `FilterSet` node is the start node of the simulation. Therefore, this node is always reachable.

FilterModule node

If the `FilterModule` node is unreachable, this indicates the conflict 'unreachable filter module'.

FilterModuleCondition node

The `FilterModuleCondition` node is unreachable if the corresponding `FilterModule` node is unreachable. Therefore, the unreachability of the `FilterModuleCondition`

node is a subordinate conflict.

Filter node
The unreachability of the `Filter` node indicates the conflict 'unreachable filter'.

FilterElement node
The unreachability of the `FilterElement` node indicates the conflict 'unreachable filter element'.

ConditionExpression node
The unreachability of the `ConditionExpression` node indicates the conflict 'unreachable condition expression'. But the `ConditionExpression` node is only unreachable if the `FilterElement` node is also unreachable. Therefore, this conflict is subordinate to the 'unreachable filter element' conflict and is not reported.

ConditionOperator node
The unreachability of the `ConditionOperator` node is either caused by the unreachability of the `FilterElement` node, or by the fact that the condition expression is the constant `false`. In both cases, the unreachability of the `ConditionOperator` node is a subordinate conflict.

MatchingPattern node
The unreachability of the `MatchingPattern` node is either caused by the unreachability of the `FilterElement` node, or by the fact that the condition expression is the constant `false`. In both cases, the unreachability of the `MatchingPattern` node is a subordinate conflict.

MatchingPart node
The unreachability of the `MatchingPart` node indicates the conflict 'unreachable matching part'. A matching part becomes unreachable if an earlier matching part in the matching pattern always accepts.

SubstitutionPart node
The unreachability of a `SubstitutionPart` node is either caused by

- The unreachability of the matching pattern.
- The fact that all matching parts always reject if the condition operator is the `enable` condition operator.
- The fact that one matching part always accepts if the condition operator is the `disable` condition operator.

In all these cases, the unreachability of the `SubstitutionNode` is a subordinate conflict.

FilterAction node
If the unreachable `FilterAction` node is the reject filter action, the filter never rejects, so always accepts. If the unreachable `FilterAction` node is the accept filter action, the filter never accepts, so always rejects.
If the unreachability of certain `FilterAction` nodes means that from that `Filter` node only `Continue FilterAction` nodes are reachable, the filter does nothing else than continue to the next filter. This means that the filter is redundant: it could have been left out without changing the external behavior of the filter set.

Is should also be noted that the unreachability of a filter also leads to the unreachability of the filter elements within the filter. Also, the matching parts within a matching pattern are unreachable if the matching pattern is unreachable. This is a natural con-

sequence of the hierarchical composition of the components in a filter set. Therefore, the unreachability conflicts of the inner components are not reported if the surrounding component in the composition is also unreachable.

7.1.2.2 Cause and Effect Relation Between Conflicts

Most conflicts do not happen in isolation. They are caused by other conflicts. For example, an unreachable filter might be caused by an earlier `Dispatch` filter that always accepts. The unreachable filter is identified by the unreachability of the corresponding `filter` node. But this node is unreachable because its incoming edge is unreachable. This edge is unreachable because its start node is unreachable. This start node is unreachable, because its incoming edges are unreachable, etc. So we can construct a back trace of unreachable nodes and edges. This back trace of unreachable nodes and edges eventually leads to one or more unreachable edges originating from reachable branching nodes. This means that in these branching nodes not every option is possible. The branching node might be a matching part that always accepts, for example. This branching node is the primary cause of all subsequent unreachable nodes and edges. Therefore, the local conflict indicated by the fact that not every outgoing edge from the branching node is reachable is the primary cause of all subsequent conflicts. This conflict will be referred to as the primary conflict.

So, there is a cause and effect relationship between the conflicts in which the primary conflicts cause all other conflicts. Figure 7.2 shows this cause and effect relationship as a graph structure. The primary conflicts are shown on top. They obviously do not have any incoming edge. The following conventions are used in this figure:

- Blocks represent conflicts
- Arrows represent the cause and effect relationship between two conflicts. The cause is the source of the arrow and the effect is the target of the arrow.
- A label on an arrow indicates another condition for this cause and effect relationship. For example, a matching pattern that always accepts only causes a filter element to always accept if the condition expression in the filter element is the constant `true`.

This figure shows, for example, that a matching part that always matches leads to the unreachability of the next matching parts. It also leads to an always accepting matching pattern, if the condition operator is the `enable` condition operator. A filter element that always rejects leads to a redundant filter element. If all filter elements in a filter always reject, the filter always rejects.

One conflict is shown in gray. This 'Filter has type that never continues' conflict indicates a filter that has a filter type that never leads to a continuation to the next filter. This is actually not a conflict, but inherent to the nature of the filter type. An example of such a filter type might be a `DispatchError` filter type, which dispatches on acceptance and gives an error on rejection. Filters of this filter type will never continue. This leads to subsequent filters being unreachable, which are conflicts and should be detected.

The cause and effect relationship can be used to give information to the user about the causes of detected conflicts and the relationship between the detected conflicts. This information might improve the users ability to solve erroneous definitions.

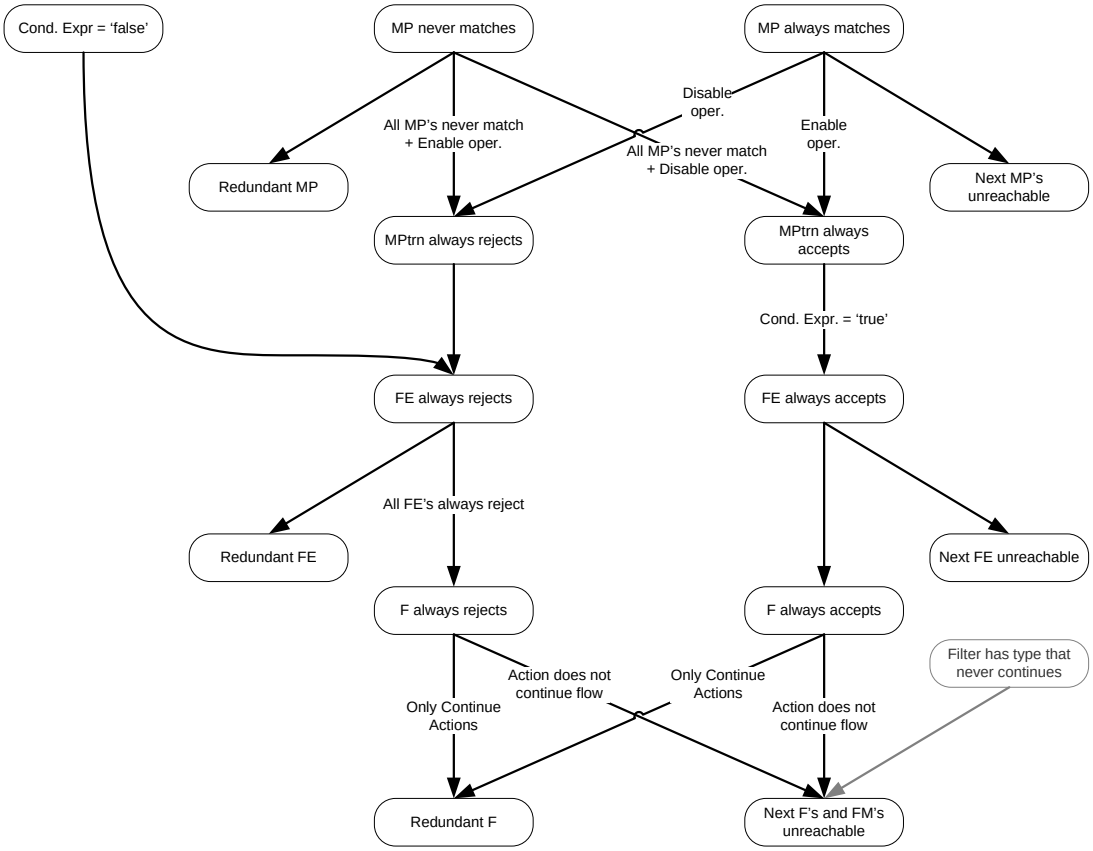


Figure 7.2: The cause and effect relation between conflicts

It should be noted that most primary conflicts concerning a matching part are not caused by a bad defined matching part but by the nature of the composition of earlier filters and filter elements. These earlier filters and filter elements prevent certain messages from reaching the matching part, leading to an always accepting or always rejecting matching part. These causes are not conflicts by themselves. Further research needs to be done about how these causes of primary conflicts can be best presented to users. The primary conflict indicating that the condition expression is the constant `false`, however, is caused by a bad definition and not by earlier filters and filter elements.

7.1.3 Computational Complexity

We will now calculate the time complexity of the algorithm to get an idea about the costs of using it.

The first step in the algorithm is finding the unreachable components in the flowchart. This is done by iterating once over all states and transitions in the execution model. This iteration costs $O(\#states + \#transitions)$ time. As was explained in Chapter 5, a state has at maximum two outgoing transitions. So, the cost to iterate over the execution model is $O(\#states)$. Chapter 5 also explained that the number of states is upper bounded by the number of nodes in the flowchart times the number of distinguishable

messages. So, the time complexity to iterate the execution model can be rewritten to $O(\#messages \cdot \#nodes)$.

The second step in the algorithm is drawing conclusions to the unreachable components in the flowchart. This was done by iterating once over the flowchart to find the unreachable components. The time complexity of this iteration is $O(\#nodes + \#edges)$. This can be simplified to $O(\#nodes)$, as the number of outgoing edges from a node is at maximum two.

So the total complexity is $O(\#messages \cdot \#nodes + \#nodes) = O(\#messages \cdot \#nodes)$. As Chapter 5 explained, the number of distinguishable messages is worst case $O(\#nodes^2)$. This makes the complexity of the algorithm in worst cases $O(\#nodes^3)$. But Chapter 5 also explained that the number of distinguishable messages is usually not related with the number of nodes and remains small. So, in most common situations the time complexity of the algorithm is $O(\#nodes)$. Remember that the number of nodes in the flowchart is linear to the size of the filter set.

So, the algorithm is worst case polynomial to the power three and in most cases linear in the size of the filter set. This makes the algorithm usable in practice.

7.2 Design and Implementation

This section describes how consistency reasoning is implemented in the Compose⋆ project. It first describes the position of consistency reasoning engine within the architecture of Compose⋆. This is followed by an explanation of the class structure of the consistency reasoning engine. Next, some implementation details of the consistency reasoning algorithm are given. Finally, various options of reporting the conflicts are discussed.

7.2.1 Position in the Compose⋆ Architecture

Figure 2.3 on page 21 shows the architecture of Compose⋆ and the position of consistency reasoning within this architecture. Consistency reasoning depends on filter reasoning, on superimposition being resolved and on the filter module orders being created. Therefore, it is placed after the filter reasoning preprocessor, the superimposition analysis and the filter composition and checking.

7.2.2 Structure of the Consistency Reasoning Engine

In this section the class diagram of the consistency reasoning engine module is presented. Figure 7.3 shows the class diagram of the consistency reasoning engine. The classes are all contained in the package `Composestar.Core.CORE2`

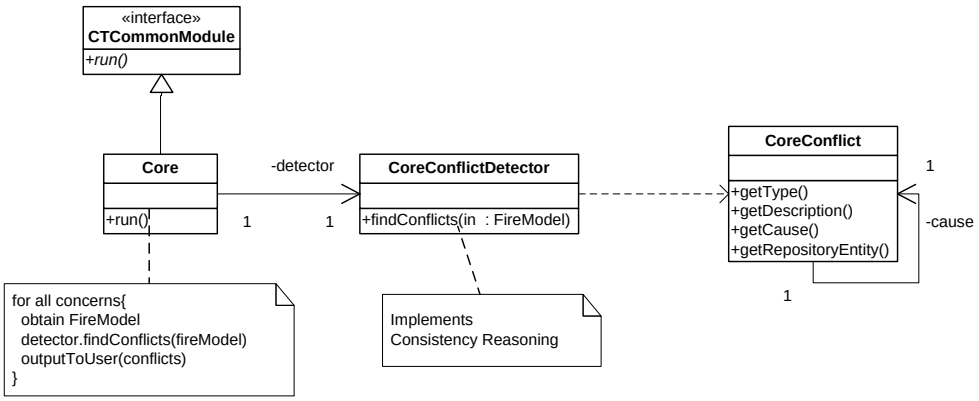


Figure 7.3: CORE class diagram

Core

`Core` is the main component in the consistency reasoning engine. It implements the `CTCommonModule` interface to make it a module in the Compose⋆ project. This component is responsible for iterating over all concerns, obtaining the corresponding fire models from `FIRE`, calling the `CoreConflictDetector` to find conflicts and notifying the user about the conflicts.

CoreConflictDetector

`CoreConflictDetector` is the component that actually implements the consistency reasoning algorithm. This implementation corresponds to the algorithm outlined in the theory section: first an iteration over the state space is done to find all

reachable components of the flowchart. Then an iteration over the flowchart is done to find the unreachable components. From these unreachable components conclusions are drawn in the form of `CoreConflict` instances. During the iteration the causing conflicts are also maintained to create the cause and effect relationship between the conflicts.

CoreConflict

Instances of `CoreConflict` represent found conflicts. `CoreConflict` maintains the type of the conflict, a description of the conflict, a link to the `RepositoryEntity` to which the conflict applies and the causing conflict, if there is any.

7.2.3 Implementation of the Consistency Reasoning Algorithm

This section describes some implementation details of the consistency reasoning algorithm. It first explains how finding reachable components is implemented. This is followed by an explanation about how unreachable components are checked for conflicts and how the cause and effect relationship between conflicts is created.

7.2.3.1 Finding Reachable Components

The first step in the consistency reasoning algorithm is finding the reachable nodes and edges of the flowchart by iterating over all states in the execution model. To iterate over all states the `ExecutionStateIterator` from the package `FIRE2.util.iterator` is used. For each state the corresponding flow node is added to a hashset. All outgoing execution transitions from the state are iterated and their corresponding flow transitions are also added to the hashset. After the iteration, the hashset contains all reachable nodes and edges. Figure 7.4 shows the UML sequence diagram for this part of the implementation.

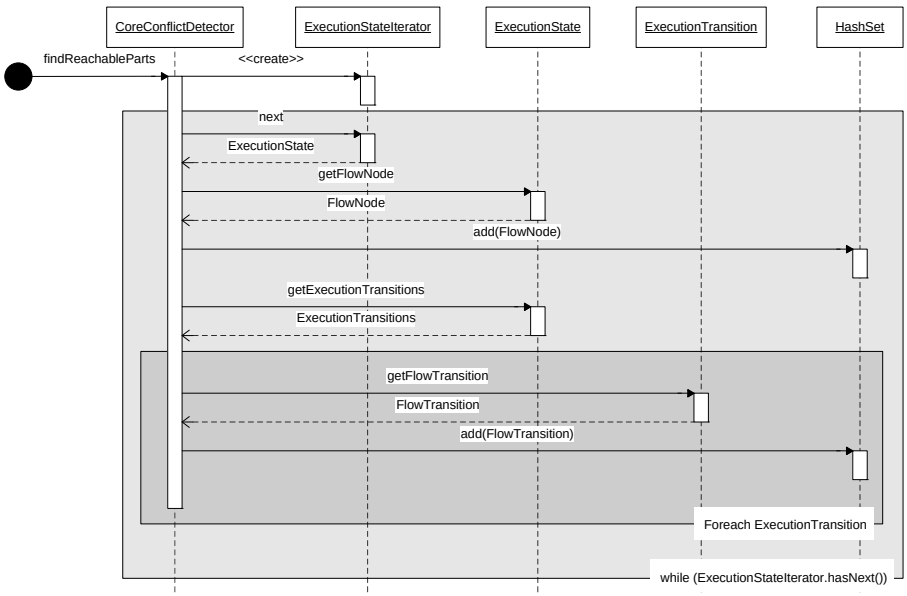


Figure 7.4: Sequence diagram showing how reachable components are found

7.2.3.2 Checking Unreachable Components

After the reachable components are found, the next step is checking the unreachable components for conflicts. The unreachable components are found by iterating over the nodes in the flowchart and checking whether the node and its outgoing transitions are reachable. Figure 7.5 shows the UML sequence diagram for this part of the implementation.

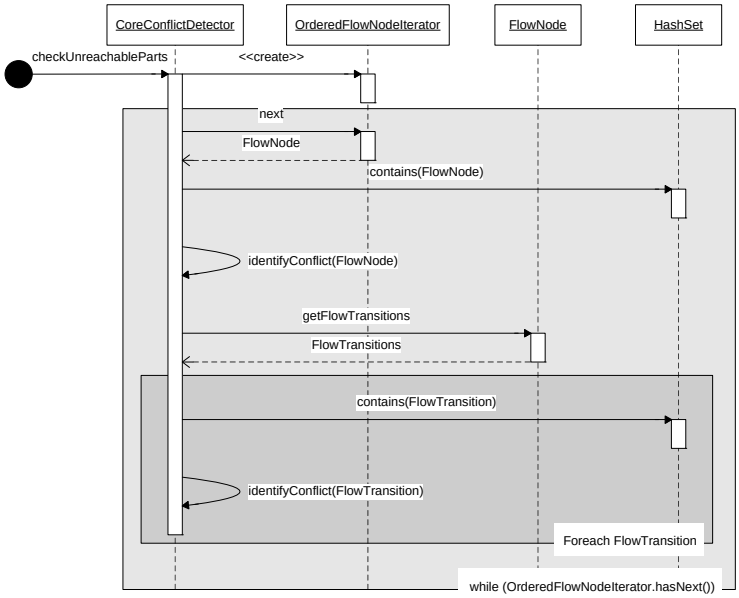


Figure 7.5: Sequence diagram showing how unreachable components are identified and checked for conflicts

To iterate over the nodes in the flowchart, the `OrderedFlowNodeIterator` from the package `FIRE2.util.iterator` is used. This iterator iterates over the nodes in such a way that a node is not visited until all nodes in all incoming traces have been visited. This is possible because there are no loops in the flowchart. Iterating in such an order makes it easier to create the cause and effect relationship between the conflicts, because causes happen earlier in the flowchart than their effects. For example, a filter node is not visited until all nodes corresponding to the previous filter have been visited. If the filter node happens to be unreachable, we know for sure that the conflict causing this has already been detected. This is the case because this causing conflict originates from a previous filter, an always accepting `Dispatch` filter for example, and all flow nodes corresponding to this filter have already been visited. In this way, the earlier conflict can immediately be used as the cause of the unreachable filter conflict and does not have to be identified first.

For each flow node a check is done whether it is contained in the hashset containing the reachable components. If it is not contained in this hashset, the flow node is unreachable. In this case, the `identifyConflict` method is called to check whether this unreachable flow node indicates a conflict.

If the flow node is contained in the hashset, the flow node is reachable. Then all

outgoing flow transitions are checked whether they are contained in the hashset. If the hashset does not contain one of these outgoing flow transitions, the `identifyConflict` method is called to check whether the unreachable flow transition indicates a conflict.

The `identifyConflict` method implements the mapping shown in Table 7.1. It checks the type of the unreachable flow node or flow transition and maps this to the corresponding conflicts.

7.2.4 Reporting Conflicts

This chapter explained how consistency conflicts can be detected. This section discusses four different approaches to report the conflicts to the user.

Complete Cause and Effect Information One approach to conflict reporting is to report each conflict and its direct cause. For the example in Listing 7.1 on page 108 the conflict report will be as shown in Listing 7.2.

```
2,30: Matching part always matches
2,30: Matching pattern always accepts, because matching part always
    matches
2,22: Filter element always accepts, because matching pattern always
    accepts and the condition expression is the constant true
2,1: Filter disp1 always accepts, because a filter element always accepts
3,1: Unreachable filter disp2, because filter disp1 always accepts
```

Listing 7.2: Consistency reasoning complete output

This example shows that a lot of information is reported. Therefore, we will discuss other approaches.

Report only Primary Conflicts Another approach is to report only the primary conflicts. This would give the output shown in Listing 7.3 for the example.

```
2,30: Matching part always matches
```

Listing 7.3: Consistency reasoning primary conflict output

This gives very little information. From this information it is, for example, not clear that the always accepting matching part in one filter causes another filter to be unreachable.

Report only Leaf Conflicts and their Primary Causes A third approach is to report the leaf conflicts and their primary causes. The leaf conflicts are the conflicts that do not cause other conflicts. All conflicts between the primary conflict and the leaf conflict are not reported. This gives the output in Listing 7.4.

```
3,1: Unreachable filter disp2, because matching part at (2,30) always
    matches.
```

Listing 7.4: Consistency reasoning leaf conflict + root cause

The problem with this approach is that the step taken is rather big: we go from one matching part that always accepts deep within one filter to the unreachability of another filter.

Different Levels of Reporting The last approach that is discussed here is the approach in which the user can indicate at which level in the filter set the reporting should stop. Conflicts concerning elements on the same level or higher level are reported. Conflicts concerning elements on a lower level are not reported. For example, a user might indicate that reporting should stop at the filter element level. Then only conflicts concerning filter elements, filters and filter modules are given. If this is applied to the example, the following output in Listing 7.5 is generated.

```
2,22: Filter element always accepts
2,1: Filter disp1 always accepts, because a filter element always accepts
3,1: Unreachable filter disp2, because filter disp1 always accepts
```

Listing 7.5: Consistency reasoning level output

This approach gives the user the ability to adjust the level of detail in the reports, while the direct cause and effect steps are maintained. Therefore, this seems the best approach.

Currently, the complete cause and effect information is given. Some future work might be to implement the different levels of reporting, because this gives the user control over the amount of details provided by the consistency reasoning engine.

Chapter 8

Signature Generation

"A signature always reveals a man's character -
and sometimes even his name.".
Evan Esar
American Humorist (1899 - 1995)

All messages to a concern pass through the input filters. The input filters can manipulate the incoming messages. Thereby, they can change the signature of the concern. They can remove methods, by preventing those methods from being dispatched. They can also add new methods, by dispatching them to an existing method in one of the implementation objects.

It is the task of the compiler to generate the modified signatures of the concerns. This generation of the modified signatures is no easy task. Signature matching parts and dispatches to other concerns make the signature of a concern depend on the signatures of other concerns. Certain conflicts might occur, such as cyclic dependency conflicts and infinite signatures. Also, type errors occur if a method can be dispatched to a non-existent target method.

This chapter explains how the modified signatures can be generated, which conflicts and type errors might occur and how these conflicts and type errors can be detected. The first section explains the theory and concepts behind signature generation. It presents a model that can be used to generate the signatures. A problem with that model is, however, that it is an infinite structure. The second section, therefore, presents a finite algorithm, using the concepts introduced in the first section. The third section describes how signature generation is implemented in Compose⋆.

8.1 Theory and Concepts

This section explains the theory and concepts behind signature generation and type checking. Signature generation is not new. There already exists an approach to generate the signatures. This approach, however, has some problems. This section therefore starts with explaining the existing approach to signature generation and the problems with this approach. One of the main problems with the existing approach is that Holljen's definition of signature generation is wrongly interpreted. The new approach is based

on a different, more literal, interpretation of Holljen's definition. This interpretation is explained in the second part of this section.

The existence of a method in the signature of a concern depends on the existence of other methods. The third part of this section presents a technique to model these dependencies for all methods. The resulting model is called a dependency graph. A dependency graph can be used to generate the signatures.

Generating the signatures is not always straightforward. There might be certain constructs in a filter set that lead to conflicts. Examples of these conflicts are infinite signatures or cyclic dependencies. Also type errors can occur. The fifth part of this section explains all conflicts and type errors that can occur and how they can be detected in the dependency graph.

The `Dispatch` action does not have to be the only filter action that acts as a requirement for the inclusion of a method in the signature. Other filter action can also be used. The last part of this section explains how other filter actions can be used as a sufficient requirement for the inclusion of a method in the signature.

8.1.1 Old Approach to Signature Generation and Type Checking

Signature generation is not new. Holljen already described a first approach to signature generation [19]. Holljen's approach was later improved by Bosman [5]. This section summarizes the old approach to signature generation and describes a number of problems with this approach.

The old approach to signature generation consists of two phases:

- First, the maximum signature is incrementally generated by taking the union of the `inner` signature and the signature of the internals and externals. These methods are all marked `Unknown`. So the maximum signature of a concern consists of the signatures of its implementation objects.

- For each `Unknown` method it is checked whether it can be dispatched. If the method can be dispatched it is marked as `InSignature`. If the method cannot be dispatched, it is marked as `NotInSignature`. The status remains `Unknown` if a signature matching part is encountered in the filter set for which it is not known whether it accepts or rejects (due to the fact that other methods still have the status `Unknown`).

  This is done iteratively until there are no more `Unknown` methods or until there are `Unknown` methods, but nothing changes anymore (an erroneous situation).

8.1.1.1 Problems with this Approach

The old approach to signature generation has some problems.

Maximum Signature to Strict Holljen defines signature generation as "Considering the input filters of a concern, find the maximum input leading to an output" [19]. Although this is a good definition, the old approach to generate the signatures, however, bounds the maximum input to the union of the signatures of the implementation objects. Bounding the maximum input in this way leads to the erroneous exclusion of certain methods from the signature.

Listing 8.1 shows an example of this problem. The `inner` implementation contains a method `existingMethod`. The `Dispatch` filter dispatches all messages with selector `newMethod` to this method `existingMethod`. So `newMethod` should be in the signature of the concern, because it can be dispatched. But if `newMethod` does not already exist in the signature of `inner` or in the signature of one of the internals and externals, then it is not added to the signature of the concern. So the signature is not complete in this case.

```
disp1 : Dispatch = { True => [*.newMethod] inner.existingMethod }
```

Listing 8.1: Example of the incomplete signature problem

Type Compatibility with Dispatch Method not Checked The old approach does not check the type compatibility between the original method and the target method of the dispatch. It only checks whether a call to the original method can reach a `Dispatch` action. In this case it adds the method to the signature.

```
Concern A {
  filtermodule FMA {
    inputfilters
      disp1 : Dispatch = { True => [*.m] inner.n }
  }

  implementation in Java {
    public class A {
      public void m() {}
      public void n(int) {}
    }
  }
}
```

Listing 8.2: Example of a type problem

Listing 8.2 gives an example that leads to a type problem in the old approach. The filters in this example dispatch method `m()` to method `n(int)` in `inner`. This is a problem, because the parameter types are not compatible. Method `m()` should therefore not be in the signature of the concern. The old approach, however, only checks whether a `Dispatch` action can be reached, which is the case. Therefore, method `m()` is added to the signature of the concern.

This problem is not a conceptual problem of the approach. Type checking could have been easily implemented by not only checking whether a `Dispatch` action can be reached, but also checking whether the dispatch method has the same types. Actually, Holljen already identified this problem and mentions that type checking should be performed in the Compose⋆ compiler [19]. Unfortunately, type checking was not incorporated into the signature checking algorithm.

8.1.2 A Different Interpretation of Holljen's Definition

The previous section explained the old approach to signature generation and the problems with this approach. One of the main problems is that not the right signature is generated. This is mainly due to a bad interpretation of Holljen's definition of signature generation.

Holljen defines signature generation as "Considering the input filters of a concern, find the maximum input leading to an output" [19]. In the old approach this was interpreted as: 'The maximum signature of a concern is defined by the signature of the implementation objects. The filter set restricts this maximum signature to the final signature by preventing certain methods from being dispatched.'

In the new approach we interpret Holljen's definition of signature generation differently. We interpret 'maximum input' as really the maximum input: a subset of the set of all possible methods. The filter set composes the signature of the concern by creating a partial mapping from the set of all possible methods to the set of existing methods in the signature of the implementation objects (`inner`, internals and externals) and the signature of `self`. All methods for which this mapping is defined should be in the signature of the concern. Note the difference between the signature of `inner` and the signature of `self`. `inner` stands for the implementation of the concern. So, the signature of `inner` is defined by the implementation. `self` stands for the concern self: the inner implementation with the input filters superimposed. So, the signature of `self` is the signature of the concern. This is the signature that is generated. But this signature might also depend on itself: a dispatch might be done to `self`.

8.1.3 Representing the Signature Mapping as a Dependency Graph

The previous section presented our interpretation of Holljen's definition of signature generation. This section presents a conceptual model to construct the signature mapping defined by a filter set.

A method is in the signature of a concern if it can be dispatched. So the existence of the method depends on the existence of the dispatch method. But this is not the only dependency. Along the path to the `Dispatch` action there might be signature matching parts. These signature matching parts also introduce dependencies. They must result in either an accept or a reject for that specific path. Signature matching parts that must accept form a positive dependency: the method that is checked in the signature matching part must exist to make the signature matching part accept. Signature matching parts that must reject form a negative dependency: the method that is checked in the signature matching part must not exist to make the signature matching part reject.

Filter sets might also introduce conflicts. An example of such a conflict is the cyclic dependency conflict; if a cyclic dependency conflict occurs, the existence of certain methods depend on their own existence. The infinite signature conflict is another example of a conflict. This conflict causes signatures with an infinite number of methods. (For more information about conflicts, see Section 8.1.5).

All these dependencies and conflicts make signature generation complex. Therefore, we will now present a conceptual model that can be used to represent these dependencies. This model is based on a graph structure and is therefore called a *dependency graph*. Although this model is infinite, so not directly practically applicable, it will give us insight into the method mapping defined by the filter set.

The dependency graph we introduce here is inspired by Bosman's dependency graph [5]. Bosman also uses a graph to represent the dependencies between methods. He, however, only introduced this model informally and only uses it for illustration purposes. Bosman's dependency graph model is too coarse-grained to be used here. It

does not give all dependency information and also does not show the relation between dependencies and paths through the filter set.

Definition 8.1.1 (Dependency Graph) *A dependency graph is a graph structure that represents the signature dependencies of all possible methods for all concerns. It represents how the existence of a method in the signature of a concern depends on the existence of or the absence of certain other methods. Because the number of possible methods is infinite, this model is infinite.*

Nodes *There are two types of nodes in the graph:*

- Method nodes*: The method nodes represent each possible method in each concern. Because the number of all possible methods in a concern is infinite, there are also an infinite number of method nodes.*
- Path nodes*: A path node represents a single execution path through the filter set to a* `Dispatch` *action for a certain method. A path node therefore corresponds to one specific method node. A path node has one incoming edge, from the corresponding method node. For each dependency in that path it has an outgoing dependency edge to the specific method node on which it depends. A method node can have multiple corresponding path nodes, one for each path to a* `Dispatch` *action.*

Edges *There are two types of edges in the graph:*

- Path edges*: A path edge is an edge from a method node to a path node. Path edges indicate the relationship between a method and its corresponding paths through the filter set to a* `Dispatch` *action.*
- Dependency edges*: Dependency edges are edges leaving from a path node to a method node. They indicate a dependency for that path. If all dependencies are resolved, the path is possible.*

  There are two different dependency types:

  - Signature matching dependency*: Dependency created by a signature matching part. This type has two sub types:*
    - Exist signature matching dependency*: The checked method must exist (the signature matching part must accepts).*
    - Not-exist signature matching dependency*: The checked method must not exist (the signature matching part must rejects).*
  - Dispatch dependency*: Dependency on the existence of the dispatch method.*

Example We will now give an example of a dependency graph, to illustrate this concept. Listing 8.3 shows the concerns in this example. There are two concerns, concern `A` and concern `B`.

Concern `A` has an internal `b` of type `B`. The first filter of concern `A` checks whether the message is in the signature of internal `b`. If this is the case, the message is dispatched to `b`. The second filter is the default inner dispatcher. The `inner` implementation of concern `A` has two methods, `m()` and `n()`.

Concern `B` has no internals or externals. The first filter of concern `B` checks whether the selector equals `z`. If this is the case, a dispatch is done to `inner.x`. The second filter is the default inner dispatcher. The `inner` implementation of concern `B` has two methods, `x()` and `y()`.

```
concern A {
  filtermodule FMA {
    internals
      b : B
    inputfilters
      disp1 : Dispatch = { true => <b.*> b.*};
      disp2 : Dispatch = { true => <inner.*> inner.*}
  }

  implementation in Java {
    public class A {
      public void m() {}
      public void n() {}
    }
  }
}

concern B {
  filtermodule FMB {
    inputfilters
      disp3 : Dispatch = { true => [*.z] inner.x};
      disp4 : Dispatch = { true => <inner.*> inner.*}
  }

  implementation in Java {
    public class B {
      public void x() {}
      public void y() {}
    }
  }
}
```

Listing 8.3: Example to illustrate the dependency graph concept

Figure 8.1 shows a partial representation of the dependency graph. Only the methods that will actually be in one of the signatures and their dependencies are shown. Note that the complete dependency graph cannot be shown, because it is an infinite structure. The following conventions are used:

- Filled dot: Method node.
- Unfilled dot: Path node.
- Label `E`: Exist signature matching dependency.
- Label `N`: Not-exist signature matching dependency.
- Label `D`: Dispatch dependency.
- If there are more dependency edges between two nodes, they are represented as one edge. The labels are represented as a comma separated list.

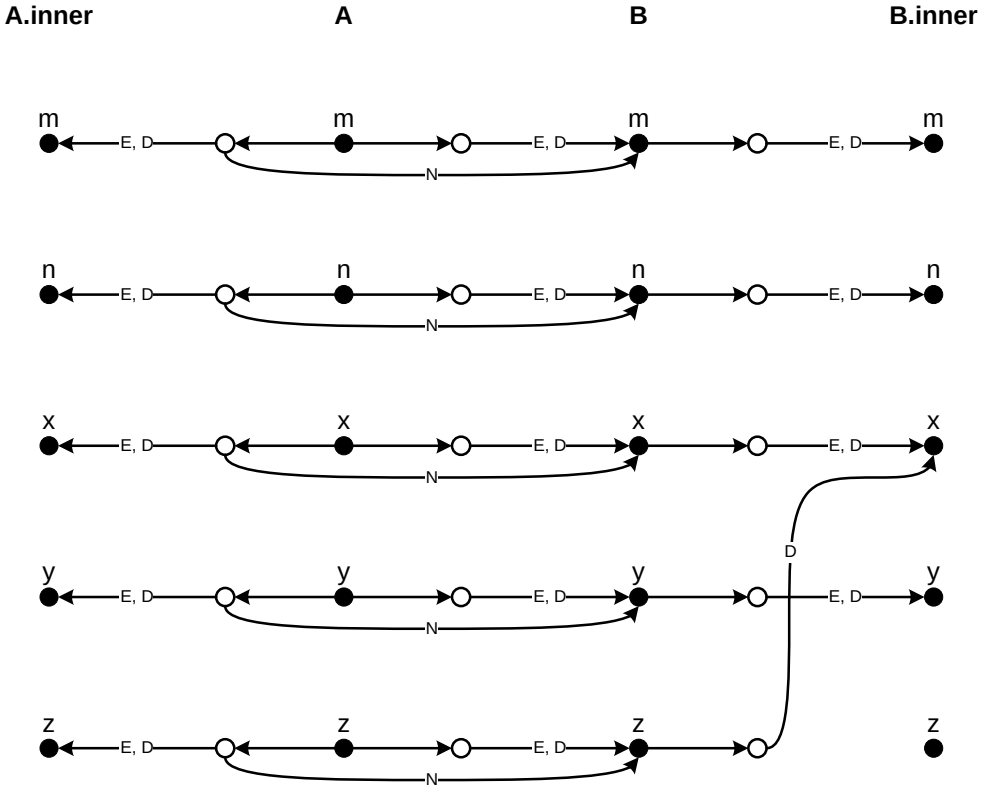


Figure 8.1: Partial representation of the dependency graph.

Note that this figure does not show the actual methods in the signature, but all potential methods, and their dependencies on the existence or absence of other potential methods.

The figure shows that all methods in concern `A` have a path with an `E` dependency and a `D` dependency on the same method in concern `B`. Filter `disp1` is responsible for this path. All methods in concern `A` also have a second path. That second path has an `E` dependency and a `D` dependency on the same method in `inner`, caused by filter `disp2`. It also has an `N` dependency on the same method in concern `B`, caused by the rejection of filter `disp1`.

In concern `B`, the methods `m()`, `n()`, `x()` and `y()` all have one path with an `E` dependency and a `D` dependency on the same method in `inner`. This path is caused by filter `disp4`. The method `z()` in concern `B` has one path with a `D` dependency on `x()` in `inner`, because of filter `disp3`.

The resulting dependency graph does not directly indicate the signature of the concern. How the signature can be generated from the dependency graph is explained in the next section.

8.1.4 Using the Dependency Graph to Generate the Signatures

The previous section presented the dependency graph as a conceptual model to represent the existence dependencies between methods. This section presents a procedure to

generate the signatures of the concerns, using the dependency graph.

The procedure consists of four steps. These four steps are informally described next, followed by an explanation of each step.

1. Initialization:
   - Method nodes in the `inner` implementations that are actually implemented are marked `Existing`.
   - Method nodes in the `inner` implementations that are not implement are marked `NotExisting`.
   - All other method nodes are marked `Unknown`.
   - All path nodes are marked `Unknown`.
2. Check for each `Unknown` path node whether all dependencies are resolved. To resolve the dependencies, the following rules apply:
   - A dependency is resolved if:
     - Exist signature matching dependency: The target method node is marked `Existing`.
     - Not-exist signature matching dependency: The target method node is marked `NotExisting`.
     - Dispatch dependency: The target method node is marked `Existing`.
   - A dependency is violated if:
     - Exist signature matching dependency: The target method node is marked `NotExisting`.
     - Not-exist signature matching dependency: The target method node is marked `Existing`.
     - Dispatch dependency: The target method node is marked `NotExisting`.
   - A dependency is unknown if the target method node is marked `Unknown`.

   To resolve a path node, the following rules apply:
   - A path node is marked `Resolved` if and only if all dependencies are resolved.
   - A path node is marked `Violated` if and only if at least one of its dependencies is violated.
   - A path node is marked `Unknown` if and only if no dependencies are violated and at least one dependency is unknown.
3. Check for each `Unknown` method node whether it exists:
   - Mark the method node `Existing` if and only if at least one of its corresponding path nodes is marked `Resolved`.
   - Mark the method node `NotExisting` if and only if all its corresponding path nodes are marked `Violated` or the method node has no corresponding path nodes.

- Mark the method node `Unknown` if and only if none of its corresponding path nodes are marked `Resolved` and at least one of its corresponding path nodes is marked `Unknown`.

4. Repeat steps 2 and 3 until there are no more changes.

Step 1

Step 1 is the initialization step. Method nodes in the inner implementations can be marked either `Existing` or `NotExisting`, because the signatures of inner implementations are known. All other method nodes and all path nodes are marked `Unknown`, because their existence depends on the existence of other methods.

Step 2

Step 2 checks whether `Unknown` path nodes can already be resolved. If all dependencies are resolved, the path is possible and the dispatch can be executed. Therefore, the path node is marked `Resolved`. If one of the dependencies is violated, then either the dispatch method does not exist or the path is impossible, due to a signature matching part. Therefore, the path node is marked `Violated`. If there are no violated dependencies but there are unknown dependencies, then it is not certain yet whether the path is resolved or violated. In that case, the marking `Unknown` of the path node is maintained.

Step 3

Step 3 checks whether the existence of `Unknown` method nodes can already be resolved. If one of its corresponding path nodes is marked `Resolved`, the method can be dispatched. Therefore, it should be in the signature of the concern. In this case the method node is marked `Existing`.

If all corresponding path nodes are marked `Violated`, however, the method can never be dispatched and should not be in the signature of the concern. Therefore, the method node is marked `NotExisting`.

If the method node has no corresponding path nodes, then there is no path to a `Dispatch` action. This means that the method can never be dispatched. The method node is therefore marked `NotExisting`.

If the method node has no `Resolved` path nodes, but there are still `Unknown` path nodes left, it is not certain yet whether the method should be in the signature. Therefore, the marking `Unknown` of the method node is maintained.

Step 4

Steps 2 and 3 resolve the signatures incrementally. If certain `Unknown` method nodes are resolved in step 3, certain `Unknown` path nodes might be resolved in step 2. This might cause that certain other `Unknown` method nodes are resolved in step 3. Therefore, steps 2 and 3 are repeated until no more `Unknown` path nodes and `Unknown` method nodes are resolved.

One might argue that there may be dependency sequences of infinite length, because the dependency graph is an infinite structure. In that case, the procedure will never end. Dependency sequences of infinite length are, however, not possible,

because composition filters are not capable of creating them. A filter set is only capable of substituting to a finite set of selectors; if the dependency sequence is long enough, ultimately the same selector is encountered again, leading to a cyclic dependency. To create an infinite sequence, a filter set should be capable to substitute to an infinite set of selectors. In this way it can be prevented that the same selector is encountered again. For example, if composition filters could substitute a given selector to a new selector that is the given selector with a character 'a' appended, it substitutes to an infinite set of selectors. If the message is repeatedly dispatched to `self`, an infinite dependency sequence is created. Composition filters are not capable of doing this.

Example Applying the procedure to the dependency graph in Figure 8.1 gives the result shown in Figure 8.2. `Violated` path nodes and `NotExisting` method nodes are drawn in grey. `Resolved` path nodes and `Existing` method nodes are drawn in black.

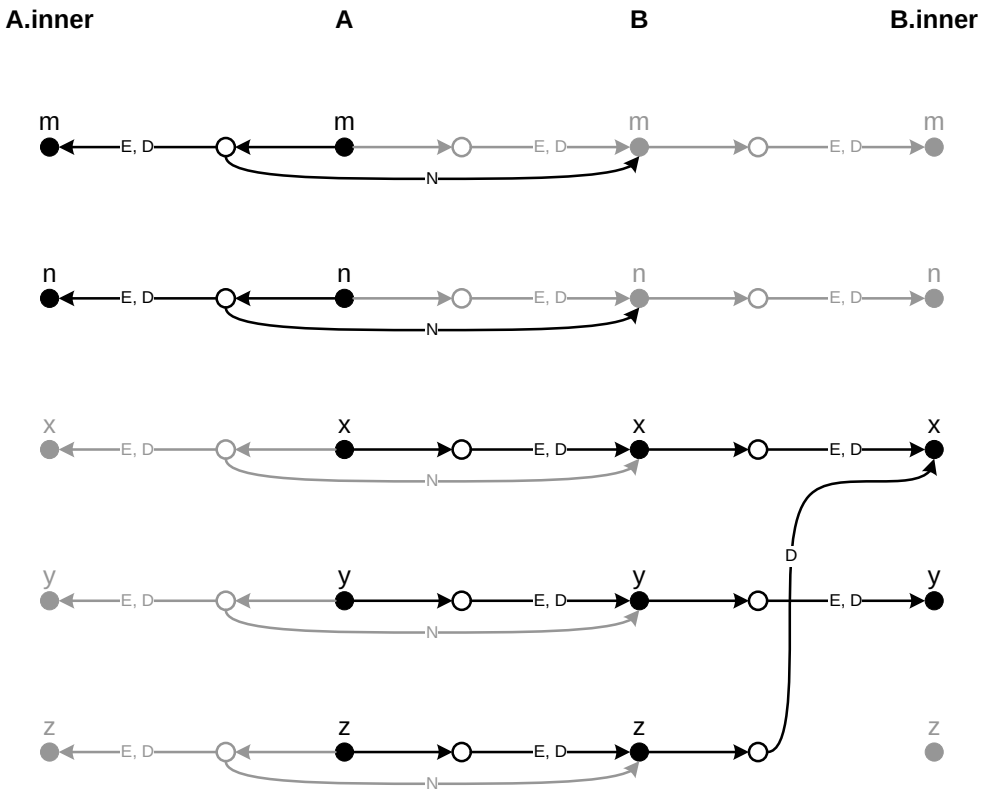


Figure 8.2: The existence of methods resolved in the dependency graph.

This figure shows that concern `A` gets methods `m()` and `n()`, because they can be dispatched to `inner`. Concern `A` also gets methods `x()`, `y()` and `z()`, because they can be dispatched to concern `B`.

Concern `B` gets methods `x()` and `y()`, because they can be dispatched to `inner`. Concern `B` also gets method `z()`, because it can be dispatched to method `x()` in `inner`. Concern `B` does not get methods `m()` and `n()`, because they cannot be dispatched.

8.1.5 Conflicts and Type Errors

The previous section explained how the signatures can be generated using the dependency graph. Signatures cannot always be generated successfully. There might be certain conflicts and type errors. This section presents the conflicts and type errors that can occur and explains how they can be detected.

8.1.5.1 Cyclic Dependency Conflict

The existence of a method depends on the existence of paths through the filter set to a `Dispatch` action. If such a path depends back on the method itself, a *cyclic dependency* is created. The dependency can be direct: the path directly refers back to the method. The dependency can also be indirect: the path refers through a sequence of dependencies on other methods back to the original method.

A cyclic dependency does not have to be a conflict. A *cyclic dependency conflict* occurs if the existence of the methods in the cycle fully depend on that cycle. This is the case if:

- All method nodes in the cycle have no other path nodes that are resolved. This means that the existence of the methods in the cycle cannot be confirmed in another way.
- All path nodes in the cycle do not have other dependencies that are violated. This means that the violation of the path nodes, and thereby the non-existence of the methods, cannot be confirmed.

A cyclic dependency conflict makes the signature ambiguous: if the method is in the signature, it can (or cannot) be dispatched. If the method is not in the signature, it cannot (or can) be dispatched.

Example Listing 8.4 gives an example of a cyclic dependency conflict. In this example, the existence of method `m` in concern `A` depends on the existence of method `m` in concern `B`. The existence of method `m` in concern `B` depends again on the existence of method `m` in concern `A`. Figure 8.3 shows this cyclic dependency conflict in the dependency graph.

```
concern A {
  filtermodule FMA {
    internals
      b : B
    inputfilters
      disp : Dispatch = { true => [*.m] b.m};
  }
}

concern B {
  filtermodule FMB {
    internals
      a : A
    inputfilters
      disp : Dispatch = { true => [*.m] a.m};
  }
```

```
}
```

Listing 8.4: Example of a cyclic dependency conflict

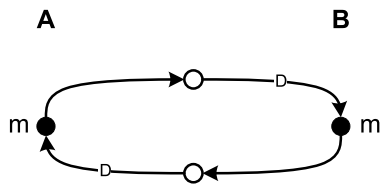


Figure 8.3: Cyclic dependency in the dependency graph.

Pattern To detect cyclic dependency conflicts, it must be checked whether there are still `Unknown` method nodes left in the dependency graph after the signature generation procedure has ended. If there are `Unknown` method nodes left, there is a cyclic dependency conflict.

If there are `Unknown` method nodes, there is always a cyclic dependency conflict. If there is not a cyclic dependency conflict, then there is no method node of which the existence fully depends on its own existence. This means that the existence of all method nodes ultimately depend on method nodes for which the existence can be resolved: the method nodes in the `inner` implementations and the method nodes without path nodes to a `Dispatch` action. This means that the existence of the `Unknown` method nodes could have been resolved and would have been resolved in the signature generation algorithm. So, if there are `Unknown` method nodes, there is a cyclic dependency conflict.

If there are no `Unknown` method nodes, on the other hand, there is no cyclic dependency conflict. If a cyclic dependency conflict occurs, the existence of all methods in the cycle fully depend on that cycle. Because all methods in the cycle initially have the status `Unknown`, they keep having that status. So, if there is a cyclic dependency conflict, there are `Unknown` method nodes.

So, `Unknown` method nodes are a necessary and sufficient requirement for a cyclic dependency conflict. Note that not every `Unknown` method needs to be in the cycle. Just depending on a cycle is enough, as can be seen in Figure 8.4 . The existence of method `m` depends on the cycle. Because the methods in the cycle keep having the status `Unknown`, method `m` keeps having the status `Unknown`.

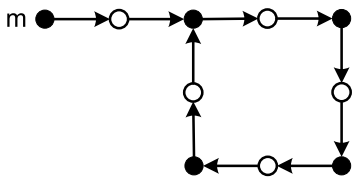


Figure 8.4: An `Unknown` method node does not need to be in the dependency cycle.

Relationship Between Cyclic Dependency Conflicts Usually, cyclic dependency conflicts do not occur as isolated cases. They occur as part of a set of cyclic dependency

conflicts that are caused by the same anomaly in the filter set. Listing 8.4 on page 132, for example, does not introduce one cyclic dependency conflict, but an infinite number of cyclic dependency conflicts; there is a cyclic dependency conflict for all methods named `m` with all possible parameter types and return type.

Such related cyclic dependency conflicts form a set of cyclic dependency conflicts. There are four types of such cyclic dependency conflict sets. These four types are explained here, because the identification of a cyclic dependency conflict is in practice slightly different for each type, as will be explained in Section 8.2. The four types of cyclic dependency conflict sets are:

- *Infinite cyclic dependency conflict set*: Both the name and the types are from an infinite set. An example of this is shown in Listing 8.5. Concern `A` dispatches all methods to concern `B`. Concern `B` dispatches all methods to concern `A` again. So, if a method is in the signatures of concern `A` and concern `B`, it can be dispatched. If the method is not in the signatures of concern `A` and concern `B`, it cannot be dispatched.

```
concern A {
 ...
      disp : Dispatch = { true => [*.*] b.*}
}

concern B {
...
      disp : Dispatch = { true => [*.*] a.*}
}
```

Listing 8.5: Example leading to an infinite cyclic dependency conflict set

- *Name finite cyclic dependency conflict set*: The set of names is finitely bounded. The set of types is an infinite set. Listing 8.4 on page 132 is an example of this type of cyclic dependency conflict set. The name is bounded to `m`. All types are possible.

- *Type finite cyclic dependency conflict set*: The set of types is finitely bounded. The names are from an infinite set. Listing 8.6 gives an example of this type of cyclic dependency conflict set. Methods in the signature of concern `A` can only be dispatched to `inner.s` if they are in the signature of concern `B`. Methods in the signature of concern `B` can only be dispatched if they are in the signature of concern `A`. Because concern `A` dispatches all methods to `inner.s`, the types of all methods in the signature of concern `A` are bounded by the types of the methods named `s` in `inner`. Therefore, the types of the methods in the cyclic dependency conflict set are bounded by the types `s` has in `inner`. The names are still from an infinite set.

```
concern A {
 ...
    disp : Dispatch = { true => <b.*> inner.s}
}

concern B {
```

```
...
    disp : Dispatch = { true => [*.*] a.*}
}
```

Listing 8.6: Example leading to a type finite cyclic dependency conflict set

- *Fully finite cyclic dependency conflict set*: Both the set of possible names as the set of possible types is finitely bounded. The example in Listing 8.6 would have been a fully finite cyclic dependency conflict set if the `Dispatch` filter in concern `A` had dispatched to `inner.*` instead of `inner.s`. Then only the methods that are also in `A.inner` could have formed the cyclic dependency conflict. All other methods would be marked `NotExisting`.

Paradoxical Cyclic Dependency Conflict If a cyclic dependency conflict occurs, the existence of a method depends on its own existence or absence. If the existence of a method depends on its own absence, this is a *paradoxical cyclic dependency conflict*. This kind of cyclic dependency conflict indicates that the method exists if and only if it does not exist.

Listing 8.7 gives an example of a paradoxical cyclic dependency conflict. If method `m` is in the signature of concern `A`, it cannot be dispatched. Therefore, it should not be in the signature of concern `A`. If method `m` is not in the signature of concern `A`, however, it can be dispatched. Therefore, it should be in the signature of concern `A`.

```
concern A {
  filtermodule FMA {
    inputfilters
      error : Error = { true ~> <self.m> };
      disp : Dispatch = { true => <inner.*> inner.* }
  }

  implementation in Java {
    public class A {
      public void m() {}
    }
  }
}
```

Listing 8.7: Example of a paradoxical cyclic dependency conflict

In general, a cyclic dependency conflict is paradoxical if there are an odd number of not-exist signature matching dependencies in the dependency cycle. Paradoxical cyclic dependency conflicts are detected in the same way as normal cyclic dependency conflicts.

8.1.5.2 Cyclic Dispatch Conflict

Related to the cyclic dependency conflict is the *cyclic dispatch conflict*. A cyclic dispatch conflict occurs if dispatching can become cyclic. This might happen if the message enters a filter set again after a dispatch. If more of such dispatches happen after each other, it is possible that eventually a message is dispatched that has already been dispatched earlier in the dispatch sequence. In this way, a loop is formed that might lead to an infinite dispatch sequence. This infinite dispatch sequence is problematic. Therefore,

cyclic dispatching is a conflict that should be detected. Note that a cyclic dispatch conflict is a cyclic dependency that did not become a cyclic dependency conflict.

Example An example of a cyclic dispatch conflict is given in Listing 8.8. Figure 8.5 shows the corresponding dependency graph. In this example, concern `A` can dispatch method `m` to concern `B`. Concern `B` can dispatch method `m` to concern `A` again. This leads to a cyclic dispatch. If condition `c` remains `true`, an infinite loop is formed.

```
concern A {
  filtermodule FMA {
    conditions
      c : condition()
    internals
      b : B
    inputfilters
      err  : Error = { true => <inner.*> };
      disp1 : Dispatch = { c => [*.m] b.m };
      disp2 : Dispatch = { <inner.*> inner.* }
  }
}

concern B {
  filtermodule FMB {
    internals
      a : A
    inputfilters
      disp : Dispatch = { true => [*.m] a.m }
  }
}
```

Listing 8.8: Example of a cyclic dispatch conflict

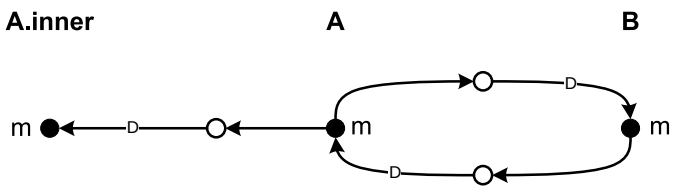


Figure 8.5: Example of a cyclic dispatch conflict.

Note that there is no cyclic dependency conflict in this example. The `Error` filter and the second `Dispatch` filter in concern `A` prevent this. The `Error` filter only lets methods in `inner` through. The second `Dispatch` filter and the condition in the first `Dispatch` filter make sure that methods that are in `inner` can always be dispatched.

Pattern The general pattern of a cyclic dispatch conflict in the dependency graph is a cycle with the following properties:

- All dependency edges in the cycle are dispatch dependency edges.
- All method nodes in the cycle are marked `Existing`.

8.1.5.3 Infinite Signature Conflict

The procedure in Section 8.1.3 might generate infinite signatures. This happens if an infinite number of methods have a dispatch dependency on just one single method that exists. Infinite signatures cannot be implemented in practice. Therefore, infinite signatures are a conflict.

Example An example of a concern with an infinite signature conflict is given in Listing 8.9. Figure 8.6 shows schematically the dependency graph. Every selector that can reach the `Dispatch` filter is dispatched to method `m` in `inner`. Because no other constraints are put on the set of selectors that can reach the `Dispatch` filter, the set of selectors contains all possible selectors. Therefore, the signature becomes infinite.

```
concern A {
  filtermodule FMA {
    inputfilters
      disp : Dispatch = { True => [*.*] inner.m }
  }
}
```

Listing 8.9: Example of an infinite signature conflict

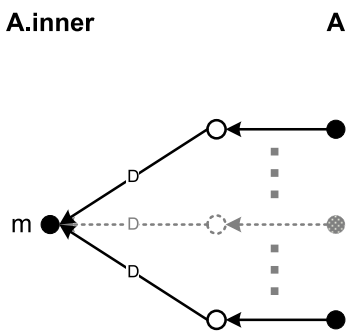


Figure 8.6: Example of an infinite signature conflict.

Pattern An infinite signature conflict has occurred if the number of method nodes marked `Existing` in the dependency graph is infinite.

8.1.5.4 Type Errors

A type error occurs if an existing method can be dispatched to a non-existent target method. The target method might be completely non-existent or there might be a method with the same name but with the wrong parameter types or return type. The entrance method exists because it can also be dispatched to an existing target method.

Example Listing 8.10 gives an example of a concern with a type error in the filter set. Actually, this example has both kinds of type errors: a dispatch to a non-existent method and a dispatch to a method with the wrong parameter types or return type. In this example, method `m` can be dispatched to the method `n` in `inner` with the wrong

parameter types. Method `m` can also be dispatched to the non-existent method `o` in `inner`. Figure 8.7 shows the dependency graph. Note that there are actually three type errors:

- Method `m()` can be dispatched to method `n` in `inner` with the wrong parameter types.
- Method `m()` can be dispatched to the non-existent method `o` in `inner`.
- Method `m(int)` can be dispatched to the non-existent method `o` in `inner`.

There is no type error 'Method `m(int)` can be dispatched to method `m` in `inner` with the wrong parameter types', because the signature dependency is also violated; there is a signature match that prevents this dispatch.

```
concern A {
  filtermodule FMA {
    conditions
      c1 : Condition1();
      c2 : Condition2();
    inputfilters
      disp1 : Dispatch = { c1 => [*.m] inner.n };
      disp2 : Dispatch = { c2 => [*.m] inner.o };
      disp3 : Dispatch = { true => <inner.*> inner.*}
  }

implementation in Java {
  public class A {
    public void m() {}
    public void n(int) {}
  }
}
}
```

Listing 8.10: Example of a concern with type errors

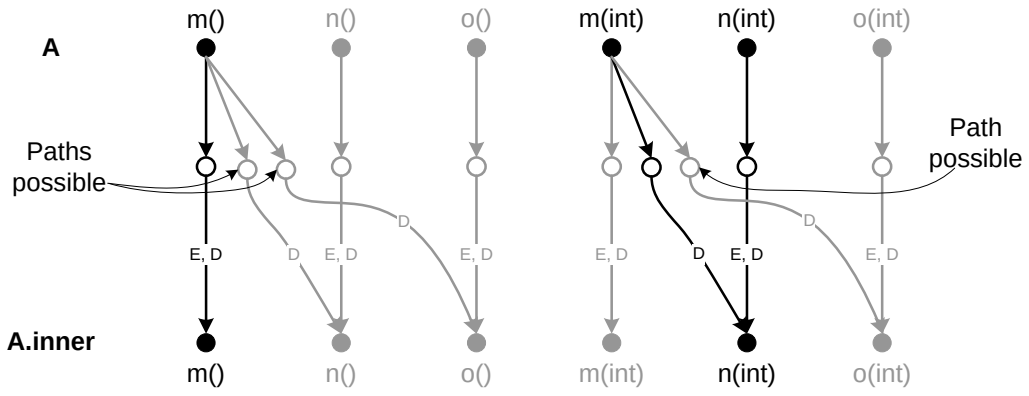


Figure 8.7: Example of a dependency graph with a type error.

Pattern The pattern of a type error in the dependency graph is a method node marked `Existing` that has a violated path node with the following properties:

- All signature matching dependencies are resolved.
- The dispatch dependency is violated.

Because all signature matching dependencies are resolved, no signature matching prevents the path. Therefore, the `Dispatch` action can be reached. This `Dispatch` action cannot be executed, because the dispatch method does not exist.

8.1.6 Using Other Filter Actions as Sufficient Requirement for Inclusion in the Signature

The signature generation algorithm uses the reachability of a `Dispatch` action as the necessary requirement for a method to be in the signature. But the `Dispatch` action does not have to be the only filter action that acts as a requirement for the inclusion of a method in the signature. The old approach also uses the `Meta` action, for example.

Suggestions for other actions that can be used are:

- The `Meta` action.
- The `Before` action.
- Filter actions that return the flow, such as the `Skip` action

If other filter actions are used, the signature generation procedure works as follows:

- Path nodes to other filter actions do not have a dispatch dependency. A path node is marked `Resolved` if all its signature matching dependencies are resolved.
- A path node is marked `Violated` if one of its signature matching dependencies is violated.
- A path node maintains the marking `Unknown` if none of its signature matching dependencies are violated and at least one of its signature matching dependencies is unknown.

A problem with using other filter actions is that they do not restrict the name and typing of the methods. Therefore, the name and typing need to be restricted in the filter set by selector matching and signature matching parts. Otherwise, infinite signatures are generated.

8.2 From Theory to Practice

The Theory and Concepts section introduced the concept dependency graph to represent the signature dependencies. It also presented algorithms to generate the signatures, to detect conflicts and to do type checking. A problem with the dependency graph is, however, that it is an infinite model. Therefore, the algorithms cannot be used in practice. This section presents a finite algorithm to generate the signatures, based on the dependency graph concept. This algorithm tries to build the dependency graph incrementally from the ground up. It thereby limits itself to a finite part of the dependency graph that is enough to generate the signatures and do the type checking and conflict detection.

8.2.1 Incrementally Building a Sufficient Subset of the Dependency Graph

If there are no conflicts, the signatures of the concerns are finite. This means that the part of the dependency graph that contains all existing methods and their dependencies is also finite. This part of the dependency graph is enough to generate the signatures.

The finite algorithm will incrementally build this part of the dependency graph, thereby incrementally building the signatures. It starts with the initialization step; all methods that do not have dependencies, methods in `inner` and methods in concerns without a filter set superimposed, exist by definition.

The next step, the start signatures step, is checking whether there are non-existent methods that have dependencies on the already existing methods. This is done by analyzing the *dispatch structure* of the filter set. The dispatch structure is a combination of the following information:

- Reached `Dispatch` actions.
- Distinguishable or undistinguishable entrance selector.
- The dispatch target and the dispatch selector.
- Signature matching parts between the entrance of the filter set and the dispatch action.

The filter reasoning models are used to obtain this information.

If a `Dispatch` state can be reached in the execution model and the target method of the dispatch exists, we can reason backward to which methods can reach this `Dispatch` state. These methods can be added to the signature of the concern. This increases the size of the finite dependency graph, making it possible for dependencies of other not existing methods to be resolved.

Not all dependencies can be checked immediately. Not-exist signature matching dependencies cannot be checked when building the dependency graph, because we cannot know for sure which methods do absolutely not exist. Therefore, during the start signatures step only accepting signature matches are taken into account. In this way, a superset of the final signatures is generated. The final signatures step narrows the results from the start signatures step down to the final signatures, by doing all signature matching checks.

After signature generation has finished, there is a checking step to detect conflicts and type errors.

8.2.2 Initialization

The first step in the signature generation algorithm is the initialization step. In this step the signature of all concerns that do not have a filter set superimposed is set to the implementation signature. For concerns that do have a filter set superimposed, the signature of the `inner` object is set to the implementation signature. All methods in these signatures are marked `Existing`, because they exist by definition. The default status of newly added methods is `Unknown`. Procedure `InitializeSignatures` shows the initialization part of the signature generation algorithm.

Procedure `InitializeSignatures`

```
1  foreach c ∈ concerns do
2      if c.superimposition = null then
3          signature[c] ← {m|m ∈ implementation[c]}
4          foreach m ∈ signature[c] do
5              m.status ← Existing
6          end
7      end
8      else
9          signature[c.inner] ← {m|m ∈ implementation[c]}
10         foreach m ∈ signature[c.inner] do
11             m.status ← Existing
12         end
13     end
14 end
```

8.2.3 Start Signatures

The start signatures step generates a superset of the final signatures. In this step, signature matching is only included if it finitely bounds the number of possible methods to add.

The start signatures step analyzes the dispatch structure: by combining the information of the entrance selector, the dispatch target, the dispatch selector and whether there has been signature matching, we can extract which methods can be dispatched. We will first introduce the concepts signature set and type set. This is followed by Table 8.1, showing the different classes of dispatch structures and the methods that should be added to the signature for each class. Each class of dispatch structures is explained. Finally, the start signatures procedure is given.

8.2.3.1 Signature Set and Type Set

The paths to a `Dispatch` state may go through signature matching parts. Signature matching parts bound the methods that can pass. Accepting signature matching parts bound the methods to a finite set: the signature of the matching target. Rejecting signature matching parts bound the methods to an infinite set: the complement of the signature of the matching target. Because we want to generate a finite superset of the signatures in the start signatures step, we only look at accepting signature matching parts.

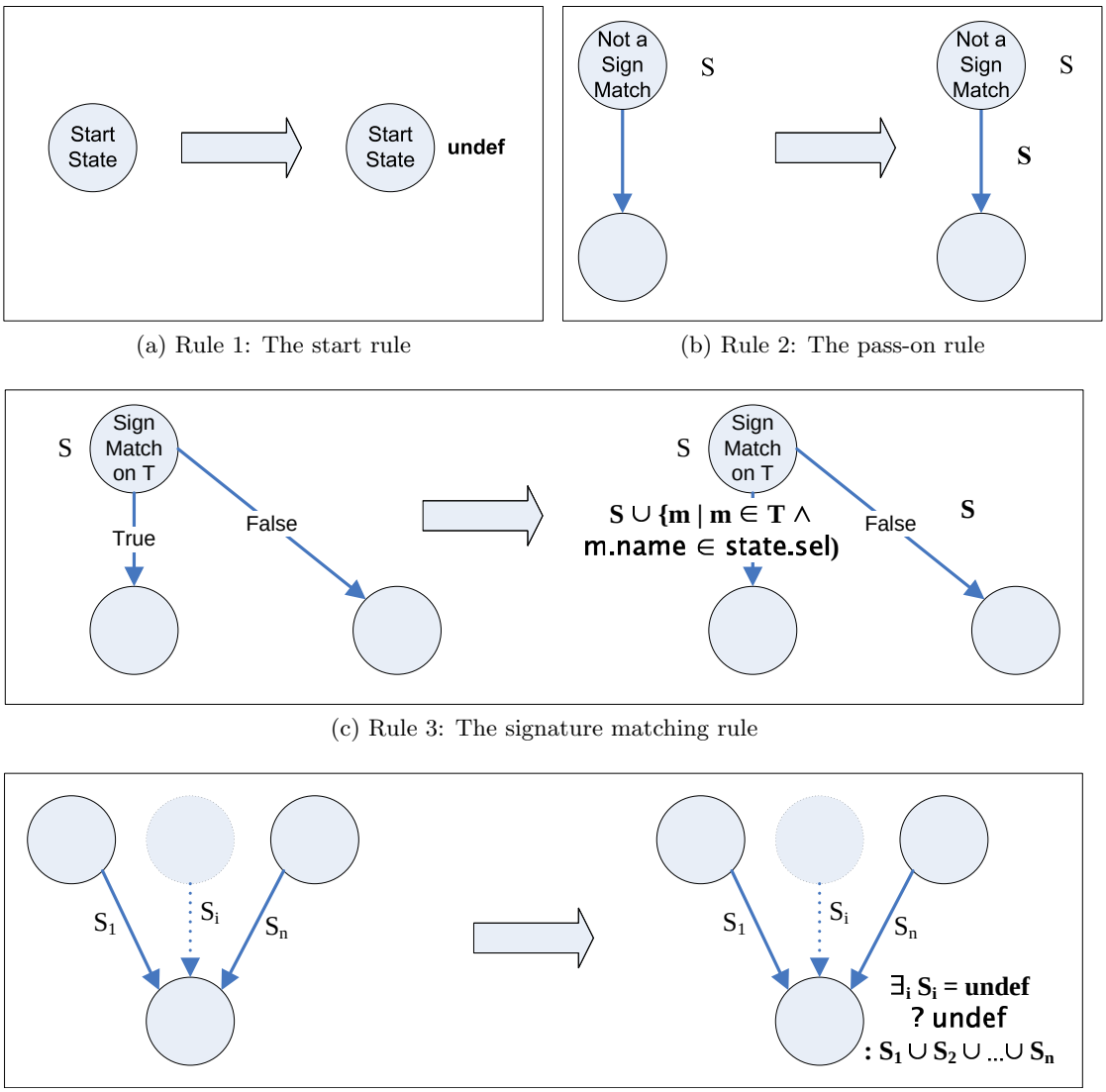
From the accepting signature matching parts we create two related sets: the *signature set* and the *type set*. These sets bound the methods that can reach the `Dispatch` state. Therefore, they are used during the generation of the set of methods that can reach the `Dispatch` state. For the following explanation of the signature set and the type set it is useful to know that the entrance selectors for which the dispatch structure is analyzed are all distinguishable selectors and the undistinguishable placeholder (refer back to Section 5.3 for more information about distinguishable and undistinguishable selectors).

The signature set gives both name and type information about the entrance methods. This is needed if the entrance selector is the undistinguishable placeholder. In this case, the names of the methods are not known. The signature set is used to get the names and types of the methods. To create the signature set, `SignatureMatchingPart` states are used at which the selector of the message is the undistinguishable placeholder. This means that the name of the selector is still the name of the method. The methods that can accept in this signature matching part must be in the signature of the matching target.

The type set gives only type information. It does not give name information, because the entrance selector has been substituted. So the name of the entrance methods cannot be deduced from the names of the methods in the type set. The type set is created from `SignatureMatchingPart` states at which the selector of the message is not the undistinguishable placeholder.

Signature sets and type sets are generated in a similar way. The difference is that for signature sets only the signature matching parts are used at which the selector of the message is the undistinguishable placeholder. For type sets only the signature matching parts are used at which the selector of the message is not the undistinguishable placeholder. Figure 8.8 shows the rules to create the signature/type set for each part of the state space.

- *Start rule*: Each possible message can reach the start state. Therefore, the signature/type set of the start state is initialized to `undefined`.
- *Pass-on rule*: If a state is not a `SignatureMatching` state, then the signature/type set of all outgoing transitions is the signature/type set of the state.
- *Signature matching rule*. If a signature matching part is encountered that corresponds to the kind of set generated, then the set of messages that can match is bounded by the signature of the matching target. Therefore, the methods in this target are part of the signature/type set if the signature matching part accepts. Only the methods that have the same name as the selector are added; if the selector



(a) Rule 1: The start rule

(b) Rule 2: The pass-on rule

(c) Rule 3: The signature matching rule

(d) Rule 4: The combination rule

Figure 8.8: The rules to create the signature/type set

is the undistinguishable placeholder, all methods with an undistinguishable name are added. If the signature matching part rejects, nothing is added to the signature/type set, because the complement of the signature of the matching target is not finite.

We take here the union and not the intersection of `S` and the appropriate methods from `T`. One might expect the intersection, because only messages that are also in the matching target can cause the acceptance of the matching part. By taking the union, however, it is ensured that all methods that are part of a fully finite or type finite cyclic dependency conflict set are also put in the signature/type set. This makes it possible to detect these cyclic dependency conflicts. If we had taken the

intersection of the two sets, they would not have been added. The reason for this is that they are not present yet in one of the two sets, due to the cyclic dependency.

If `S` is `undefined`, the empty set is used in the union, instead of `S`.

- *Combination rule.* This rule calculates the signature/type set of a state based on its incoming transitions. If there is an incoming transition with an `undefined` signature/type set, than the signature/type set of the state is also `undefined`. The reason for this is that the set of messages/types that can reach the state is not finite. If no incoming transition has an `undefined` signature/type set, than the signature/type set of the state is the union of the signature/type sets of all incoming transitions.

8.2.3.2 Dispatch Classes

To generate the start signatures, we look at the dispatch structures in the filter set. There are different classes of dispatch structures. These classes differ in the entrance selector, in the dispatch selector and whether there is a signature/type set. For each class different methods need to be added to the signature. Table 8.1 shows the seven different classes of dispatch structures and the methods that need to be added to the signature. The following conventions are used in the table:

- p^m creates a new method with name `p` and types copied from method `m`.
- $m \in target$ is sometimes used as an abbreviation of $m \in signature[target]$.
- Probe methods: Each class adds certain probe methods to detect cyclic dependency conflicts. The question mark symbol (?) is used as a placeholder if a certain part of the probe methods is from an infinite set:
  - ?(?): Fully infinite cyclic dependency conflict set.
  - $?^m$: Type finite cyclic dependency conflict set.
  - $sel(?)$: Name finite cyclic dependency conflict set.
  - Probe methods for cyclic dependency conflicts from a fully finite set are normal methods. They are added by taking the union of different methods sets, instead of the intersection.

Class 1

Class 1 is the first of two classes for selectors from the distinguishable set. The distinguishable set is initially the set of distinguishable selectors from the execution model. It can, however, be changed in the start signatures step, as will be explained later.

In this class the type set is `undefined`. There is never a signature set, because the entrance selector is not the undistinguishable placeholder. A distinguishable selector is always dispatched to some `t.s`, where `s` might be the same selector as the entrance selector.

The entrance selector `p` can only be dispatched as `t.s` if `s` exists in `t` and if the types are the same. Therefore, we add `p` to the signature with all types `s` has in `t`.

Class	Entr. msg	Signature/Type set	Disp. msg	Add to signature
1	$p \in distSet$	Type set = `undef`	$t.s$	$\{p^m \mid m \in t \wedge m.name = s\}$ $\cup\{p(?)\}$ $t.distSet = t.distSet \cup \{s\}$
2	$p \in distSet$	Type set = T	$t.s$	$\{p^m \mid m \in t \wedge m.name = s\}$ $\cup\{p^m \mid m \in T\}$ $\cup\{p(?)\}$ $t.distSet = t.distSet \cup \{s\}$
3	$*$	Sig set = `undef`	$t.*$	$\{m \mid m \in t \wedge m.name \in undist.\}$ $\cup\{?(?)\}$
4	$*$	Sig set = S	$t.*$	$\{m \mid m \in t \wedge m.name \in undist.\}$ $\cup S$ $\cup\{?(?)\}$
5	$*$	Sig set = `undef`	$t.s$	Infinite Signature Conflict
6	$*$	Sig set = S Type set = `undef`	$t.s$	S $\cup\{?^m \mid m \in t \wedge m.name = s\}$ $\cup\{?(?)\}$ $t.distSet = t.distSet \cup \{s\}$
7	$*$	Sig set = S Type set = T	$t.s$	S $\cup\{?^m \mid m \in t \wedge m.name = s\}$ $\cup\{?^m \mid m \in T\}$ $\cup\{?(?)\}$ $t.distSet = t.distSet \cup \{s\}$

Table 8.1: Different classes of dispatch structures for signature generation

The following probe methods are also added to check for cyclic dependency conflicts:

- Infinite: The entrance selector is always `p`, so no infinite cyclic dependency conflict set can be caused by this structure.
- Name finite: The name is bounded to selector `p`. So, method `p(?)` is added to check for this type of cyclic dependency conflict set. The question mark is used as a placeholder type, meaning a type different from normally existing types. This ensures that the type of the probe method is in the infinite type set.
- Type finite: The entrance selector is always `p`, so no type finite cyclic dependency conflict set can be caused by this structure.
- Fully finite: Only methods with name `s` in target `t` can finitely bound the set of entrance methods. So the probe methods are methods named `p` with all types selector `s` has in target `t`. These methods are already added.

Adding `s` to the distinguishable set of `t`? We also add selector `s` to the distinguishable set of target `t`, if `t` is a concern that has a filter set superimposed. In this way, selector `s` is always present as an explicit dispatch method in target `t`. This prevents problems with checking for the name finite cyclic dependency conflict set. Listing 8.11 shows an example to explain this. This example has two concerns: concern `A` and concern `B`. Both concerns do not have methods in there inner implementation.

Suppose that the selector is not added to the distinguishable set of the target. In the first iteration of `StartSignatures`, concern `A` gets signature $\{p(?)\}$ (class 1) and concern `B` gets signature $\{?(?)\}$ (class 6). Nothing changes anymore in the second iteration.

The final signatures step, which is explained later, checks whether the methods in the start signature can really be dispatched to an existing target method. This step concludes that method `p(?)` in concern `A` cannot be dispatched, because method `s(?)` does not exist in concern `B`. This also leads to the exclusion of `?(?)` from concern `B`. So, the signatures of both concerns will be empty and the cyclic dependency conflict is not detected.

The problem in this example is that `s(?)` is part of the set for which `?(?)` is the placeholder. But because the algorithm does strict name checking, it concludes that `s(?)` is not in the signature of `B`. To prevent this, the dispatch selector `s` is explicitly added to the distinguishable set of concern `B`. This leads to `s(?)` being explicitly added to the signature of `B` (class 2), which leads to the detection of the cyclic dependency conflict.

```
concern A {
  filtermodule FMA {
    internals
      b : B
    inputfilters
      disp1 : Dispatch = { true => [*.p] b.s};
  }
}

concern B {
  filtermodule FMB {
    internals
      a : A
    inputfilters
      error : Error    = { true => <self.*> };
      disp2 : Dispatch = { true => [*.*] a.p}
  }
}
```

Listing 8.11: Example to illustrate why a selector is added to the distinguishable set of the target concern

Class 2

Class 2 is the second class for distinguishable selectors. In this class, the type set is defined as `T`. There is never a signature set, because the entrance selector is not

the undistinguishable placeholder. Just as in class 1, we add `p` with all types `s` has in `t`.

The following probe methods are also added to check for cyclic dependency conflicts:

- Infinite: The entrance selector is always `p`, so no infinite cyclic dependency conflict set can be caused by this structure.
- Name finite: The name is bounded to selector `p`, so method `p(?)` is added to check for this type of cyclic dependency conflict set
- Type finite: The entrance selector is always `p`, so no type finite cyclic dependency conflict set can be caused by this structure.
- Fully finite: The type checking in the type set `T` can put a type bound on a cyclic dependency conflict. Because the name is always bounded to selector `p`, we add `p` with all types from `T`. Also all methods named `s` in target `t` can finitely bound the methods. These probe methods are already added.

We also add `s` to the distinguishable set of `t`, for the same reason as in class 1.

Class 3

Class 3 is the first of the five classes for the undistinguishable entrance selectors. The signature generation differs for undistinguishable entrance selectors depending on whether there is a signature set, whether there is a type set and whether the dispatch selector is different (due to substitution).

Class 3 represent the dispatch structures where there is no signature set and where the dispatch selector is the same as the entrance selector. In this case, there is also never a type set, because the undistinguishable placeholder is never substituted.

Methods are in the signature if they can be dispatched to target `t`. Therefore, all methods from `t` that have an undistinguishable name are added to the signature.

The following probe methods are also added to check for cyclic dependency conflicts:

- Infinite: The probe method `?(?)` is added.
- Name finite: In this structure the name is not finitely bounded, so no probe methods are added for this type of cyclic dependency conflict set.
- Type finite: In this structure the type is not finitely bounded, so no probe methods are added for this type of cyclic dependency conflict set.
- Fully finite: Only the methods in the dispatch target `t` can finitely bound the cyclic dependency conflict set. These methods are already added.

Class 4

Class 4 represents the dispatch structures where there is an undistinguishable entrance selector, there is a signature set and the dispatch selector is the same as the entrance selector. In this case, there is never a type set, because the undistinguishable placeholder is never substituted.

Just as with class 3, methods are in the signature if they can be dispatched to the target `t`. Therefore, all methods from `t` that have an undistinguishable name are added to the signature.

The following probe methods are also added to check for cyclic dependency conflicts:

- Infinite: The probe method `?(?)` is added.
- Name finite: In this structure the name is not finitely bounded, so no probe methods are added for this type of cyclic dependency conflict set.
- Type finite: In this structure the type is not finitely bounded, so no probe methods are added for this type of cyclic dependency conflict set.
- Fully finite: The methods in the signature set S can put a finite bound on a cyclic dependency conflict set. Therefore, all methods from S are added as probe methods. Also the methods from the dispatch target `t` can put a finite bound on a cyclic dependency conflict set. These methods are already added.

Class 5

Class 5 represents the dispatch structures where there is an undistinguishable entrance selector, there is no signature set and the entrance selector is substituted to a certain selector `s`. There might be a type set, but this is not important.

Because there is no signature set, an infinite number of undistinguishable entrance selectors can reach the state. All these entrance selectors are also substituted to selector `s`. If selector `s` exists in target `t`, all selectors from the infinite set can be dispatched. This leads to an infinite signature. Infinite signatures are not possible in practice. Therefore, nothing is added to the signature. Instead, an infinite signature conflict error is given.

Class 6

Class 6 represents the dispatch structures where there is an undistinguishable entrance selector, there is a signature set , there is no type set and the entrance selector is substituted to a certain selector `s`.

Only methods in signature set `S` can reach the `Dispatch` state. Signature sets are always finite. Therefore, we do not have the problem of an infinite signature conflict as we had in Class 5.

The only methods that can be dispatched are the methods from `S` for which there is a method named `s` in target `t` with the same typing. All methods from `S` are added, however. In this way all methods that are part of a fully finite cyclic dependency conflict set are also added.

The following probe methods are also added to check for cyclic dependency conflicts:

- Infinite: The probe method `?(?)` is added.
- Name finite: There can be a name finite cyclic dependency conflict set. The name is bounded by the dispatch selector `s`. The name of the cyclic entrance selector cannot be derived from this structure. This is no problem, because

selector `s` is added to the distinguishable set of target `t`. This causes method `s(?)` to be added to `t`. This will again cause the other probe methods to be added to the other concerns in the cycle, including this concern.

- Type finite: Although the entrance selector is always substituted to the single selector `s`, there can be cyclic dependency conflicts with names from an infinite set between this concern and the targets of the signature matching parts. In this case, the types will always be bounded by the types of methods named `s` in target `t`. Therefore, selector `?` is added with all types `s` has in `t`.
- Fully finite: The signature matching in the signature set `S` can put a finite bound on a cyclic dependency conflict set. Therefore, all methods from `S` are added.

We also add selector `s` to the distinguishable set of target `t`, for the same reason as in class 1.

Class 7

Class 7 represents the case where there is an undistinguishable entrance selector, there is a signature set, there is a type set and the entrance selector is substituted to a certain selector `s`.

Just as with class 6, the only methods that can be dispatched are the methods from `S` for which there is a method named `s` in target `t` with the same typing. Again, all methods from `S` are added to make the detection of fully finite cyclic dependency conflicts possible.

The following probe methods are also added to check for cyclic dependency conflicts:

- Infinite: The probe method `?(?)` is added.
- Name finite: Just as in class 6, there can be a name finite cyclic dependency conflict set in this class. This again involves the dispatch selector `s`. This selector is therefore added to the distinguishable set of the dispatch target.
- Type finite: Although the entrance selector is always substituted to the single selector `s`, there can be cyclic dependency conflicts with name infinite between this concern and the targets of the signature matching parts. Just as in class 6, the types can be bounded by the types selector `s` has in target `t`. Therefore, selector `?` is added with all types `s` has in `t`. In this class, the types can also be bounded by the type set. Therefore, selector `?` is also added with all types of the methods in the type set.
- Fully finite: The signature matching in the signature set `S` can put a finite bound on a cyclic dependency. Therefore, all methods from `S` are added.

We also add selector `s` to the distinguishable set of target `t`, for the same reason as in class 1.

8.2.3.3 The Start Signatures Procedure

Procedure `StartSignatures` shows the main procedure of the generation of the start signatures. While there is a change (new methods added), it iterates over all concerns and tries to extend the start signature of each concern by calling `StartSignature`.

Procedure `StartSignatures`

```
1  repeat
2  |  change ← false
3  |  foreach c ∈ concerns do
4  |  |  if c.superimposition ≠ null then
5  |  |  |  StartSignature(c)
6  |  |  end
7  |  end
8  until ¬change
```

Procedure `StartSignature(`*Concern c*`)`

```
1  StartSignatureDistinguishable(c)
2  StartSignatureUndistinguishable(c)
```

Procedure `StartSignature` does the calculation for one concern. It does this by calling `StartSignatureDistinguishable` and `StartSignatureUndistinguishable` .

Procedure `StartSignatureDistinguishable(`*Concern c*`)`

```
1  fireModel ← firemodels[c]
2  sig ← ∅
3  foreach p ∈ fireModel.distinguishableSelectors do
4  |  executionModel ← fireModel.getExecutionModel(p)
5  |  typeSet ← CreateTypeSet(executionModel)
6  |  foreach state ∈ dispatchStates(executionModel) do
7  |  |  if typeSet[state] = undefined then
8  |  |  |  sig ← sig ∪ StartSignatureClass1(p, state)
9  |  |  end
10 |  |  else
11 |  |  |  sig ← sig ∪ StartSignatureClass2(p, state, typeSet[state])
12 |  |  end
13 |  end
14 end
15 if sig ⊈ signature[c] then
16 |  signature[c] ← signature[c] ∪ sig
17 |  change ← true
18 end
```

Procedure `StartSignatureDistinguishable` does the generation of the start signatures for each distinguishable selector in the filter set. This represents class 1 and class 2 of Table 8.1. It iterates over all distinguishable selectors. For each selector it creates the execution model and iterates over the `Dispatch` states. For each `Dispatch` state it is checked whether there is a type set. If there is no type set, function `StartSignatureClass1` is called to generate the methods according to class 1. If there is a type set, function `StartSignatureClass2` is called to generate the methods according to class 2.

If there are methods in the generated set that are not contained in the signature of the concern, these methods are added to the signature and the `change` flag is set to `true`.

Function `StartSignatureClass1(`*Selector p, ExecutionState state*`)`

1 $sig \leftarrow \{p^m | m \in state.substitutionTarget \wedge m.name = state.substitutionSelector\}$
  `// For cyclic dependency conflict detection:`
2 $sig \leftarrow sig \cup \{p(?)\}$
3 **if** $state.substitutionSelector \notin distSet[state.substitutionTarget]$ **then**
4 $distSet[state.substitutionTarget] \leftarrow$
  $distSet[state.substitutionTarget] \cup \{state.substitutionSelector\}$
5 $change \leftarrow true$
6 **end**
7 **return** sig

Function `StartSignatureClass2(`*Selector p, ExecutionState state, Set typeSet*`)`

1 $sig \leftarrow \{p^m | m \in state.substitutionTarget \wedge m.name = state.substitutionSelector\}$
  `// For cyclic dependency conflict detection:`
2 $sig \leftarrow sig \cup \{p^m | m \in typeSet\}$
3 $sig \leftarrow sig \cup \{p(?)\}$
4 **if** $state.substitutionSelector \notin distSet[state.substitutionTarget]$ **then**
5 $distSet[state.substitutionTarget] \leftarrow$
  $distSet[state.substitutionTarget] \cup \{state.substitutionSelector\}$
6 $change \leftarrow true$
7 **end**
8 **return** sig

Procedure `StartSignatureUndistinguishable` does the generation for classes 3, 4, 5, 6 and 7 from Table 8.1. It iterates over all `Dispatch` states in the execution model of the undistinguishable placeholder. For each `Dispatch` state it is checked whether the dispatch selector is the same undistinguishable selector (classes 3 and 4) or has been substituted to another selector (classes 5, 6 and 7). Then it is checked whether there is a signature set, to distinguish between class 3 and class 4 or between class 5 and classes 6 and 7. Classes 6 is distinguished from class 7 by checking whether there is a type set. When the right class is found, the corresponding function is called to generate the methods according to Table 8.1.

Just as with the distinguishable selectors, after the method set has been calculated it is checked whether there are methods not already in the signature of the concern. If

this is the case, these methods are added and the `change` flag is set to `true`.

Procedure `StartSignatureUndistinguishable(`*Concern c*`)`

```
1  fireModel ← firemodels[c]
2  executionModel ←
   fireModel.getExecutionModel(UNDISTINGUISHABLE_SELECTOR)
3  signatureSet ← CreateSignatureSet(executionModel)
4  typeSet ← CreateTypeSet(executionModel)
5  sig ← ∅
6  foreach state ∈ dispatchStates(executionModel) do
7      if state.substitutionSelector = UNDISTINGUISHABLE_SELECTOR
       then
8          if signatureSet[state] = undefined then
9              sig ← sig ∪ StartSignatureClass3(state)
10         end
11         else
12             sig ← sig ∪ StartSignatureClass4(state, signatureSet[state])
13         end
14     end
15     else
16         if signatureSet[state] = undefined then
17             Infinite Signature Error
18         end
19         else
20             if typeSet[state] = undefined then
21                 sig ← sig ∪ StartSignatureClass6(state, signatureSet[state])
22             end
23             else
24                 sig ← sig ∪
                   StartSignatureClass7(state, signatureSet[state], typeSet[state])
25             end
26         end
27     end
28 end
29 if sig ⊈ signature[c] then
30     signature[c] ← signature[c] ∪ sig
31     change ← true
32 end
```

Function `StartSignatureClass3(`*ExecutionState state*`)`

1 $sig \leftarrow \{m | m \in state.substitutionTarget \wedge m.name \in undistinguishable\}$
`// For cyclic dependency conflict detection:`
2 $sig \leftarrow sig \cup \{?(?)\}$
3 **return** sig

Function `StartSignatureClass4(`*ExecutionState state, Set signatureSet*`)`

1 $sig \leftarrow \{m | m \in state.substitutionTarget \wedge m.name \in undistinguishable\}$
`// For cyclic dependency conflict detection:`
2 $sig \leftarrow sig \cup signatureSet$
3 $sig \leftarrow sig \cup \{?(?)\}$
4 **return** sig

Function `StartSignatureClass6(`*ExecutionState state, Set signatureSet*`)`

1 $sig \leftarrow signatureSet$
`// For cyclic dependency conflict detection:`
2 $sig \leftarrow sig \cup \{?^m | m \in state.substitutionTarget \wedge m.name = state.substitutionSelector\}$
3 $sig \leftarrow sig \cup \{?(?)\}$
4 **if** $state.substitutionSelector \notin distSet[state.substitutionTarget]$ **then**
5 $\quad distSet[state.substitutionTarget] \leftarrow distSet[state.substitutionTarget] \cup \{state.substitutionSelector\}$
6 $\quad change \leftarrow true$
7 **end**
8 **return** sig

Function `StartSignatureClass7(`*ExecutionState state, Set signatureSet, Set typeSet*`)`

1 $sig \leftarrow signatureSet$
`// For cyclic dependency conflict detection:`
2 $sig \leftarrow sig \cup \{?^m | m \in state.substitutionTarget \wedge m.name = state.substitutionSelector\}$
3 $sig \leftarrow sig \cup \{?^m | m \in typeSet\}$
4 $sig \leftarrow sig \cup \{?(?)\}$
5 **if** $state.substitutionSelector \notin distSet[state.substitutionTarget]$ **then**
6 $\quad distSet[state.substitutionTarget] \leftarrow distSet[state.substitutionTarget] \cup \{state.substitutionSelector\}$
7 $\quad change \leftarrow true$
8 **end**
9 **return** sig

8.2.4 Final Signatures

The previous step generated a superset of the final signatures. The next step is decreasing this superset to the final signatures.

Procedure `FinalSignatures`

```
1 repeat
2   change ← false
3   foreach c ∈ concerns do
4     if c.superimposition ≠ null then
5       finalSignature(c)
6     end
7   end
8 until ¬change
```

Procedure `FinalSignatures` is the main procedure to generate the final signatures. It iterates over all concerns and calls for each concern the procedure `FinalSignature` to generate the final signature for that concern. It does this iteratively until there are no more changes.

Procedure `FinalSignature(`*Concern c*`)`

```
1 foreach m ∈ signature[c] do
2   if m.status = UNKNOWN then
3     checkDispatchable(m, c)
4   end
5 end
```

The procedure `FinalSignature` does the final signature generation for one concern. It iterates over all methods with the status `Unknown` and checks whether the method is dispatchable by calling the procedure `CheckDispatchable`.

A method has to be in the signature of a concern if it can be dispatched. Procedure `CheckDispatchable` checks for a certain method whether it can be dispatched. The dispatch check is done by searching the execution model for `Dispatch` states. Signature checks in the signature matching parts are enabled in this execution model. In this way, the signature matching dependencies are resolved.

The procedure first does the search on the execution model with *strict signature checking*. Strict signature checking means the following:

- A signature matching part only accepts if the matched method has the status `Existing` in the signature of the matching target.
- A signature matching part only rejects if the matched method has the status `NotExisting` in the signature of the matching target.
- A signature matching part does not accept and reject if the matched method has the status `Unknown` in the signature of the matching target. This means that this

Procedure `checkDispatchable(`*Method m, Concern c*`)`

```
1  fireModel ← firemodels[c]
2  executionModel ←
   fireModel.getExecutionModel(m, STRICT_SIGNATURE_CHECK)
3  foreach state ∈ dispatchStates(executionModel) do
4  |  if ∃_n ∈ state.substTarget : n = state.substSelector^m ∧ n.status = Existing
   |  then
5  |  |  m.status ← Existing
6  |  |  change ← true
7  |  |  return
8  |  end
9  end
10 executionModel ←
   fireModel.getExecutionModel(m, LOOSE_SIGNATURE_CHECK)
11 foreach state ∈ dispatchStates(executionModel) do
12 |  if
   |  ∃_n ∈ state.substTarget : n = state.substSelector^m ∧ n.status ≠ NotExisting
   |  then
13 |  |  return
14 |  end
15 end
16 m.status ← NotExisting
```

signature matching part is a dead end in the execution model. It is not sure yet which option is the right one. Therefore, none of the options is taken, to be sure that no wrong path is visited.

- A dispatch is only valid if the dispatch method has the status `Existing` in the signature of the dispatch target.

If in this execution model a valid `Dispatch` state can be reached, it is certain that the method can be dispatched. Therefore, the status of the method is set to `Existing`.

If a valid `Dispatch` state cannot be reached in this execution model, the method does not have to be `NotExisting`. There might be other `Unknown` methods that can make the method dispatch if they are `Existing` or `NotExisting`. Therefore, if the first part of the procedure did not set the status of the method to `Existing`, the second part of the procedure tries to find a valid `Dispatch` state with *loose signature checking*. Loose signature checking means the following:

- A signature matching part only accepts if the target method has the status `Existing` in the signature of the matching target.
- A signature matching part only rejects if the target method has the status `NotExisting` in the signature of the matching target.
- A signature matching part accepts as well as rejects if the target method has the

status `Unknown` in the signature of the matching target. This means that both the accepting path and the rejecting path can be taken in the execution model.

- A dispatch is valid if the dispatch method has the status `Existing` or the status `Unknown` in the signature of the dispatch target.

If a valid `Dispatch` state can be reached in the execution model with loose signature checking, it is still possible that the method can be dispatched. Therefore, the `Unknown` status of the method is maintained. If no valid `Dispatch` state can be reached, it is sure that the method cannot be dispatched. Therefore, the method gets the status `NotExisting`.

8.2.5 Type Checking and Conflict Detection

After the generation of the final signatures, the last step involves type checking and conflict detection. In this step, the cyclic dependency conflicts and the cyclic dispatch conflicts will be detected and the type compatibility between an entrance method and the dispatch method is checked. Detecting infinite signature conflicts has already been done in the start signature generation step.

Procedure `Checking` is the main procedure of the checking step. It calls all specific checking procedures. The cyclic dependency conflict check is called first, because if there are cyclic dependency conflicts, there are `Unknown` methods left. These `Unknown` methods might lead to unnecessary type errors and cyclic dispatch conflicts.

Procedure `Checking`

```
1 CyclicDependencyConflictCheck()
2 CyclicDispatchCheck()
3 TypeCheck()
```

8.2.5.1 Cyclic Dependency Conflicts

Procedure `CyclicDependencyConflictCheck`

```
1 foreach c ∈ concerns do
2     foreach m ∈ signature[c] do
3         if m.status = Unknown then
4             Cyclic dependency conflict
5         end
6     end
7 end
```

The procedure `CyclicDependencyConflictCheck` detects cyclic dependency conflicts. This is done by finding a method that still has the status `Unknown`. The existence of such a method indicates a cyclic dependency conflict, as was explained in Section 8.1.5. All

four types of cyclic dependency conflict sets are being detected, due to the adding of certain probe methods in the start signatures step.

If there is no cyclic dependency conflict, all probe methods of the three infinite types of cyclic dependency conflict sets will be marked `NotExisting`, because they can never be dispatched. The probe methods of the fully finite type might be marked `Existing`; they are marked `Existing` if they can be dispatched and they are not part of a cyclic dependency conflict.

8.2.5.2 Type Checking

Procedure `TypeChecking`

```
1  foreach c ∈ concerns do
2      if c.superimposition ≠ null then
3          fireModel ← fireModels[c]
4          foreach m ∈ signature[c] do
5              executionModel ← fireModel.getExecutionModel(m)
6              foreach state ∈ dispatchStates(executionModel) do
7                  if state.substitutionSelector^m ∉ state.substitutionTarget then
8                      if
                       ∃_{n∈state.substitutionTarget} : n.name = state.substitutionSelector
                       then
9                          Type Error: Wrong parameter types or return type
10                     end
11                     else
12                         Type Error: Non-existent dispatch method
13                     end
14                 end
15             end
16         end
17     end
18 end
```

The procedure `TypeChecking` checks for every method in every concern whether its dispatch methods exist. If a dispatch method does not exist, there are two possibilities:

- There is another method with the same name but different types. This is a type incompatibility error.
- There is no such other method. This is a non-existent dispatch method error.

Also, the following, more informative, type checking can be done (not shown in the procedure):

- Find the reasons of the removal of implementation methods. There can be two reasons: no dispatch action can be reached or the target methods of all reached dispatch actions do not exist. The information given by this type checking can be

marked informative, because it might actually be the intention of the filter set to exclude certain methods.

- Check for all distinguishable selectors that do not have a method with the same name in the signature why this is the case. Again the two reasons are: no dispatch action can be reached or the target methods of all reached dispatch actions do not exist. The information given by this type checking can be marked as a warning, because this might indicate an error in the filter set. There is a reason for the selector to be distinguishable in the filter set. It should not be an error, because it might also be the intention of the filter set to exclude the selector.

8.2.5.3 Cyclic Dispatch Conflicts

The procedure `CyclicDispatchConflictCheck` detects the cyclic dispatch conflicts. Because of the complexity of this checking, the procedure is divided into three subprocedures.

Procedure `CyclicDispatchConflictCheck`

1 $CyclicDispatchConflictCheckInit()$
2 $CyclicDispatchConflictCheckProcess()$
3 $CyclicDispatchConflictCheckFinal()$

The first subprocedure, `CyclicDispatchConflictCheckInit`, does the initialization. All methods that are in a concern that has a filter set superimposed get the cyclic dispatch status `true`. These are the only methods that can form a cyclic dispatch. All other methods have the default value `false`.

Procedure `CyclicDispatchConflictCheckInit`

```
1 foreach c ∈ concerns do
2     if c.superimposition ≠ null then
3         foreach m ∈ signature[c] do
4             m.cyclicDispatch ← true
5         end
6     end
7 end
```

The subprocedure `CyclicDispatchConflictCheckProcess` does the processing. It iteratively does the following:

- If a method can be dispatched to a method with `cyclicDispatch` status set to `true`, the `cyclicDispatch` status of the method is also set to `true`.
- If all dispatches of a method are to methods with `cyclicDispatch` status set to `false`, the `cyclicDispatch` status of the method is also set to `false`.

This is done iteratively, until there are no more changes.

Procedure `CyclicDispatchConflictCheckProcess`

```
1  repeat
2      change ← false
3      foreach c ∈ concerns do
4          if c.superimposition ≠ null then
5              fireModel ← firemodels[c]
6              foreach m ∈ signature[c] do
7                  cyclDisp ← false
8                  execModel ← fireModel.getExecutionModel(m)
9                  foreach state ∈ dispatchStates(execModel) do
10                     if ∃_n ∈ state.substitutionTarget : (n =
                       state.substitutionSelector^m ∧ n.cyclicDispatch then
11                         cyclDisp ← true
12                     end
13                 end
14                 if m.cyclicDispatch ∧ ¬cyclDisp ∨ ¬m.cyclicDispatch ∧ cyclDisp
                   then
15                     change ← true
16                     m.cyclicDispatch ← cyclDisp
17                 end
18             end
19         end
20     end
21 until ¬change
```

The subprocedure `CyclicDispatchConflictCheckFinal` checks whether there are still methods left with `cyclicDispatch` status set to `true`. If this is the case, there is a cyclic dispatch conflict.

If there is a method left with `cyclicDispatch` status set to `true`, there always is a cyclic dispatch conflict. If there was no such conflict, then all dispatch sequences would eventually lead to a leaf method, a method not part of a superimposed concern. This method has `cyclicDispatch` status `false`. Then all methods in the sequences would also get the `cyclicDispatch` status `false`. Eventually, the original method gets the `cyclicDispatch` status `false`. So, if there are methods with `cyclicDispatch` status set to `true`, there is a cyclic dispatch conflict.

If there are no methods with `cyclicDispatch` status set to `true`, there also is no cyclic dispatch conflict. If there was a cyclic dispatch conflict, the cyclic dispatch would cause each method in the cycle to keep having the `cyclicDispatch` status `true`. Each method keeps having the `cyclicDispatch` status `true`, because one of its dispatch methods keep having the `cyclicDispatch` status `true`. So, if there are no methods with `cyclicDispatch` status set to `true`, there is no cyclic dispatch conflict.

So, a method with `cyclicDispatch` status set to `true` is a necessary and sufficient requirement for a cyclic dispatch conflict.

Procedure `CyclicDispatchConflictCheckFinal`

```
1 foreach c ∈ concerns do
2     foreach m ∈ signature[c] do
3         if m.cyclicDispatch then
4             Cyclic dispatch conflict.
5         end
6     end
7 end
```

8.2.6 Using Other Filter Actions as Sufficient Requirement for Inclusion in the Signature

As was explained in Section 8.1.6, other actions than the `Dispatch` action can be used as a requirement for the existence of a method. This section explains how these other actions can be incorporated in the finite algorithm.

A problem with using other actions is that they do not indicate typing. The `Dispatch` action indicates typing through the types of the dispatch selector in the dispatch target. Other actions, such as the `Meta` action, do not indicate typing. Therefore, if a certain selector can reach such an action in the filter set, the typing needs to be finitely constrained by some earlier accepting signature matching part. Otherwise, an infinite signature is created, because the selector can reach the action with infinitely many types.

Table 8.2 shows the different classes of structures in the filter set for other actions than the `Dispatch` action.

Class	Entr. msg	Signature/Type Set	Disp. msg	Add to signature
8	$p \in distSet$	Type set = `undef`	Other action	Infinite Signature Conflict
9	$p \in distSet$	Type set = T	Other action	$\{p^m \mid m \in T\}$ $\cup\{p(?)\}$
10	$*$	Sig set = `undef`	Other action	Infinite Signature Conflict
11	$*$	Sig set = S Type set = `undef`	Other action	S $\cup ?(?)$
12	$*$	Sig set = S Type set = T	Other action	S $\cup\{?^m \mid m \in T\}$ $\cup ?(?)$

Table 8.2: Classes of filter set structures with other actions than the `Dispatch` action.

Class 8

Class 8 is the first of two classes for the distinguishable selectors. In class 8 there is no type set. Therefore, any method named `p` can reach the action. Because there

are an infinite number of methods named `p` possible, an infinite signature conflict is caused.

Class 9

Class 9 is the second class for the distinguishable selectors. In class 9 there is a type set. Therefore, the set of methods named `p` that can reach the action is finitely bounded by the type set; only methods named `p` that have the same type as a method in the type set can reach the action. Therefore, selector `p` is added to the signature with all types from the type set.

The following probe methods are also added to check for cyclic dependency conflicts:

- Infinite: The entrance selector is always `p`, so no infinite cyclic dependency conflict set can be caused by this structure.
- Name finite: The name is bounded to selector `p`, so method `p(?)` is added to check for this type of cyclic dependency conflict set.
- Type finite: The entrance selector is always `p`, so no type finite cyclic dependency conflict set can be caused by this structure.
- Fully finite: Only the types from the type set can finitely bound a cyclic dependency conflict. The probe methods are already added.

Class 10

Class 10 is the first of three classes for undistinguishable selectors. In class 10 there is no signature set. There might be a type set, but this is not relevant. Because there is no signature set, there is no finite bound on the name of the methods that can reach the action. This causes an infinite signature conflict.

Class 11

In class 11 there is a signature set, but no type set. Because there is a signature set, the set of methods that can reach the action is finitely bounded. Therefore, we do not have an infinite signature conflict. Only methods in the signature set can reach the action. Therefore, all methods from the signature set are added to the signature of the concern.

The following probe methods are also added to check for cyclic dependency conflicts:

- Infinite: method `?(?)` is added to detect this conflict.
- Name finite: In this class no bound is put on only the name, so no probe methods are added.
- Type finite: In this class no bound is put on only the types, so no probe methods are added.
- Fully finite: Only the methods from the signature set put a finite bound on the entrance methods. These methods are already added.

Class 12

In class 12 there is a signature set and a type set. Just as with class 11, only the methods in the signature set can reach the action. Therefore, these methods are added to the signature of the concern.

The following probe methods are also added to check for cyclic dependency conflicts:

- Infinite: method `?(?)` is added to detect this conflict.
- Name finite: In this class no bound is put on only the name, so no probe methods are added.
- Type finite: The type set can put a type bound on the entrance methods. Therefore, the selector `?` is added with all types from the type set.
- Fully finite: Only the methods from the signature set put a finite bound on the entrance methods. These methods are already added.

The procedures for these classes are similar to the procedures for the `Dispatch` action classes. Therefore, they are not given here.

8.3 Design and Implementation

This section presents the design and implementation details of the signature generation engine in Compose⋆. First, the location of signature generation in the compilation process is explained. This is followed by an explanation of the class structure. No further implementation details are given of the algorithms, because the implementations correspond directly to the algorithms explained in the previous section.

8.3.1 Location in the Compilation Process

Figure 2.3 on page 21 shows the architecture of Compose⋆ and the position of signature generation within this architecture. Signature generation depends on filter reasoning, on superimposition being resolved and on the filter module orders being created. Therefore, it is placed after the filter reasoning preprocessor, the superimposition analysis and the filter composition and checking. Modules that depend on signature generation are the signature transformer, the weavespec generator and the inliner. Signature generation should be placed before these modules.

8.3.2 Class Structure of the Signature Generation Engine

This section explains the class structure of the signature generation engine. Figure 8.9 shows the class structure of the signature generation engine.

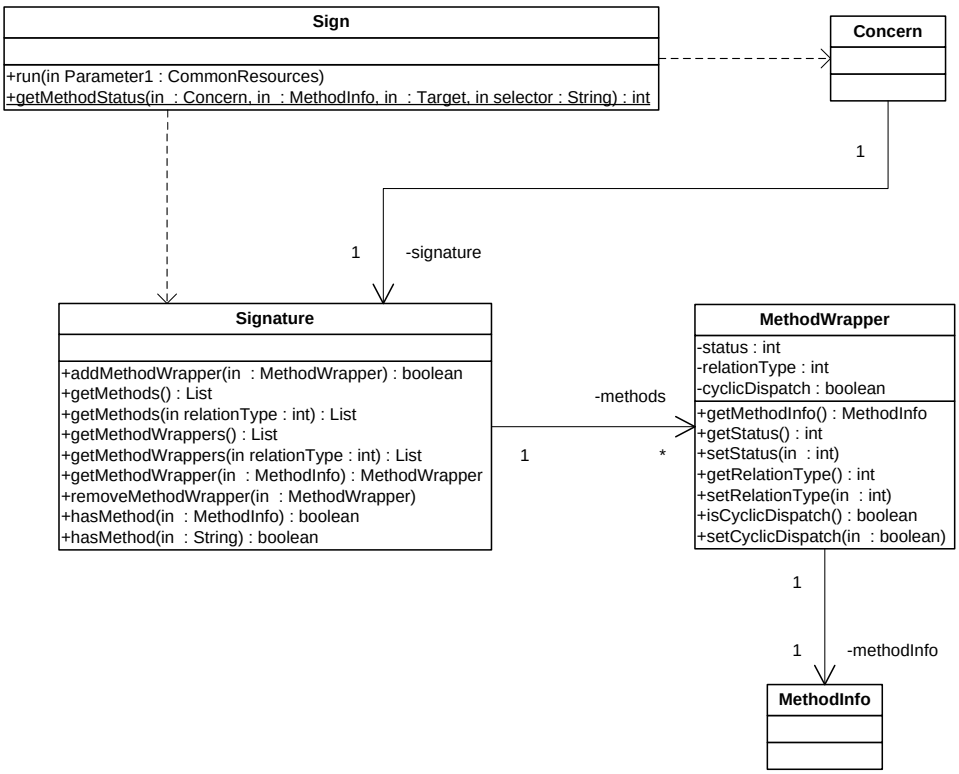


Figure 8.9: Class structure of the signature generation engine.

Sign

The main class of the signature generation engine is `Sign`. This class is located in the package `Composestar.Core.SIGN2`. `Sign` implements all procedures given in this chapter.

Signature

The `Signature` class is used to represent the signature generated by `Sign` for a concern. A `Signature` object has zero or more `MethodWrapper` objects. These objects represent the methods in the signature. `Signature` has the following methods:

- `addMethodWrapper(MethodWrapper)`: To add a new method to the signature.
- `getMethods()`: To get a list of all methods, as `MethodInfo` objects.
- `getMethods(int relationtype)`: To get a list of all methods that have a specific relationtype, as `MethodInfo` objects.
- `getMethodWrappers()`: To get a list of all methods, as `MethodWrapper` objects.
- `getMethodWrappers(int relationtype)`: To get a list of all methods that have a specific relationtype, as `MethodWrapper` objects.
- `getMethodWrapper(MethodInfo)`: To get the `MethodWrapper` object corresponding to the given `MethodInfo` object.
- `removeMethodWrapper(MethodWrapper)`: To remove a method from the signature.
- `hasMethod(MethodInfo)`: To check whether the signature contains a certain method.
- `hasMethod(name)`: To check whether the signature contains a method with the given name.

MethodWrapper

The `MethodWrapper` class is a wrapper class for the `MethodInfo` class in the `Signature` . This wrapper class contains, beside the corresponding `MethodInfo` object, the following information:

- Status information: The status of the method, used during the signature generation. The status can be `Existing`, `NotExisting` or `Unknown`.
- Relation type information: The relation of the method with a method in the implementation signature. This information is set after the signature generation has ended and is used by the module that handles signature expansion/contraction. The relation type has the value `Added` if the method is new in the concern. The relation type has the value `Removed` if the method is removed from the concern (the status is also `NotExisting`). The relation type has the status `Normal` if the method is maintained from the implementation.

Concern

The class `Concern` is the standard class to represents a concern. This class is located in the package `Composestar.Core.RepositoryImplementation.DeclaredRepositoryEntity`. The signature generation engine attaches a `Signature` object to each concern.

MethodInfo

The class `MethodInfo` is the standard method info class. It is located in the package `Composestar.Core.LAMA`.

Chapter 9

Behavioral Reasoning

"Behavior is the mirror in which everyone shows their image."
Johann Wolfgang von Goethe
German poet (1749 - 1832)

Imposing multiple aspects onto a single join point might introduce several problems. One class of these problems are behavioral conflicts; behavioral conflicts are conflicts caused by the "purpose or side effects of the aspects" [11]. Behavioral reasoning tries to detect these conflicts using a resource operation model; a filter set execution is translated to sequences of operations on certain resources. To these sequences of operations specific regular expressions are applied to find conflicting patterns or assertions.

Filter reasoning can be used by behavioral reasoning to obtain all possible execution paths through the filter set. This chapter explains how behavioral reasoning can make best use of filter reasoning and how the execution model obtained from filter reasoning can make behavioral reasoning actually more efficient. This chapter does not explain behavioral reasoning in dept, as there are no conceptual changes made to behavioral reasoning itself. The first section explains the theory and concepts using filter reasoning for behavioral reasoning. The second section describes the design and implementation details of the integration of the filter reasoning engine into the behavioral reasoning engine in Compose⋆.

9.1 Theory and Concepts

This section explains the theory and concepts behind using filter reasoning to do behavioral reasoning. It explains how behavioral reasoning can use the models obtained from filter reasoning and how these models can actually make behavioral reasoning more efficient.

The first part of this section gives an introduction into behavioral reasoning. This is followed by an explanation of how behavioral reasoning can benefit from filter reasoning. This leads to a first approach of how to use filter reasoning for behavioral reasoning, which, however, proves to have worst case exponential time complexity. The last part of this section explains that this exponential time complexity is not inherent to the

problem. A smarter regular expression matching technique is presented to make the time complexity of behavioral reasoning polynomial without loosing any power.

9.1.1 Introduction to Behavioral Reasoning

Behavioral reasoning tries to detect behavioral conflicts in filter set definitions. It uses a resource operation model to do this. The semantics of certain steps in a filter set execution can be represented as operations on specific resources. For example, checking whether a matching part matches a message leads to a `read` operation on both the `selector` resource and the `target` resource. A complete execution of the filter set leads to a sequence of operations on each resource.

An execution of a filter set might lead to a behavioral conflict. An example of such a conflict on the selector resource might be two consecutive `read` operations followed by two consecutive `write` operations. This might indicate a mutual exclusion problem. In this example, the correct sequence should have been `read-write-read-write`.

Behavioral reasoning tries to detect these conflicts by using specific patterns of resource operations that indicate these conflicts. Regular expressions are used to define these patterns. For each possible execution of a filter set, the resulting sequences of operations on each resource are searched for these patterns. If a match is found, a conflict has been detected.

More information about behavioral reasoning can be found in [9, 10, 11].

9.1.2 Using Filter Reasoning for Behavioral Reasoning

Behavioral reasoning checks every execution path through the filter set whether the resulting sequence of resource operations matches certain patterns. To obtain every execution path through the filter set filter reasoning can be used. The advantage of using filter reasoning is that filter reasoning only gives the valid execution paths; execution paths that are impossible due to message matching, are not given. In this way, only conflicts are detected that can really occur in practice.

Efficiency Problems So, in its simplest form the algorithm works as follows: filter reasoning is used to give every possible execution path. For each execution path the sequences of resource operations are generated. These sequences of resource operations are matched against the patterns indicating the conflicts. If a match is found, a conflict has been detected.

A problem with this algorithm is that there might be exponentially many execution paths through a filter set. Listing 9.1 gives an example of a filter set that has exponentially many execution paths. Figure 9.1 shows a schematic representation of the flowchart of this filter set. Each node represents a filter. Each filter node has two outgoing edges, one indicating an accept and one indicating a reject. In this example, all filters can accept as well as reject at any time, independent from the acceptance or rejection of earlier filters. On top of this, flow always continues to the next filter. The combination of these two properties leads to a doubling of the number of possible execution paths for every additional filter. Therefore, the given example has $2^3 = 8$ possible execution paths.

```
logging : Logging      = { C1 => [*.*]   };
timing  : Timing       = { C2 => [*.*]   };
subst   : Substitution = { C3 => [*.*] *.a };
```

Listing 9.1: Example of a filter set with exponentially many execution paths

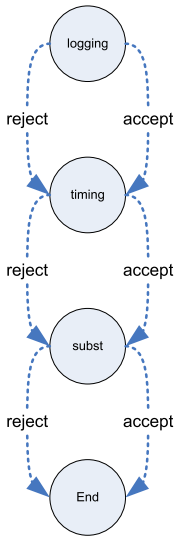


Figure 9.1: Schematic representation of the flowchart.

This exponential growth of the number of execution paths in the size of the filter set makes the presented algorithm inefficient.

9.1.3 An Algorithm with Polynomial Time Complexity

The exponential time complexity of the behavioral reasoning conflict detection algorithm is not fundamentally inherent to its nature, but a result of its design. The detection algorithm can be changed to an algorithm with polynomial time complexity, without loosing any power. This section describes how this can be done.

9.1.3.1 A Smarter Matching Algorithm

The Execution Model as a Non-Deterministic Finite Automaton The execution model generated during filter reasoning is a state transition system. A transition between two states corresponds to an evaluation step in a filter set execution, as was explained in Chapter 5. During such an evaluation step certain operations on certain resources might occur. Because an evaluation step corresponds to a transition in the execution model, the same resource operations also apply to that transition. So, we can attach information to an execution transition indicating which resource operations occur in the corresponding evaluation step. This information can be attached to the execution transition by means of a label. The empty label (ϵ) is used for transitions that have no corresponding resource operations.

If we apply this labeling to all execution transitions for one specific resource, the result is a labeled execution model for that specific resource. This labeled execution

model is basically a non-deterministic finite automaton with ϵ transitions (ϵNFA). The entrance state of the execution model is the entrance state of the ϵNFA. The `End`, `Return` and `Stop` states of the execution model are the accepting states of the ϵNFA.

So, each execution path through the labeled execution model is an accepting path through the ϵNFA and each accepting path through the ϵNFA is an execution path through the labeled execution model. From this it follows that the set of all resource operation sentences for all execution paths through an execution model M is equal to the language generated by ϵNFA_M:

Theorem 9.1.1 (Execution model - ϵNFA language equality) *Let M be an execution model corresponding to a filter set and ϵNFA_M be the ϵNFA constructed by labeling all transitions of M with the corresponding operations on a specific resource, taking the entrance state of M as the entrance state and the ending states of M as the accepting states. Then the following holds:*

$$\{s|s = operationSequence(path), path \in executionPaths(M)\} = L(\epsilon NFA_M)$$

Where $executionPaths(M)$ is the set of all execution paths through M and $operationSequence(ExecutionPath)$ gives the sequence of resource operations for a certain execution path.

Checking for Violated Constraints The matching patterns to detect conflicts consist of regular expressions. Each regular expression accepts a language. The sentences in the language are sequences of operations on the resource to which the regular expression applies. The ϵNFA_M constructed from the execution model M also accepts a language consisting of sequences of operations on that specific resource. If the intersection of both languages is not empty, there is an execution on which the regular expression matches:

Theorem 9.1.2 (Constraint violation) *Let M be an execution model and ϵNFA_M the corresponding ϵNFA for a certain resource R. Let P be a regular expression indicating a constraint on resource R. Then there is an execution path through M that violates P on resource R if and only if:*

$$L(\epsilon NFA_M) \bigcap L(P) \neq \emptyset$$

Transforming a Regular Expression to an ϵNFA A regular expression can be transformed in polynomial time to an ϵNFA (ϵNFA_{expr}) that accepts the same language as the regular expression [50]. From this ϵNFA and the ϵNFA constructed from the execution model (ϵNFA_M), a new ϵNFA (ϵNFA_{int}) can be constructed that is the intersection of both ϵNFA's. This intersection ϵNFA accepts the intersection of the languages accepted by the two original ϵNFA's. So, to check the violation of the constraint we need to check whether the language accepted by the intersection ϵNFA is empty. How this intersection ϵNFA can be constructed is explained next.

Intersecting Two Non-Deterministic Finite Automata with ϵ Transitions Before we describe how two ϵNFA's can be intersected, we first give a formal definition of an ϵNFA.

Definition 9.1.1 (Non-deterministic finite automaton with ϵ transitions) *A non-deterministic finite automaton with ϵ transitions (ϵNFA) can be represented by the tuple $\langle Q, \Sigma, \delta, s, F \rangle$, where*

- *Q is a finite set of states*
- *Σ is the alphabet*
- *$s \in Q$, is the start state*
- *$F \subseteq Q$, is the set of accepting states*
- *δ is the total function $Q \times (\Sigma \bigcup \{\epsilon\}) \rightarrow \wp(Q)$, called the transition function, that maps a start state and a word from the alphabet Σ or ϵ to a set of next states.*

[50]

The intersection of two ϵNFA's can now be constructed as follows:

Theorem 9.1.3 (Intersecting two ϵNFA's) *Let $\epsilon NFA_1 = \langle Q_1, \Sigma, \delta_1, s_1, F_1 \rangle$ and $\epsilon NFA_2 = \langle Q_2, \Sigma, \delta_2, s_2, F_2 \rangle$ be two ϵNFA's. The intersection $\epsilon NFA_{int} = \epsilon NFA_1 \bigcap \epsilon NFA_2 = \langle Q_{int}, \Sigma, \delta_{int}, s_{int}, F_{int} \rangle$ can now be constructed as follows:*

- $Q_{int} = Q_1 \times Q_2$
- $s_{int} = \langle s_1, s_2 \rangle$
- $F_{int} = F_1 \times F_2$
- $\delta_{int}(\langle q_1, q_2 \rangle, w) = \delta_1(q_1, w) \times \delta_2(q_2, w)$

[50]

Checking Whether the Intersection is Empty To check whether the language accepted by ϵNFA_{int} is not empty, we need to check whether there exists a path from the start state to the end state. This can be done by a reachability analysis, which means that it is checked which states are reachable from the entrance state.

Summary So to summarize, the new algorithm works as follows:

- Labels representing resource operations are added to the execution model. This turns the execution model into an ϵNFA (ϵNFA_{em}).
- The regular expression to match is also transformed to an ϵNFA (ϵNFA_{expr}).
- The intersection of the execution model ϵNFA_{em} and the regular expression ϵNFA_{expr} is created. This is also an ϵNFA (ϵNFA_{int}).
- For ϵNFA_{int} it is checked whether the language is empty. If it is not empty, the constraint is violated.

9.1.3.2 Time Complexity of the New Algorithm

We will now calculate the time complexity of constructing the intersection of two ϵNFA's and of checking whether the language of the resulting ϵNFA is empty. First, the time complexity of each part of the construction of the intersection of two ϵNFA's is calculated:

Construction of the States Q_{int} The states in the intersection are constructed by taking the cross product of the states of both original ϵNFA's. The time complexity of this cross product is $O(\#Q_1 \cdot \#Q_2)$.

Construction of the Entrance State s_{int} The entrance state of the intersection is just the combination of the entrance states of both original ϵNFA's. So, the time complexity is $O(1)$.

Construction of the Accepting States F_{int} The accepting states are created by taking the cross product of the accepting states of both original ϵNFA's. So, the time complexity is $O(\#F_1 \cdot \#F_2)$. Because the size of the set of accepting states can be as large as the size of the set of states, the worst case complexity is equal to $O(\#Q_1 \cdot \#Q_2)$.

Construction of the Transition Function δ_{int} The result of the transition function for one state and one word is constructed by taking the cross product of the results of the transition functions of both original ϵNFA's. The time complexity of taking this cross product is $O(\#Q_1 \cdot \#Q_2)$, because the size of the result set of the transition function can be as large as the number of states.

This cross product is done for each state in the intersection ϵNFA and for each word in the language. This means that the cross product is done $O(\#Q_1 \cdot \#Q_2 \cdot \#L)$ times.

So, the total time complexity of constructing the transition function becomes $O(\#Q_1^2 \cdot \#Q_2^2 \cdot \#L)$. Fortunately, both the number of outgoing transitions from a state in the execution model as from a state in the regular expression automaton is bounded by a certain constant value (see Chapter 5 and [50]). This means that the size of the output of the transition functions in both the original ϵNFA's is bounded by that constant value. So, the complexity of constructing the result of the transition function for one state and one word becomes $O(c)$ instead of $O(\#Q_1 \cdot \#Q_2)$. Therefore, the complexity of constructing the transition function becomes $O(\#Q_1 \cdot \#Q_2 \cdot \#L)$ for this application of automata intersection, instead of $O(\#Q_1^2 \cdot \#Q_2^2 \cdot \#L)$.

Checking Emptiness of the Intersection Language The time complexity of the reachability analysis to check the emptiness of the intersection language is linear in the number of states, so $O(\#Q_1 \cdot \#Q_2)$

Total time complexity Adding the complexities gives a total time complexity of $O(\#Q_1 \cdot \#Q_2 \cdot \#L)$ for the application of the automata intersection in this context. So, the algorithm is both linear in the size of the execution model as linear in the size of the regular expression. As explained in Section 5.4, the size of the execution model is worst case the size of the filter set to the power three.

In general, the time complexity would have been $O(\#Q_1^2 \cdot \#Q_2^2 \cdot \#L)$.

9.1.3.3 What About Assertions?

The presented algorithm tries to find one trace that matches the given regular expression. This is sufficient for regular expressions that denote a constraint. Finding a path that matches means that the constraint is violated. But behavioral reasoning does not only want to check constraints with regular expressions. It also want to check assertions with them. Finding one path that matches the regular expression is not sufficient for assertions, because the assertion needs to be true on all paths. So, checking whether the assertion is never violated means checking every path. This becomes a problem again, because there might be exponentially many paths.

Fortunately, we can take another approach for matching assertions. An assertion holds if and only if the sentences generated by all paths in the execution model are accepted by the regular expression. The sentences of all paths in the execution model form the language of the execution model. If a sentence is accepted by a regular expression, the sentence is in the language of the regular expression. So, the assertion holds if and only if all sentences from the language of the execution model are in the language of the regular expression:

$$\text{Assertion } \alpha \text{ holds on execution model EM} \Longleftrightarrow L_{EM} \subseteq L_\alpha$$

Checking whether L_{EM} is a subset of L_α is equivalent to checking whether the intersection of L_{EM} with the complement of L_α is empty:

$$L_{EM} \subseteq L_\alpha \Longleftrightarrow L_{EM} \bigcap \overline{L_\alpha} = \emptyset \qquad \text{Where } \overline{L_\alpha} \text{ is the complement of } L_\alpha, \text{ defined as } \Sigma^* - L_\alpha$$

But this is equivalent to checking whether the constraint $\overline{\alpha}$ that accepts language $\overline{L_\alpha}$ is violated. Because the complement of a regular language is also a regular language [50], the constraint $\overline{\alpha}$ is also a regular expression. This means that checking this constraint can be done in the same way as checking other constraints.

So, checking an assertion can be done by creating the complementary regular expression and checking this complementary regular expression as a constraint. The complement of a regular expression can be constructed as follows [50]:

- Create the ϵNFA corresponding to the regular expression. The size of this ϵNFA is linear to the size of the regular expression.
- Transform the ϵNFA to an equivalent deterministic finite automaton (DFA). The size of this DFA is worst case exponential to the size of the ϵNFA
- Invert all end states of the DFA. This constructs the complementary DFA.
- Transform the complementary DFA to a regular expression.

The problem with creating the complementary regular expression is that the size of this expression might be exponentially larger than the size of the original regular expression.

9.2 Design and Implementation

The `java.util` package provides a regular expression matcher. This matcher is, however, not used in the implementation, because it is specific for matching regular expression on strings. To be as efficient as possible, our matcher needs to match a regular expression directly on the execution model. Therefore, a new regular expression matcher, specifically for matching regular expressions on execution models, is implemented

This section describes the design and implementation of this matcher. In the first part the implemented regular expression language is described. This is followed by an explanation of the class structure of the matcher. The third part of this section explains the important implementation details; how a regular expression is transformed to an automaton and what variant of the intersection algorithm is implemented. Finally, the integration of the matcher with the behavioral reasoning tool `SECRET` is described.

9.2.1 Implemented Regular Expression Language

The regular expression matcher implements only the basic regular expression language features. This section presents the syntax of the implemented regular expression language.

Listing 9.2 shows the grammar of the implemented regular expression language in EBNF. The various aspects of the language are explained next.

```
regexpr ::= unionexpr
unionexpr ::= concatexpr ('|' unionexpr)?
concatexpr ::= starexpr (concatexpr)?
starexpr ::= basicexpr ('*')?
basicexpr ::= word | '[' regexpr ']' | '!' wordsequence

wordsequence ::= '(' word ('|' word)* ')'
word ::= letterordigit | '(' letterordigit+ ')' | '.'
```

Listing 9.2: The regular expression grammar in EBNF

9.2.1.1 Operators

The regular expression language contains three operators: The union operator, the concatenation operator and the Kleene star operator.

Union

The union of two regular expressions can be taken by placing a '|' between them:

$$e = e_1|e_2$$

A union of two regular expressions means that at least one of the operand expressions needs to match to let the combined expression match.

Concatenation

A concatenation of two regular expressions can be made by just placing them in sequence:

$$e = e_1e_2$$

A concatenation of two regular expressions e_1 and e_2 means that the regular expression matches on a sentence if and only if e_1 matches the first part of the sentence and e_2 matches the second part of the sentence.

Kleene Star

The Kleene star operator can be applied by placing a '*' behind the expression:

$$e = e_1*$$

If the Kleene star operator is applied to regular expression e_1, e_1 needs to match zero or more times.

9.2.1.2 Operator Precedence

The Kleene star operator has the highest precedence. This is followed by the concatenation operator. The union operator has the lowest precedence.

The precedence ordering can be circumvented by placing the operand between square brackets. For example, if we want to apply the Kleene star operation to the regular expression $a|b$, then we have to write $[a|b]*$. If we had written $a|b*$ then the Kleene star operator would only have been bound to the b and the meaning of the expression would have been one occurrence of a or zero or more occurrences of b.

9.2.1.3 Constructing Words

The regular expression language contains constructs to create single words, a wild card operator to match any word from the language and constructs for matching on the complement of a set of words.

Single Words

A single word is either one letter or a sequence of letters between parentheses. For example (abc) creates the single word abc instead of a concatenation of a, b and c.

Wild Card Symbol

The wild card symbol '.' can be used to match any word from the language.

Negation

The negation operator '!' can be used to match on the complement of the set of words. The negation operator only works on a sequence of words. A sequence of words starts with a left parenthesis and ends with a right parenthesis. The words in the sequence are divided by '|' symbols:

$$e = !(a|b|c|d|e)$$

The negation expression matches if the to be matched word is not in the set of words.

The negation operator can be seen as another operator just as the other three operators. The difference is that for the other three operators any regular expression can be used as an operand; the negation operator can only take a sequence

of words as the operand. Therefore the negation operator always binds harder to the sequence of words than the other operators.

The negation operator is not implemented for regular expressions in general, because creating the complement of a regular expression other than a union of single words is a complex operation. How to create a complement of a regular expression was explained in Section 9.1.3

9.2.2 Class Structure of the Regular Expression Matcher

This section presents the class structure of the regular expression matcher. This class structure is divided into two parts, the public structure and the internal structure. First the public structure is explained. This is the class structure as it can be used by other packages. Part of the class structure is needed only for the implementation of the algorithm and should not be used by other packages. This part is hidden in the internal structure.

9.2.2.1 Public Structure

The regular expression matcher is contained in the package `FIRE2.tools.regex`. Figure 9.2 shows the public structure of the regular expression matcher. The interface of the regular expression matcher is designed in such a way that it is as similar as possible to the regular expression matcher in the `java.util` package.

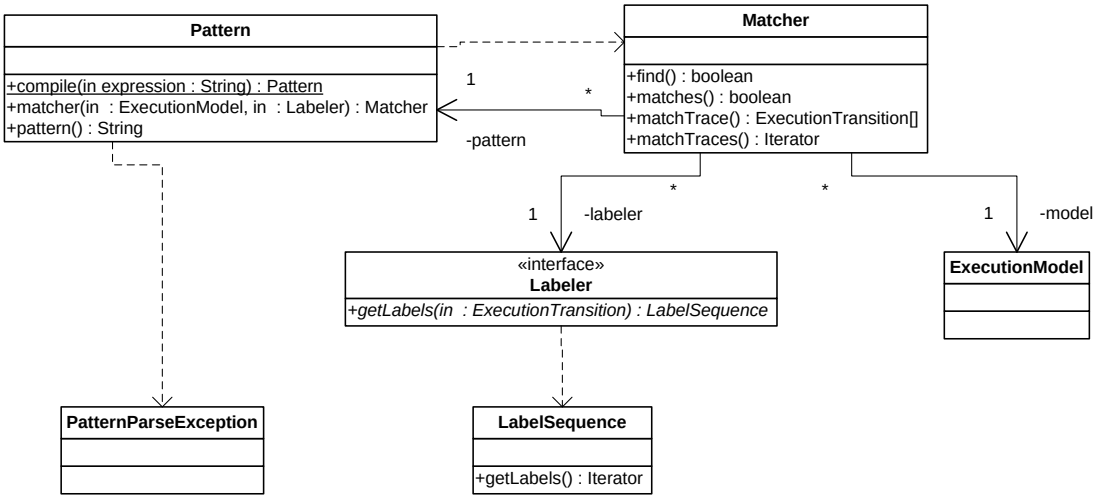


Figure 9.2: Regular expression matcher public structure

Labeler

Implementations of the `Labeler` interface are responsible for adding labels to execution transitions. They do not really attach the labels to the transitions. Instead, they implement the `getLabels` method, which is used to query the label of a certain transition. In this way different tools can add different labels to the transitions, each for their own purpose, without interfering with each other. The `getLabels` method returns a `LabelSequence` object.

LabelSequence

`LabelSequence` represents a sequence of labels that can be added to a transition. In certain cases a transition might not be atomic, meaning that one transition might actually be a sequence of transitions with a sequence of labels. Therefore, the method `getLabels` returns a `LabelSequence` instead of a single label.

Pattern

`Pattern` represents a regular expression. It contains a static method `compile` which gets as a parameter a string representation of a regular expression and returns a corresponding pattern object. If the string given to the `compile` method is not a legal regular expression, a `PatternParseException` is thrown. `Pattern` also contains a method `matcher` that returns a `Matcher` object that matches the pattern on a given `ExecutionModel`.

Matcher

`Matcher` implements the matching algorithm. An instance of `Matcher` tries to match a given `Pattern` on a given `ExecutionModel`, using a given `Labeler` to add the labels to the execution transitions. Such an instance of `Matcher` can be created by calling the `matcher` method on an instance of `Pattern` with an `ExecutionModel` and a `Labeler` as parameters.

`Matcher` contains the method `matches`, which returns `true` if there is a path in the execution model that matches the pattern.

The method `find` returns `true` if there is a path in the execution model that contains a subpath that matches the pattern. This checking is done by expanding the pattern with a wildcard-kleene star sequence (.*) at the beginning and at the end of the pattern and using the `matches` method.

If the pattern matches, the `matchingPath` method can be used to get the first matching path through the execution model. This path is returned as a sequence of execution transitions, as such a sequence uniquely represents a path.

The `matchingPaths` method can be used to get an iterator to iterate over all matching paths. Problem with this is that there might be exponentially many matching paths. Therefore the iterator generates the next matching path on the fly, instead of generating all paths in advance. In this way a client can prevent the iterator from becoming exponential by restricting the number of paths being iterated to a certain constant, for example only the first hundred.

PatternParseException

A `PatternParseException` is thrown if the string given to the `compile` method in `Pattern` is not a valid regular expression.

ExecutionModel

`ExecutionModel` is the representation of an execution model from the package `FIRE2 .model`, as explained in Section 6.5.

9.2.2.2 Internal Structure

The previous section explained the public structure of the regular expression matcher. This public structure is not the complete structure of the regular expression matcher. There are also a number of components that are needed for the implementation of the algorithm. These components are unimportant for the users of the regular expression matcher. Therefore they are hided in the internal structure of the package. Figure 9.3 shows the internal components in the package and their relationship with `Pattern` and `Matcher`.

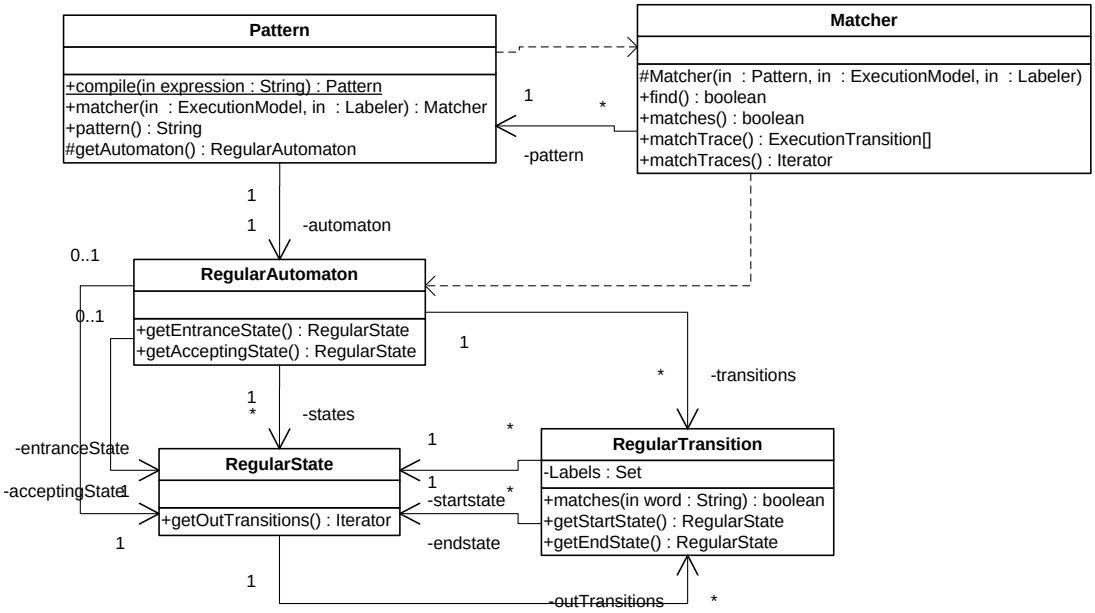


Figure 9.3: Regular expression matcher internal structure

RegularAutomaton

`Pattern` compiles the regular expression internally to a non-deterministic finite automaton with ϵ transitions. An instance of the `RegularAutomaton` class represents such an automaton. The automaton is constructed in such a way that it has only one accepting state. The automaton is used by `Matcher` in the matching algorithm. How this automaton is constructed is discussed in Section 9.2.3.

RegularState

`RegularState` represents a state in the automaton. It maintains a list of outgoing transitions.

RegularTransition

`RegularTransition` represents a transition in the automaton. It maintains the start state and the end state of the transition. It also contains a method `matches` to check whether a certain word from the alphabet matches the transition.

9.2.3 Implementation Details

This section explains important implementation details of the regular expression matcher. It first explains how a regular expression is transformed to an automaton. Then it explains the implemented variant of the matching algorithm.

9.2.3.1 Transforming a Regular Expression to an Automaton

To do the matching, the regular expression needs to be transformed to an automaton. For this transformation we use the algorithm from [50]. This section explains that algorithm.

Regular expressions can be divided into two classes: basic regular expressions and complex regular expressions. Basic regular expressions consist of just a single word. Complex regular expressions consist of an operator and one or two operands, depending on the type of the operator. The operands are also regular expressions.

A complex regular expression is recursively transformed to an automaton. First the operands are transformed to automata. Then these automata are combined in a way specific for the type of the operator. This results in the automaton of the complete regular expression. The algorithm works in such a way that an automaton constructed from a regular expression has always only one accepting state.

We will describe first how the base case, a regular expression consisting of just one word, is translated to an automaton. This is followed by an explanation for each type of operator of how the corresponding automaton can be created as a combination of the automata of the operands.

In the explanation of the operators the operands are represented by e_1 and e_2. Figure 9.4 shows how the automaton corresponding to e_1 is represented. The entrance state (S_{e1}) and the accepting state (F_{e1}) are explicitly shown. All other states are abstracted away in the box A_{e1}. The automaton of e_2 is represented in a similar way.

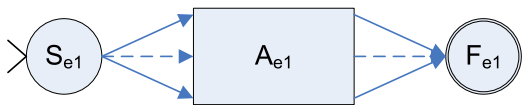


Figure 9.4: Schematic representation of the automaton corresponding to regular expression e_1.

Base Case: Single Word Regular Expression

Basic regular expressions consist of just a single word:

$$e = w$$

Figure 9.5 shows how such a regular expression is translated to an automaton. Two states are created, one entrance state and one accepting state. A transition is added from the entrance state to the accepting state. The transition is labeled with the word from the regular expression.

The automata for the wild card symbol and for the complement (negation) sequence are created in the same way. For the wild card symbol, the transition will

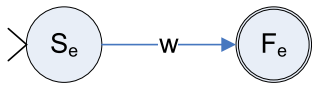


Figure 9.5: A single word regular expression.

match any word. For the complement sequence, the transition will match if the word is not in the complement sequence.

Concatenation

Two regular expression can be concatenated to one regular expression:

$$e = e_1 e_2$$

Figure 9.6 shows that such a regular expression is translated to an automaton by placing the two automata of e_1 and e_2 in sequence. The entrance state of the automaton of e_1 becomes the entrance state of the combined automaton. The accepting state of the automaton of e_2 becomes the accepting state of the combined automaton. An ϵ transition is added from the accepting state of the automaton of e_1 to the entrance state of the automaton of e_2.

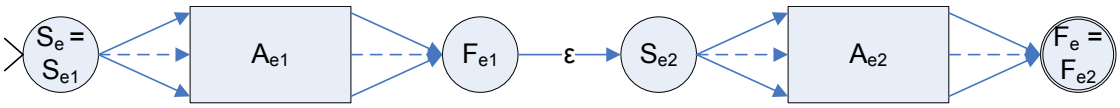


Figure 9.6: Concatenation of e_1 and e_2.

Union

A regular expression can also be a union of two regular expressions:

$$e = e_1 | e_2$$

This means that a sentence has to match at least on one of the regular expressions. Therefore the combined automaton makes an arbitrary choose which one of the operand automata to execute, as can be seen in Figure 9.7. To create the combined automaton, an entrance state and an accepting state are added. Two ϵ transitions originating from the entrance state are added, one to the entrance state of each operand automaton. These ϵ transitions implement the arbitrary choice between the two automata. The accepting state is added to not to break the rule of each automaton having just one accepting state. From the accepting states of both operand automata an ϵ transition to the accepting state of the combined automaton is added.

Kleene Star

Certain regular expressions might be repeated zero or more times. Such a regular expression is constructed with the Kleene star operator:

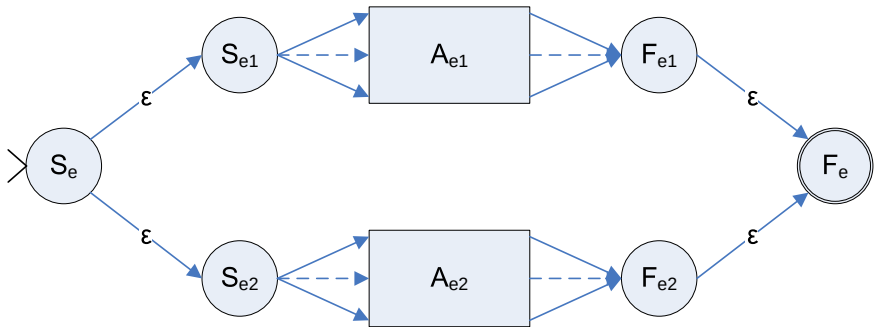


Figure 9.7: Union of e_1 and e_2.

$$e = e_1^*$$

Figure 9.8 shows how such a regular expression is transformed into an automaton. An entrance state and an accepting state are added. From the entrance state an ϵ transition to the accepting state is added, to construct the zero-times iteration. Also an ϵ transition is added from the entrance state of the combined automaton to the entrance state of the operand automaton and an ϵ transition from the accepting state of the operand automaton to the accepting state of the combined automaton, to implement the one time iteration. To implement the multiple times iteration of the operand expression, an ϵ transition is added from the accepting state of the operand automaton to the start state of the operand automaton.

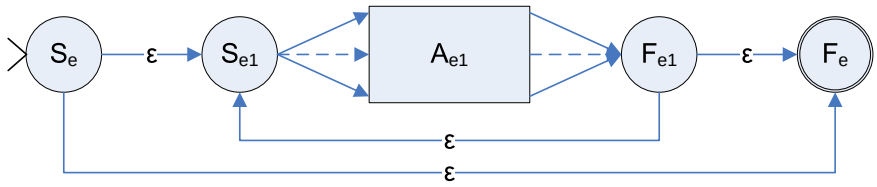


Figure 9.8: Kleene star operator on e_1.

9.2.3.2 Implemented Variant of the Matching Algorithm

The implementation of the matching algorithm does not create the entire intersection ϵNFA. Instead it incrementally adds states reachable from the entrance state until an accepting state is reached, or until no more states can be added. If an accepting state is reached, there is a path from the entrance state to the accepting state, so the regular expression matches. If no accepting state is reached and no more states can be added, there is no path from the entrance state to an accepting state, so the regular expression does not match.

The algorithm starts with adding the entrance state of the intersection ϵNFA. Then iteratively: for each newly added state all outgoing transitions are created. If the end state of such a transition does not already exist in the model, it is added. If the added state is an accepting state, a path from the entrance state to an accepting state has been

found and the algorithm ends with a match. If there are no more new states to check and no accepting state has been found, the algorithm ends with a no match.

Algorithm 9.1 formally describes the algorithm.

Algorithm 9.1: Constructing the intersection of two ϵNFA's

```
   input  : Two automata α1 and α2
   output: Boolean indicating whether L(α1) ⋂ L(α2) ≠ ∅
1  s1 := α1.entranceState
2  s2 := α2.entranceState
3  states := {⟨s1, s2⟩}
4  newStates := states
5  while newStates ≠ ∅ do
6      ⟨q1, q2⟩ ∈ newStates
7      newStates := newStates/⟨q1, q2⟩
8      foreach ⟨q1, w1, r1⟩ ∈ q1.outTransitions do
9          foreach ⟨q2, w2, r2⟩ ∈ q2.outTransitions do
10             if w1 = w2 ∧ ⟨r1, r2⟩ ∉ states then
11                 if r1 = α1.acceptingState ∧ r2 = α2.acceptingState then
12                     return true
13                 end
14                 states := states ⋃ {⟨r1, r2⟩}
15                 newStates := newStates ⋃ {⟨r1, r2⟩}
16             end
17         end
18     end
19 end
20 return false
```

9.2.4 Integration with the Behavioral Reasoning Engine

This section describes how the regular expression matcher is integrated with the behavioral reasoning tool `SECRET`. First it is explained why the `ExecutionAnalysis` component is removed. Then the integration of the regular expression matcher into the abstract virtual machine component is described.

9.2.4.1 Removal of the execution analysis

The original version of `SECRET` did the analysis on each filter set execution. Figure 9.9 shows the sequence diagram corresponding to `SECRET`'s original analysis.

In the original version, the `FilterSetAnalysis` component was responsible for generating all sequences through the filter set. This was done by just taking all possible flow paths through the filter set, ignoring the fact that some of these paths might not be possible due to message matching.

The `ExecutionAnalysis` component was responsible for doing the analysis for one single execution path through the filter set. It translated the execution path into a

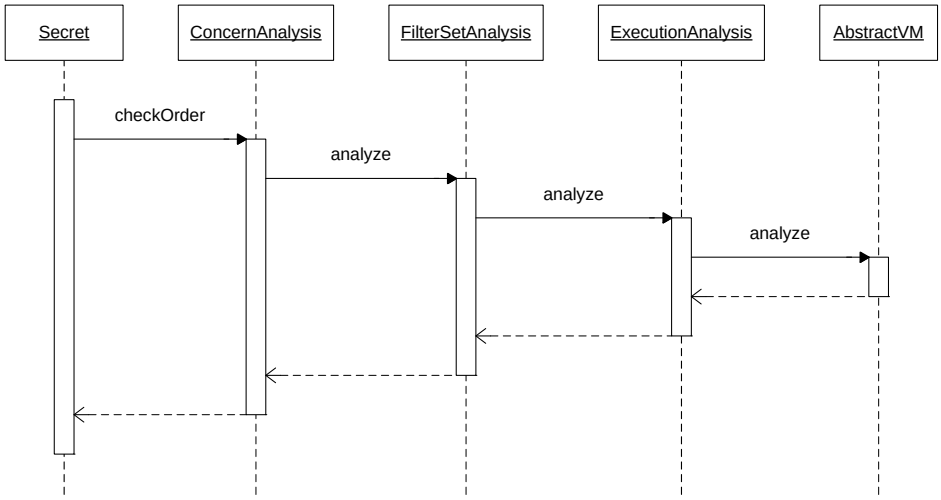


Figure 9.9: Sequence diagram of SECRET's original analysis phase

sequence of filter actions and called the `AbstractVM` component to do the analysis on the sequence of actions.

The `AbstractVM` component used the sequence of actions to create sequences of resource operations. These sequences of resources operations were tested whether they matched one of the constraint regular expressions, to check for conflicts.

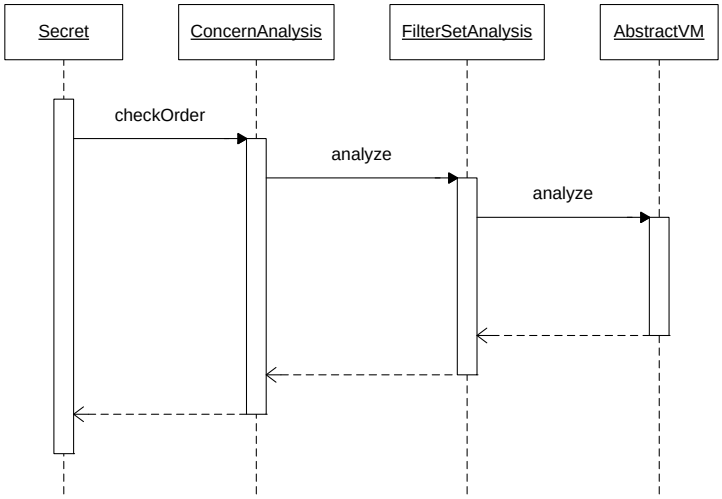


Figure 9.10: Sequence diagram of SECRET's new analysis phase.

With the integration of `FIRE` into `SECRET`, the analysis phase has been changed from doing the analysis on each execution path to doing the analysis on the entire execution model at once. Therefore the `ExecutionAnalysis` component has become obsolete and is removed. Figure 9.10 shows the sequence diagram corresponding to the new analysis phase.

The `FilterSetAnalysis` component does not create every possible execution anymore. Instead, it obtains the execution model corresponding to the filter set and directly calls

the `AbstractVM`, giving the execution model as an argument.

The `AbstractVM` component is responsible for adding resource operation labels to the execution model and for matching the constraint regular expressions. So `AbstractVM` is the actual component that uses the regular expression matcher. How the regular expression matcher is integrated within `AbstractVM` is explained next.

9.2.4.2 Integration into the Abstract Virtual Machine

Figure 9.11 shows how the regular expression matcher is integrated with the `AbstractVM` component in `SECRET`. A subclass of `Labeler` is created to label the transitions with resource operations. The `Constraint` object maintains a reference to the corresponding `Pattern` object.

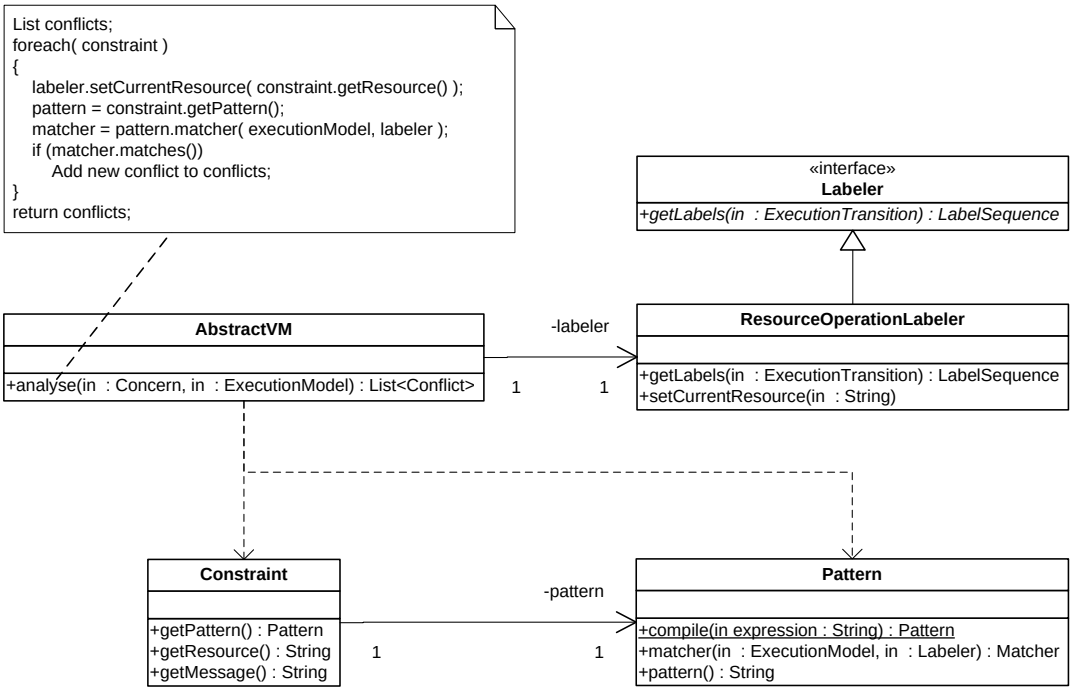


Figure 9.11: Regular expression matcher integrated with SECRET

ResourceOperationLabeler

The class `ResourceOperationLabeler` is an implementation of the interface `Labeler` . It adds labels that represent resource operations to execution transitions. The specific resource for which labels need to be added to the transitions is configurable with the `setResource` method.

The `ResourceOperationLabeler` obtains the information that describes which resource operations should be added to which transition from different sources:

- From the file `filterdesc.xml` in the `SECRET` package or in the project folder.
- From custom attributes on the filter action implementations.
- From hard coded mappings in `ResourceOperationLabeler` itself.

Constraint

The class `Constraint` represents a constraint. It contains an instance of `Pattern`, which is the pattern corresponding to the constraint.

AbstractVM

`AbstractVM` implements the behavioral analysis on one execution model. Its `analyse` method iterates over all constraints. For each constraint a `Matcher` is created. This `Matcher` tries to match the constraint's pattern on the execution model.

Labeler, Pattern

These are classes from the regular expression matcher, explained in Section 9.2.2.

Chapter 10

Filter Inlining

"Efficiency is intelligent laziness."
David Dunham

Compose⋆/.NET uses an interpreter to execute a filter set. Although an interpreter provides some degree of flexibility, it is usually not very efficient. To increase performance, it is better to translate the filter set to the target language code and weave this code at the appropriate places. This is called *filter inlining*. Filter inlining is made possible by filter reasoning. Filter reasoning provides, at compile time, information about how a certain message behaves in a filter set. This information can be used to generate the target language code. This chapter explains filter inlining and how filter reasoning is used to achieve filter inlining.

The first section explains the theory and concepts behind filter inlining. It explains how the filter reasoning model is used to translate the filter set to code and how this generated code can be woven in the target program, to embed the composition filter functionality. The second section describes how filter inlining is designed and implemented in the StarLight version of Compose⋆.

10.1 Theory and Concepts

This section explains the theory and concepts behind filter inlining.

The first task in filter inlining is transforming the filter set specification to (procedural) code, so that it can be woven in the program. Two different solutions, with certain variants, have been identified to transform the filter reasoning model to an inlining structure:

Flow Based Inlining
This solution generates a procedural flow structure that corresponds with the execution of the filter set. This solution can, however, not cope with substitution.

Condition Based Inlining
A filter action is executed if a number of conditions are true. This solution creates a conditional structure to execute the correct filter actions.

Flow Based Inlining with Jump Instruction
This is a variant of the flow based inlining solution that can cope with substitution. It uses `jump` instructions to jump to the correct code when substitution occurs.

Flow Based Inlining with Message Conditions
This variant of flow based inlining can also cope with substitution. It maintains an integer value that represents the message. A `switch` structure is used to execute the corresponding code.

These solutions are explained first. Then, it is described how conditional superimposition can be incorporated into the solutions. After the filter set has been translated to code, this code needs to be woven in the target program. This is explained in the last part of this section.

10.1.1 Flow Based Inlining

The first approach to translate a filter set to code is the *flow based inlining* approach. This approach tries to build a flow structure that corresponds to the natural flow through the filter set. It therefore divides the execution model into blocks of related states.

In this solution it is assumed that there is no substitution. This means that the message stays the same in the entire execution of the filter set. Note that there might be a different working copy of the message created, to execute a `Dispatch` action, for example. In following sections, variants of this approach are introduced that can cope with substitution.

10.1.1.1 Dividing the Execution Model into Blocks of Related States

The states in en execution model can be divided into blocks of related states.

Filter Blocks A *filter block* is the highest level block that is identified in an execution model. All states within a filter block correspond to one filter. Figure 10.1 shows the flow structure between those filter blocks.

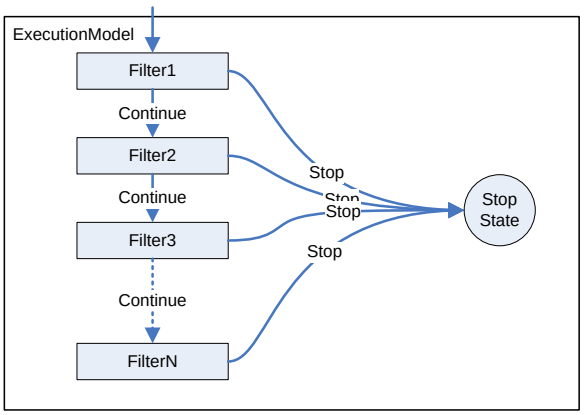


Figure 10.1: Dividing the state space into filter blocks

Flow from one filter block can continue to the next filter or can stop and exit the filter set. For a specific filter block, either one of these outgoing transitions is available or both are available, depending on the filter type. For example, if the filter is a `Before` filter, only the `continue` transition is available. If the filter is a `Dispatch` filter, both transitions are available.

This structure is transformed to code by just placing the code for each filter block in sequence, as can be seen in Listing 10.1. In this way, flow goes naturally through the filter set when a `continue` occurs. To enforce the return or exit behavior, the part of the algorithm that generates the code for the filter block has the requirement that it takes care of exiting this complete code block.

```
//code Filter_1

//code Filter_2

//code Filter_3
...
//code Filter_n
```

Listing 10.1: Generating code for filter blocks

Filter Element Blocks Within a filter block, a second level of blocks can be identified, the *filter element blocks*. All states within a filter element block correspond to one filter element. Figure 10.2 shows the flow structure between the filter element blocks within a filter block.

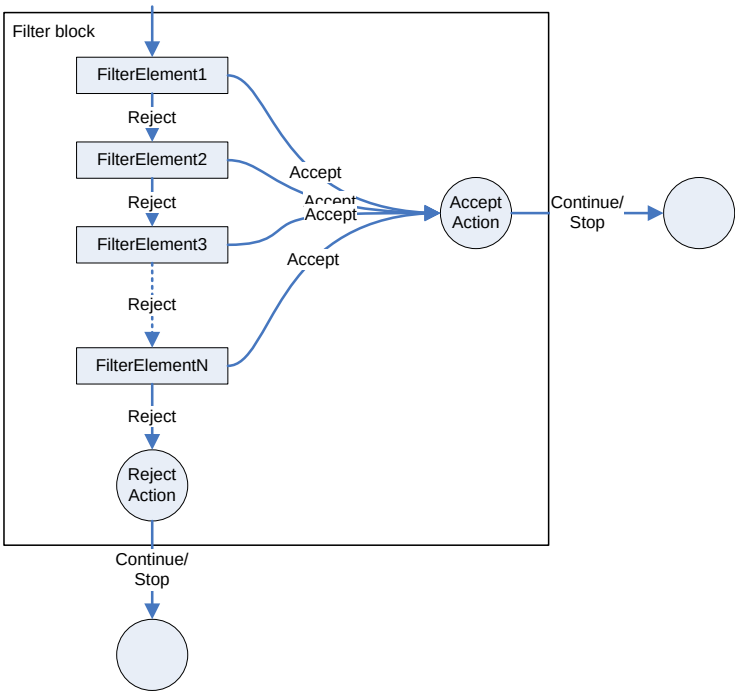


Figure 10.2: Dividing a filter block into filter element blocks

In general, a filter element can accept as well as reject. If a filter element rejects, flow continues to the next filter element. If the last filter element rejects, flow goes to the reject action. If a filter element accepts, flow goes to the accept action.

It should be noted that certain filter element blocks have only one outgoing edge; they always accept or always reject. Remember that a filter element block contains the states from the simulation of the execution of one specific message, not the flow in general. A filter element always accepts if the condition expression is the constant `true` and the matching part always matches for the specific message in the simulation. A filter element always rejects if the condition expression is the constant `false` or if the matching part never matches for the specific message in the simulation.

The flow behavior exiting from both action nodes depends on the type of the action: it either continues or stops.

Within a Filter Element Block Within a filter element block, no more blocks are identified. Figure 10.3 shows the flow between states inside the filter element block.

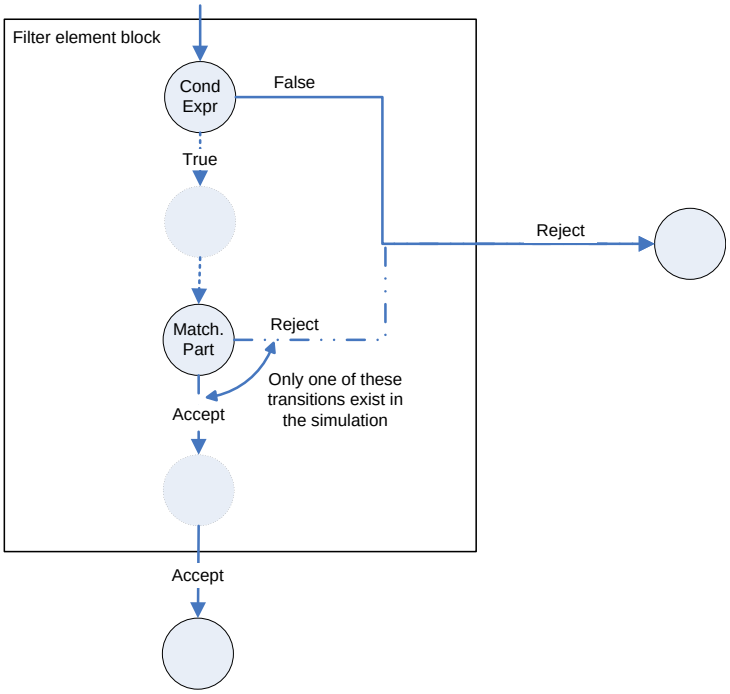


Figure 10.3: States within a filter element block

The first state in the filter element block is the `ConditionExpression` state. This is also the only branching state. If the condition expression evaluates to `false`, the filter element immediately rejects. If the condition expression evaluates to `true`, flow continues sequentially through a number of, for this algorithm less important, states, until the `MatchingPart` state is reached.

The `MatchingPart` state has only one outgoing transition, because message matching has been evaluated by filter reasoning. The evaluation of the message matching leads either to an accept transition or to a reject transition.

If the matching part rejects, the filter element immediately rejects. If the matching part accepts, flow goes through some additional states and eventually leaves the filter element as an accept.

The code generated for a filter element and for the flow between the filter elements depends on the accept and reject behavior of the filter element. The code example in Listing 10.2 shows the generated code for a filter element that can accept as well as reject. If the condition expression evaluates to `true`, the filter element accepts, and so the accept action will occur. If the condition expression evaluates to `false`, the filter element rejects. In this case, flow will either continue to the next filter element or the reject action occurs. So, the code for the next filter element or for the reject action needs to be inlined in the `else` block of the filter element.

```
//FilterElement:
If (ConditionExpression){
    [generated code for the accept action]
}
Else{
    [generated code for next filter element or for the reject action]
}
```

Listing 10.2: Generating code for filter element blocks that might accept as well as reject

Not every filter element might accept as well as reject. If a filter element always accepts, only the code for the accept action is generated. Because this filter element always accepts, flow never continues to a next filter element, so no next filter element needs to be inlined. Note that there might be one that will be reached if we start with another message. Listing 10.3 shows the generated code for a filter element that always accepts.

```
//FilterElement that always accepts:
[generated code for the accept action]
```

Listing 10.3: Generating code for filter element blocks that always accept

If a filter element always rejects, no code is generated other than the code for the next filter element, or the code for the reject action, if there is no next filter element. This is shown in Listing 10.4.

```
//FilterElement that always rejects:
[generated code for next filter element or for the reject action]
```

Listing 10.4: Generating code for filter element blocks that always rejects

10.1.1.2 Problems

The problem with this solution is that it cannot cope with substitution. With substitution it is possible that the filter flow is not a linear structure. There can be different parallel filter blocks for the same filter, as shown in Figure 10.4 (the stop-state is omitted for clarity).

This flow structure cannot be implemented by just placing the filters in sequence. The following section provides a different solution that does not have this problem. In the sections thereafter, variants of the flow based solution are introduced that solve this problem.

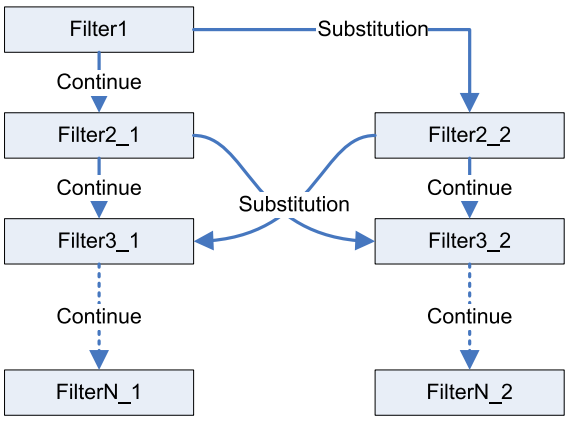


Figure 10.4: Substitution breaks sequential flow

10.1.2 Condition Based Inlining

The second solution does not use the natural flow through the filter set to decide which filter actions are to be executed. It uses the combination of all condition expressions that lead to a certain filter action. Because the flow only branches at condition expressions, we can calculate for each state in the execution model the combined condition expression that leads to this state.

10.1.2.1 The Condition Based Inlining Algorithm

The algorithm is based on four different rules, shown in Figure 10.5.

Start rule

The *start rule* applies to the start state and annotates this state with the condition expression `true`. This means that the execution of the filter set will always reach this state.

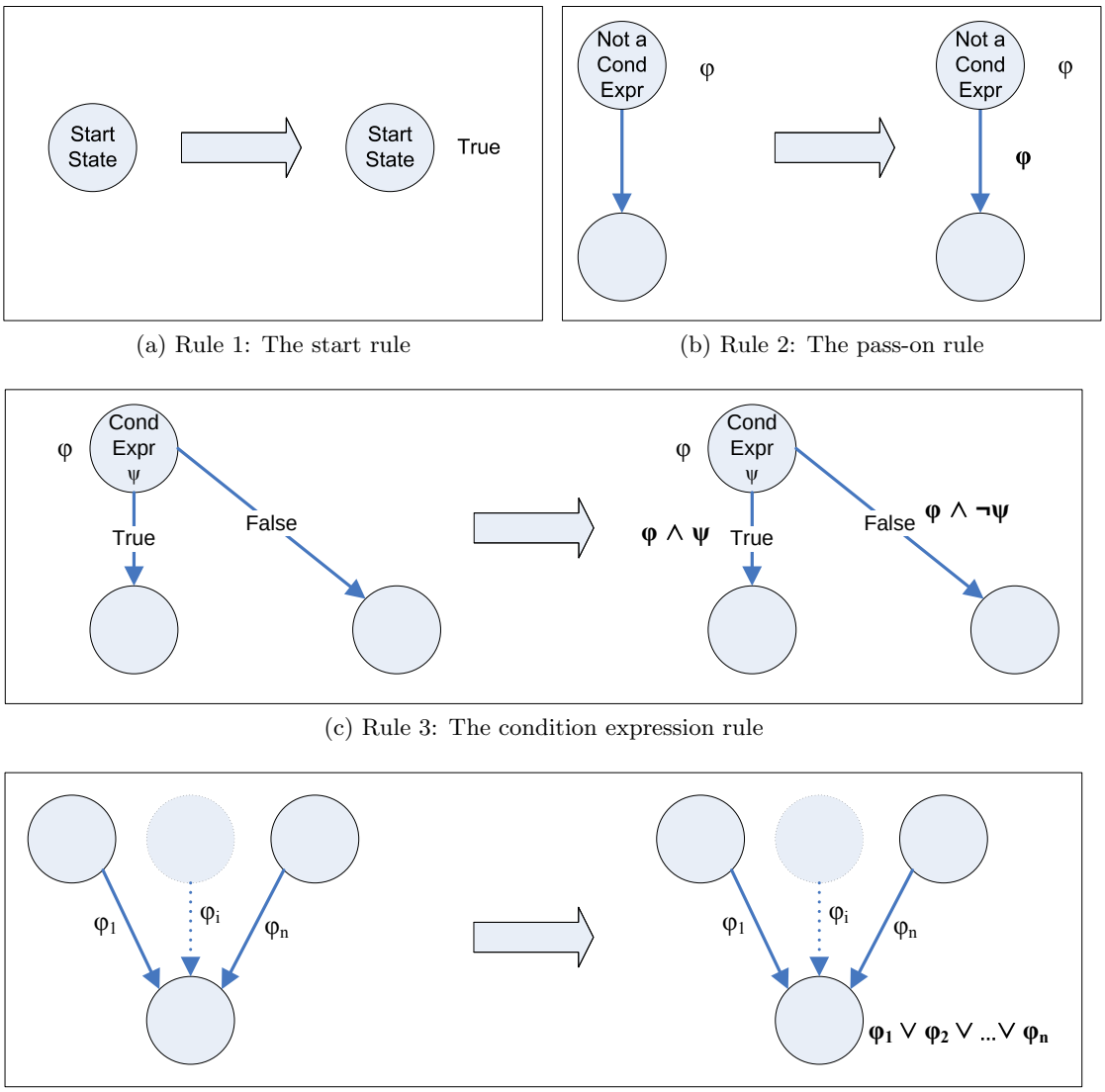
Pass-on rule

If a state is not a `ConditionExpression` state, then it has only one outgoing transition. This transition is reached in an execution if the start state of the transition is reached. Therefore, the transition gets the same annotated condition expression as the start state of the transition. This is specified in the *pass-on rule*.

Condition expression rule

The *condition expression rule* applies if the state is a `ConditionExpression` state. If a state is a `ConditionExpression` state, it has two outgoing transitions: a `true` transition and a `false` transition.

To reach one of the transitions in an execution of the filter set, the state should be reached first. Therefore, the condition expression ϕ annotated to the state should be `true`. The transition taken from the state is based on the value of the condition expression ψ corresponding to the `ConditionExpression` state. If ψ is `true`, the `true` transition is taken. Therefore, the `true` transition is annotated with $\phi \wedge \psi$. If ψ is



(a) Rule 1: The start rule

(b) Rule 2: The pass-on rule

(c) Rule 3: The condition expression rule

(d) Rule 4: The combination rule

Figure 10.5: The condition based inlining rules

`false`, the `false` transition is taken. Therefore, the `false` transition is annotated with $\phi \wedge \neg\psi$.

Combination rule

The *combination rule* combines the annotated condition expressions on all incoming transitions to the condition expression for the state. A state is reachable in an execution if one of its incoming transition is reachable. An incoming transition is reachable, if the condition expression annotated to the incoming transition is `true` . So, the state is reachable if one of the condition expressions annotated to the incoming transitions is `true`. Therefore, the state is annotated with the condition expression that combines the condition expressions from all incoming transitions

with an `or` operator.

10.1.2.2 Generating Code

Code is generated for each filter action by placing the code for that filter action within an *if* statement, which checks whether the condition expression annotated to the filter action state is *true*. The code for each filter action can be placed in sequence, if it is ensured that if a filter action happens before another filter action, the code of the first filter action also comes before the code of the second filter action. This is possible, because there are no cycles in the execution.

10.1.2.3 Problems

One problem with this solution is that the conditions used in the condition expressions cannot change during the evaluation of the filter set. If a condition would change, the control flow in the implementation does not correspond anymore to the message flow through the filter set.

Another problem is that the condition expressions can become quite large, potentially exponential in the size of the state space if there are many branches coming together again. This can reduce performance. Also, for every filter action the condition expression needs to be evaluated. If there is a lot of branching and substitution in the filter set, there might be much more filter action states present then are actually executed in one execution path. This leads to the evaluation of many, potentially large, condition expressions.

Listing 10.5 shows an example of a filter set that leads to condition expressions of exponential size. Listing 10.6 shows the generated code. The problem is caused by the fact that the execution of the filter set after a `Before` filter continues to the next filter, both on an accept as well as on a reject of the `Before` filter. This leads to a doubling of the number of paths to each next filter. This also causes a doubling in the size of the condition expression for the filter action of each next filter. The condition expressions can be simplified, but this is NP-hard in the size of the condition expression, however [29].

```
before1 : Before   = { C1 => [*.*] a.before1 };
before2 : Before   = { C2 => [*.*] a.before2 };
before3 : Before   = { C3 => [*.*] a.before3 };
disp    : Dispatch = { true => <inner.*> inner.*}
```

Listing 10.5: filter set leading to large condition expressions

```
If (C1){
    [Before action before1]
}

If (C2 ∧ (C1 ∨ ¬C1)){
    [Before action before2]
}

If (C3 ∧ ((C2 ∧ (C1 ∨ ¬C1)) ∨ (¬C2 ∧ (C1 ∨ ¬C1)))){
    [Before action before3]
}
```

```
If ((C3∧((C2∧(C1∨¬C1))∨(¬C2∧(C1∨¬C1))))∨(¬C3∧((C2∧(C1∨¬C1))∨(¬C2∧(C1∨¬C1))))){
    [Dispatch action]
}
```

Listing 10.6: Generated code

10.1.3 Flow Based Inlining with Jump Instructions

In the original flow based solution the flow between the filters was implemented as a flat sequential structure where each filter's code followed the previous filter's code. This becomes a problem with substitution, because the flow between filters is not sequential anymore. Substitution leads to different possible messages in the filter set. This leads to different parallel flow traces in the execution model, with substitution as the flow connection between the flow traces, as shown in Figure 10.6. Each possible message in the filter set execution has its own flow trace. Keep in mind that we start with one single message and the other possible messages are only obtained through substitution.

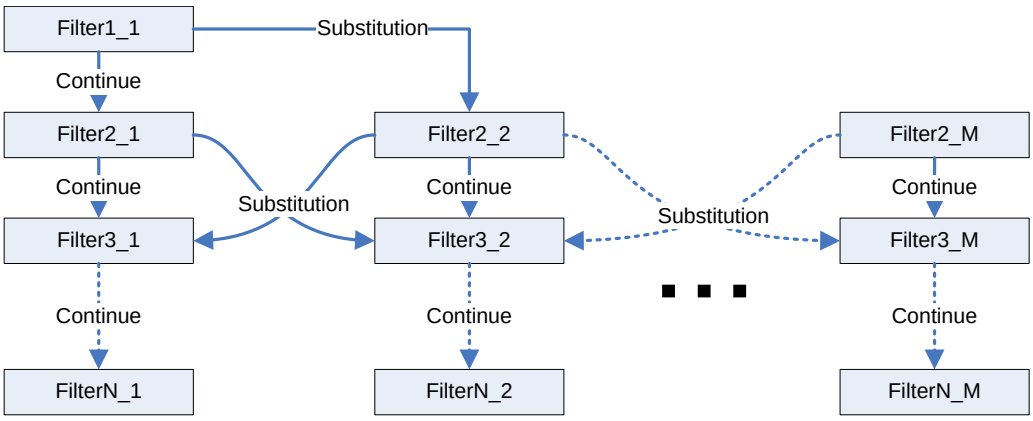


Figure 10.6: Substitution leads to parallel flow traces through the filter set

One solution for this problem is to use `jump` instructions to jump between the different parallel flow traces when a substitution occurs. This can be implemented by just generating each parallel flow as a sequence, placing labels before the code of each filter and jumping to the correct label when a substitution occurs. If we apply this to the example in Figure 10.4 on page 190, we get the code in Listing 10.7.

```
//Filter1
Label Filter1:
...
//SubstitutionAction:
Jump Filter2_2;

//Filter2_1
Label Filter2_1:
...
//SubstitutionAction:
Jump Filter3_2;

//Filter3_1
Label Filter3_1:
[code Filter3_1]

.
.
.

//FilterN_1
Label FilterN_1:
```

```
23 [code FilterN_1]
24 //jump to the end of the filter set:
25 Jump End;
26
27
28 //Filter2_2
29 Label Filter2_2:
30 ...
31 //SubstitutionAction:
32 Jump Filter3_1;.
33
34 //Filter3_2
35 Label Filter3_2:
36 [code Filter3_2]
37
38 .
39 .
40 .
41
42 //FilterN_2
43 Label FilterN_2:
44 [code FilterN_2]
45
46
47 //End
48 Label End:
49 //end of filter set
```

Listing 10.7: Using jump instructions to jump between parallel filter flows

Note that this algorithm creates the filter code for one specific filter module order. If we want the ability to select a filter module order on runtime, the code for each specific filter module order needs to be generated. Also, a decision structure should be generated around it, to select the right filter module ordering and execute the corresponding code.

This algorithm can cope with substitution and does not have the problems the condition based inlining algorithm has. Therefore, this algorithm is the preferred algorithm to implement. Certain procedural languages, however, do not have a `jump` statement. For those languages, another variant of the flow based inlining algorithm has been developed that can cope with substitution but does not have `jump` statements. This algorithm is explained in the next section.

10.1.4 Flow Based Inlining with Message Conditions

In the previous section a solution was presented to cope with substitution. This solution, however, made use of `jump` instructions, a language construct that may not be available in every language. This section presents a variant of the flow based solution that can cope with substitution but uses only high level language constructs.

This solution to the substitution problem is based on attaching a unique number to every flow trace during code generation. In this way, every possible message in the execution of the filter set is given a unique number. This is shown in the example in Figure 10.6, where the flow traces are numbered 1 to M. In every part of the execution we know the current message, so we know the number of the message. We can store this number in a variable. This can now be used before a certain filter block is executed, to determine whether it corresponds to the right message, as shown in Listing 10.8

```
1 Filter3_2:
2 If ( number == 2 ){
3 //filtercode
4 }
```

Listing 10.8: Using message conditions to select between parallel filter flows

To make sure the filters are executed in the correct order, the code has to be generated layer by layer, as shown in Figure 10.7 and Listing 10.9. Instead of using `if` statements, we use a `switch` statement for each layer.

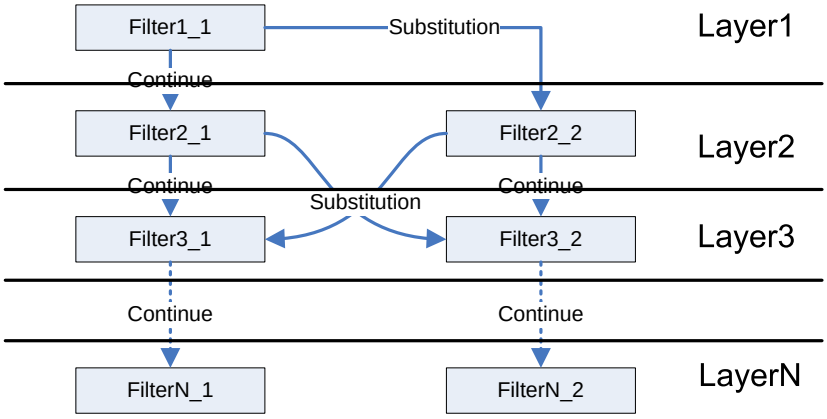


Figure 10.7: Generate code layer by layer to ensure the correct execution order

```
Int number = 1;

//Layer 1
Switch( number ){
//Filter1_1
Case 1:
...
//SubstitutionAction:
  number = 2;
}

//Layer 2
Switch( number ){
//Filter2_1
Case 1:
  ...
//SubstitutionAction:
  number = 2;

//Filter2_2
Case 2:
  ...
//SubstitutionAction:
  number = 1;
}

//Layer 3
Switch( number ){
//Filter3_1
Case 1:
  [code Filter3_1]

//Filter3_2
Case 2:
  [code Filter3_2]
}
.
.
.
//Layer N
Switch( number ){
//FilterN_1
Case 1:
  [code FilterN_1]

//FilterN_2
Case 2:
  [code FilterN_2]
}
```

Listing 10.9: Generate code layer by layer to ensure the correct execution order

The drawback of this solution in comparison with the solution based on `jump` instructions is that we have to execute a `switch` statement before we can execute the right code. With the `jump` instructions we immediately jumped to the right filter block. This is somewhat more efficient.

10.1.5 Incorporating Conditional Superimposition

This section explains how conditional superimposition can be incorporated into the flow based inlining with jump instructions algorithm.

10.1.5.1 Identifying Blocks in the State Space

In the flow based solution, the execution model was divided into filter blocks: blocks of states that belong to one filter. With conditional superimposition there are additional states possible between filter blocks, the `FilterModuleCondition` states. Figure 10.8 shows this in an example. These `FilterModuleCondition` states are treated as separate blocks.

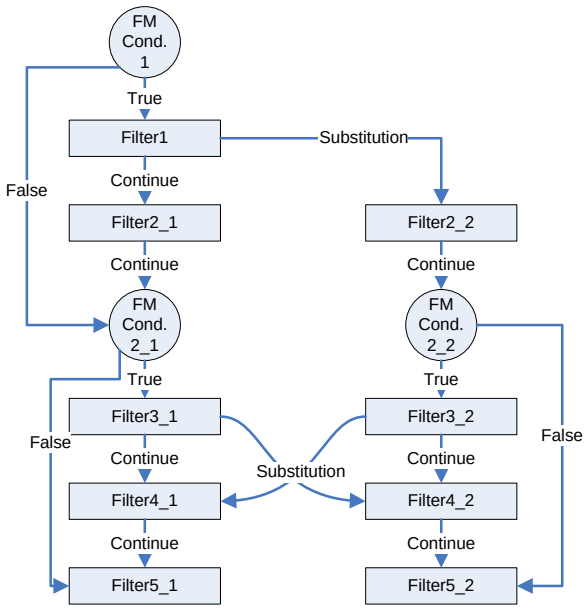


Figure 10.8: Dividing the state space into filter blocks and conditional superimposition states.

10.1.5.2 Generating Code

The code generated for a filter module condition is placed in the same sequence as the code for the filter blocks. It also gets a label, just as the filter blocks. Listing 10.10 shows the code generated for the filter module condition. The code generated is an `if` structure. If the condition is `true`, a jump is done to the true-target filter block. If the condition is false, a jump is done to the false-target. This is either the filter module condition of the next filter module, or the first filter block of the next filter module, if the next filter module is not conditionally superimposed.

```
If (ConditionExpression){
    jump to the true-filter.
}
Else{
    jump to the false-filter or false filter module condition.
}
```

Listing 10.10: Generating code for a `FilterModuleCondition` state

Listing 10.11 shows the code generated for the example in Figure 10.8.

```
//FMCond1
Label FMCond1:
If(condition1) {
  Jump Filter1;
}
Else {
 Jump FMCond2_1
}

//Filter1
Label Filter1:
...
//SubstitutionAction:
Jump Filter2_2;

//Filter2_1
Label Filter2_1:
[code Filter2_1]

//FMCond2_1
Label FMCond2_1:
If(condition2) {
  Jump Filter3_1;
}
Else {
 Jump Filter5_1;
}

//Filter3_1
Label Filter3_1:
...
//SubstitutionAction:
Jump Filter4_2;

//Filter4_1
Label Filter4_1:
[code Filter4_1]

//Filter5_1
Label Filter5_1:
[code Filter5_1]
Jump End;


//Filter2_2
Label Filter2_2:
[code Filter2_2]

//FMCond2_2
Label FMCond2_2:
If(condition2) {
  Jump Filter3_2;
}
Else {
 Jump Filter5_2;
}

//Filter3_2
Label Filter3_2:
...
//SubstitutionAction:
Jump Filter4_1;

//Filter4_2
Label Filter4_2:
[code Filter4_2]

//Filter5_2
Label Filter5_2:
[code Filter5_2]
Jump End;

//End
Label End:
//end of filter set
```

Listing 10.11: Example with conditional superimposition

10.1.6 Weaving Input Filters

After the filter set has been translated to code, the next step is weaving this code in the target program. This section explains how input filters can be woven. The next section explains how output filters can be woven.

Input filters work on the called object, so they should be woven in the corresponding class. The input filters are executed before the called method is executed. So, the filter code for each method should be woven at a position that ensures that it is executed before the original code. It should also work in such a way that when an inner call to the method occurs, the filter code is not executed. Two possible solutions are discussed. The first solution uses method wrapping. The second solution places the filter code directly at the beginning of the method body, followed by the original code.

10.1.6.1 Method Wrapping

With method wrapping, the original body of the method is placed inside a new method, as can be seen in Figure 10.9. The filter code is placed in the body of the original method. When the method is called, the filter code is executed first. An inner call to the method, either from the methods own filter code or from another methods filter code, should be targeted to the new method and not to the original method. This makes sure that the filter code is not executed on an inner call.

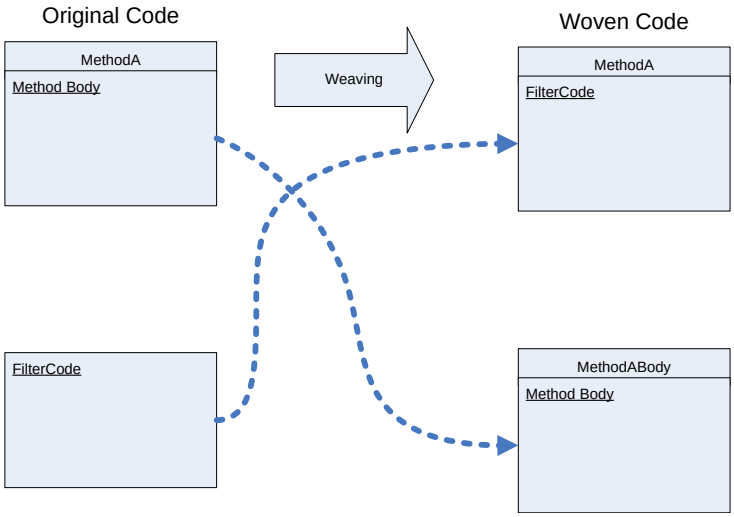


Figure 10.9: Approach 1: Using method wrapping to weave input filters.

Problems Method wrapping has one serious problem. Figure 10.10 shows this problem. This problem occurs when the original method is overridden in a subclass that has no filter set superimposed and the method is called with an inner call from filter code in another method. In this case, the inner call is targeted at the wrapper method. This leads to the execution of the original code instead of the overriding code in the subclass.

10.1.6.2 Filter Code Insertion

The second approach to weave input filters is to insert the filter code directly at the beginning of the method body, followed by the original code, as shown in Figure 10.11. This solution solves the problem that method wrapping has with overriding, but it introduces the problem of how to do inner calls. We have to find a solution to skip the filter code at the beginning of the method body when an inner call is done.

Introducing Filter Context To solve this inner call problem, a filter context is introduced. In the filter context, information that supports the execution of the filter code is maintained. The filter context maintains an inner call flag. This inner call flag can be queried to check whether the call is an inner call. This is shown in Listing 10.12. The filter code is only executed if the call is not an inner call. If the call is an inner call, first the inner call flag is reset. Then, the original method body is executed.

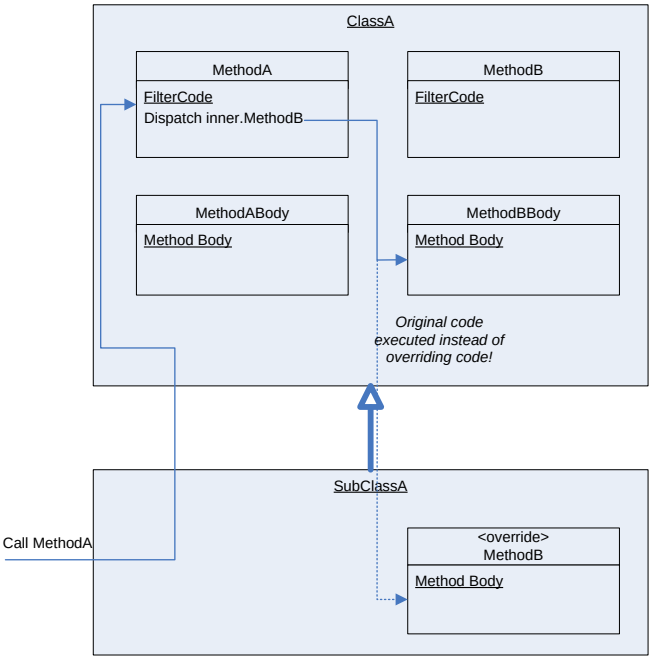


Figure 10.10: Overriding can lead to problems with method wrapping.

Called method:

```
If (!FilterContext.isInnerCall()){
    //filter code
}
FilterContext.resetInnerCall();

//original method body
```

Listing 10.12: Using filter context to cope with inner calls

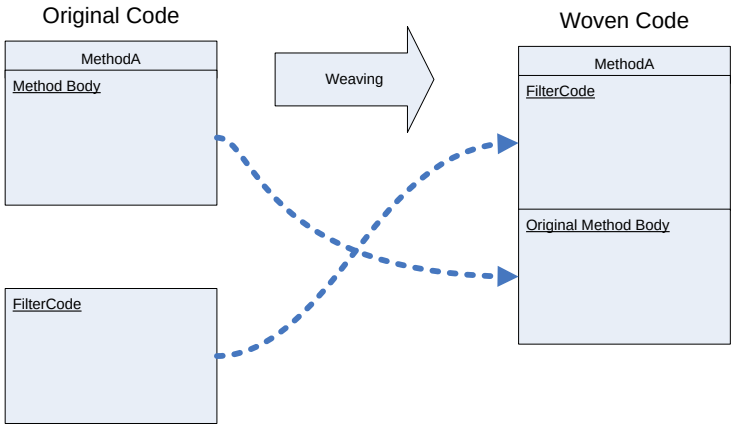


Figure 10.11: Approach 2: Inserting filter code at the beginning of the method body

```
Calling filter code:
FilterContext.setInnerCall();
this.<calledMethod>(..); //the call
```

Listing 10.13: Setting the inner call context

An inner call always originates from filter code. This is, for example, a `Dispatch` action with `inner` as target. If an inner call is executed, the filter code on the calling side is responsible for setting the inner call flag in the filter context. This is shown in Listing 10.13.

If the called method does not have filters inlined, the inner call context should not be set, because it might lead to problems when the called method calls (indirectly) another method that has filters inlined. These filters should be executed, but are not because the inner call context has been set.

At first hand this solution seems to work fine, but it has a problem. If the inlined method is being overridden in a subclass and this overriding method has no inlined filters and does not call the super method, the inner call context is not reset on an inner call, because the overriding method is being executed. This again leads to problems when the overriding method calls another method which has filters inlined, as shown in Figure 10.12

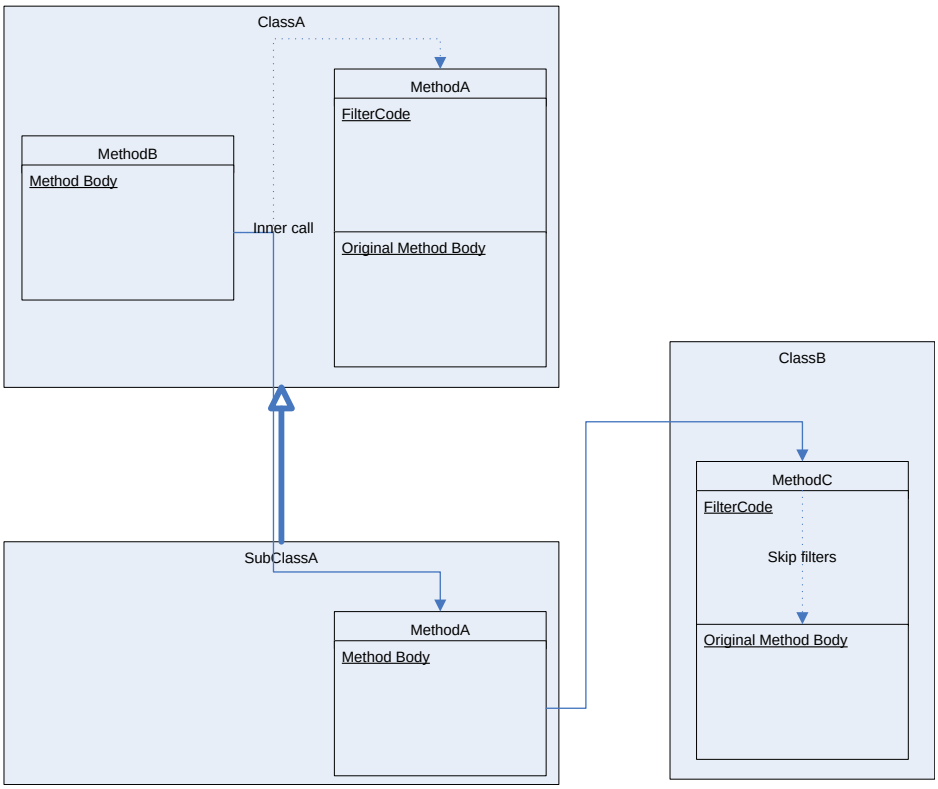


Figure 10.12: Overriding can lead to problems in the inner call context if only a Boolean value is used

One solution to solve this problem is to do a reset of the inner call flag at the start of every method that has no filters inlined. This, however, is not always possible. For example, if an inlined assembly is used in a project that does not use Compose⋆ but just the base language. It also might give unwanted behavior on a super call; the filters are then executed.

Another solution is to add information about the methods signature and the target object for which the inner call context is set. When the inner call context is checked, this information is used to determine whether it is the right method. This solution is used in the implementation.

As a final remark, it should be noticed that the filter context should be thread dependent, to avoid concurrency problems.

10.1.7 Weaving Output Filters

Output filters work on the calling object. So, they should be woven in the corresponding class. The output filters are executed when a call occurs. So, the best place to weave them is at the place where the call originates. The filter code then replaces the original call. This is shown in Figure 10.13.

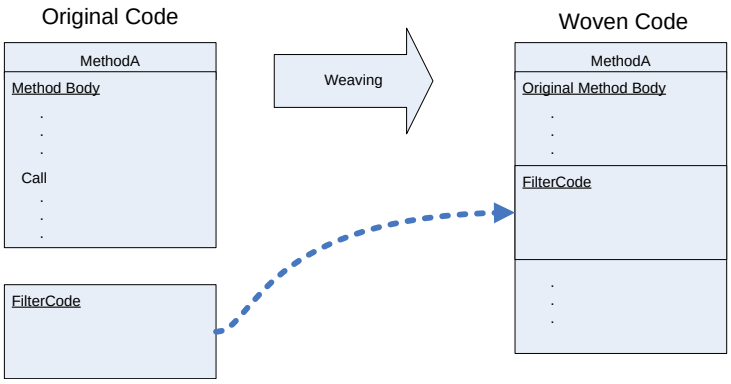


Figure 10.13: Output filter code replaces the original call

With output filters, the inner call flag is not checked. It, however, needs to be set if an inner call is done.

10.2 Design and Implementation

This section describes how filter inlining is designed and implemented in Compose⋆. First, an overview of the inlining process is given. In the sections following the overview, each step in the inlining process is explained. Next, a comparison is made between using a runtime and using filter inlining. Finally, a number of implementation issues are discussed.

10.2.1 Overview of the Inlining Process

Figure 10.14 gives an overview of the inlining process. The process starts with the `FIRE` model of a given filter set. The inlining engine translates the `FIRE` model to a code structure corresponding to flow based inlining with jump instructions (see Section 10.1.3). Conditional superimposition is also incorporated. This code structure is represented as calls to an inlining strategy interface. An implementation of the inlining strategy interface is given in the form of a model builder. This implementation translates the calls to an object model. The abstract instruction model generated by the model builder is given to the emitter which outputs it to an XML file. This XML file is read by the weaver, which transforms the abstract instruction model to actual code and weaves this code into the assemblies.

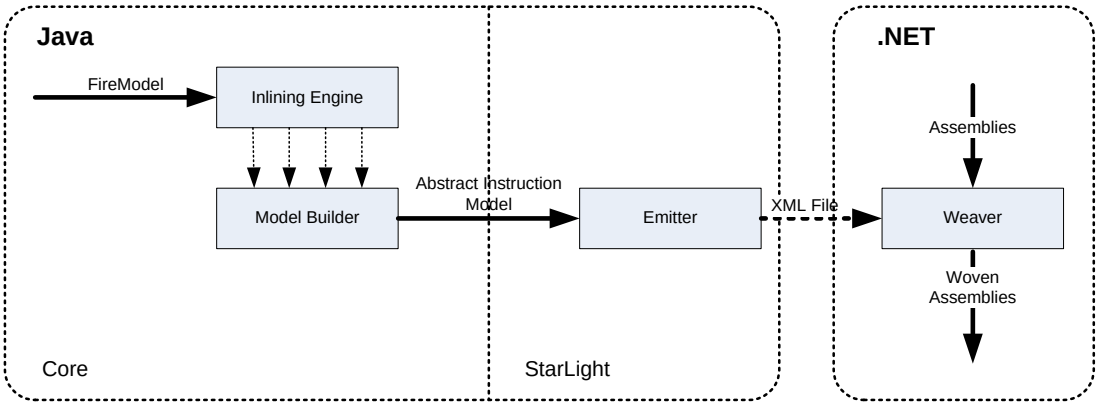


Figure 10.14: An overview of the inlining process

10.2.2 The Abstract Instruction Model

Figure 10.15 shows the class diagram of the abstract instruction model generated by the model builder. These classes are contained in the package `Composestar.Core.INLINE.Model`.

Instruction

The class `Instruction` is the abstract base class for all instructions. It only contains an optional label.

Block

The `Block` instruction is a container for other instructions.

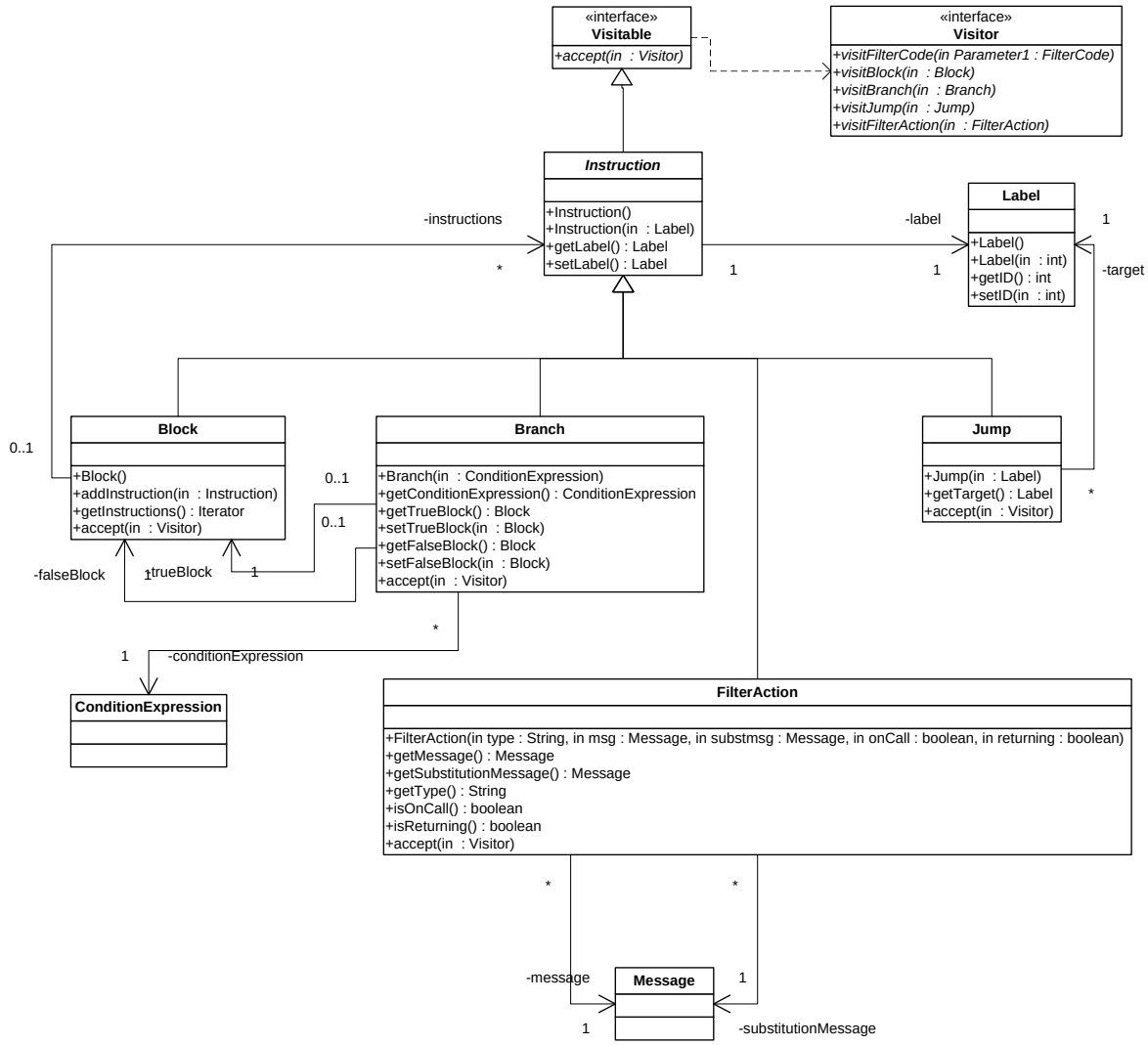


Figure 10.15: The abstract instruction model

Branch

The `Branch` instruction implements branching. Based on the value of the condition expression a branch to either the `true` block or the `false` block should be executed.

Jump

The `Jump` instruction is an instruction to jump to a another instruction. That target instruction of the `Jump` instruction is the instruction that has the label specified as target in the `Jump` instruction.

FilterAction

The `FilterAction` instruction represents a certain filter action that needs to be executed. It only abstractly represents the action; it does not represent how the action should be translated to code. The reason for this is that it is platform specific how a filter action is translated to code. It is the task of the weaver to

generate the code for a filter action. The `FilterAction` class only provides the following information:

- The name of the filter action.
- The message and the substitutionmessage in that specific state of the execution.
- A Boolean value indicating whether the filter action should be executed during the call-flow through the filter set or during the return-flow through the filter set.
- A Boolean value indicating whether the filter action returns the flow.

Label

A `Label` object can be attached to an `Instruction` as the label of that instruction. This label can be used by `Jump` instructions to jump to.

Visitor, Visitable

An implementation of the visitor pattern [14] is given to visit the instructions.

ConditionExpression

The is the standard `ConditionExpression` class from the package `Composestar.Core.CpsProgramRepository.CpsConcern.Filtermodules`.

Message

This is the `Message` class from the `FireModel`.

10.2.3 The Inlining Engine and Model Builder

The *inlining engine* and the *model builder* are responsible for translating a filter set to the abstract instruction model. Figure 10.16 shows the class structure of the inlining engine and the model builder. These classes are contained in package `Composestar.Core.INLINE.Engine`

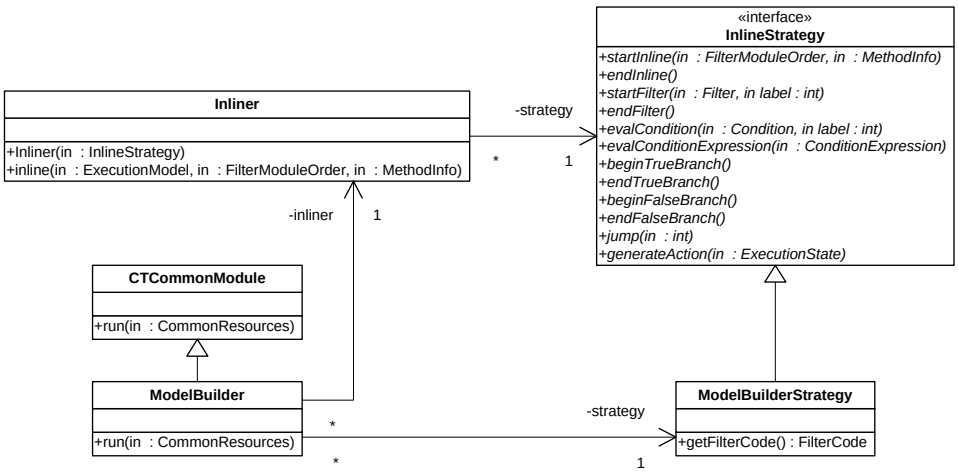


Figure 10.16: Structure of the inlining engine and the model builder.

10.2.3.1 The Inlining Engine

The inlining engine translates the FIRE model to a code structure as described in Section 10.1.3. This code structure is represented as calls to an inlining strategy interface. The strategy pattern is used here to provide flexibility in using the inlining engine. For example, a strategy can do further, possible platform dependent, processing. Also one strategy might use the calls to directly weave the code in the target modules, while another strategy creates an intermediate representation that is being used by other modules before weaving.

10.2.3.2 The Model Builder

The `ModelBuilderStrategy` is an implementation of the inlining strategy interface. This implementation translates the calls to the object model representation of the abstract instruction model. This is a direct mapping. The `ModelBuilderStrategy` does not do any processing.

The `Inliner` and the `ModelBuilderStrategy` are used by the `ModelBuilder`. The `ModelBuilder` is a Compose⋆ module. It iterates over all methods and calls and creates the filter code. The generated abstract instruction model is wrapped in a `FilterCode` object, as shown in Figure 10.17. This `FilterCode` object is attached to the `MethodInfo` or the `CallInfo`. `FilterCode` also contains a number of check conditions. These are the conditions of all conditional superimpositions. They are used for the conditions superimposition efficiency improvement, explained in Section 10.2.5.

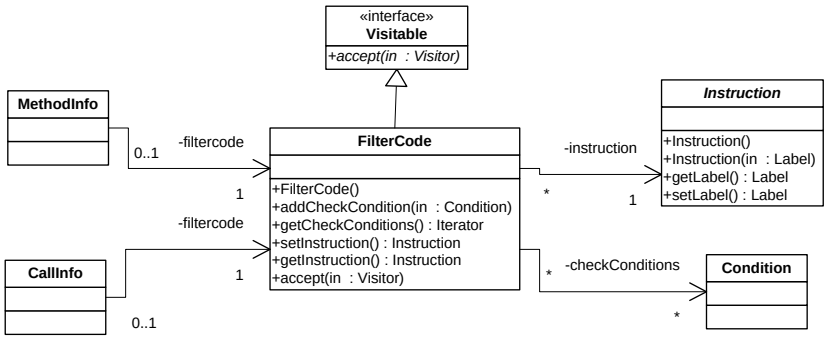


Figure 10.17: FilterCode structure attached to the method or call.

10.2.4 The Emitter: Communication from Java to .NET

The platform independent components of the inlining tool reside in the Java Core module, while the .NET specific parts reside in the .NET Starlight module. Therefore, information needs to be communicated from Java to .NET. This section describes how this communication is implemented.

10.2.4.1 Communication through XML files

The communication between the Java part of the compile time and the .NET part of the compile time is implemented by using XML files. Communication through files

is needed because the different processes are sequential, so the information needs to be stored between the processes. Maintaining the information in files also opens possibilities to reuse the information in an incremental compilation process.

A weave specification is written to the XML files, indicating how the weaver should weave the filters in the different methods and calls. For each assembly a separate weave specification file is generated. In the .NET part, the weave specification is deserialized from the XML file using the standard .NET XML serialization. In the Java part, the Apache XMLBeans package is used to serialize the weave specification to an XML file. This package is used to create an XML serialization object model for Java. This object model is constructed from the XML Schema Definition created by the .NET XML serialization. So, the weave specification object model only needs to be maintained in the .NET part of the compile time; during the compilation of StarLight, the object model for the Java part of the compile time is generated.

10.2.4.2 The Weave Specification Object Model

The weave specification contains the filter code that needs to be woven in the methods and calls. It also contains the internals, externals and conditions used in a concern. Figure 10.18 shows the .NET object model of the weave specification. The generated Java object model is not explained.

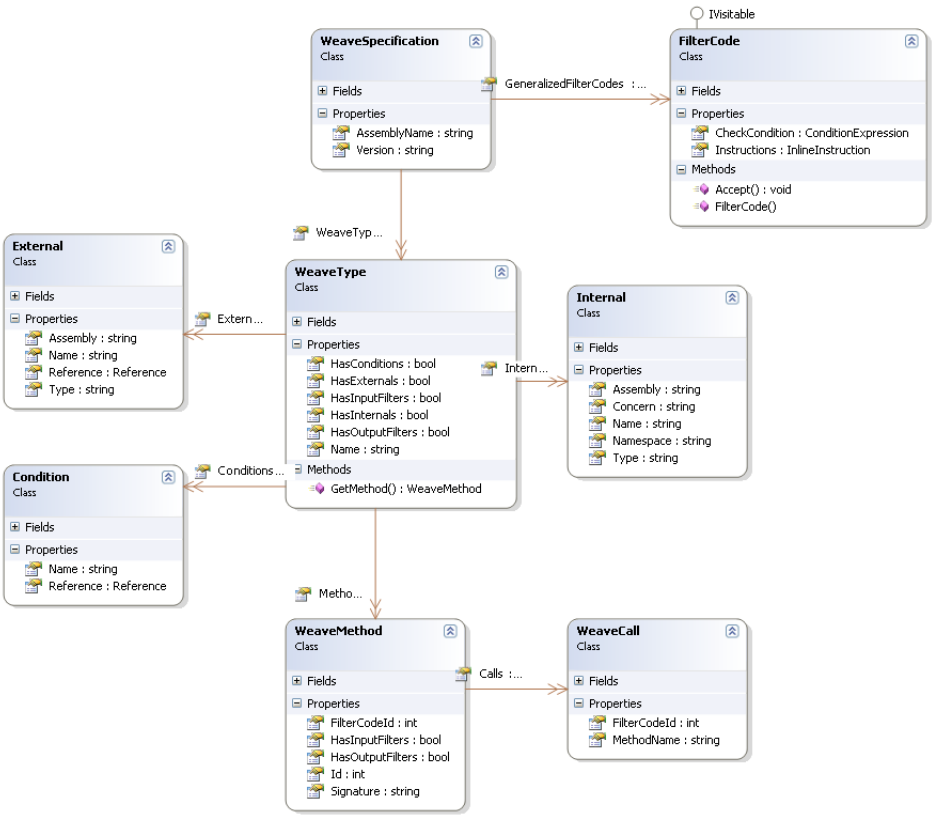


Figure 10.18: The weave specification object model

This model maintains a hierarchical structure similar to the object oriented type structure. The weave specification contains weave types. The weave types contain weave methods. The weave methods contain weave calls. A type, method or call is only present of weaving needs to be performed on itself or one of its child objects.

FilterCode
This is the `FilterCode` class from the abstract instruction model.

WeaveSpecification
This class forms the top-level object of a weave specification. It contains a number of generalized abstract instruction models and the types in the assembly that need to be woven.

WeaveType
This object contains the weave specification for a specific type. It contains information about which externals and internals need to be woven, which conditions are used and which methods need weaving.

WeaveMethod
A `WeaveMethod` object contains the weave specification for a specific method. It contains an id of a generalized abstract instruction model, if input filters need to be woven. It also contains a set of calls to which output filters need to be woven.

WeaveCall
`WeaveCall` contains the weave specification for a specific call. It contains the id of the generalized abstract instruction model that needs to be woven.

External, Internal, Condition
These classes represent externals, internals and conditions. The `External` and `Internal` classes contain information about the type of the external or internal. The `External` and `Condition` classes contain information about how to obtain their values/instances at runtime.

10.2.4.3 Abstract instruction model compression

Certain applications have filter modules superimposed on many classes. For example, to add tracing to the system. If the filter code for each individual method and call is added to the weave specification, the weave specification can become large. This large weave specification can form a bottleneck in the communication between Java and .NET.

To reduce the size of the weave specification, the emitter does not add the filter code to each individual method and call. Instead, it uses a compression mechanism. This mechanism is based on the fact that if there are many methods with input filters or many calls with output filters, the behavior of these messages in the filter set is very similar in many cases. The reason behind this is that there are usually only a few filter modules compared to classes. Also, not every selector is treated differently in a filter module, because otherwise every method should have been explicitly mentioned as a selector in a matching part.

If the behavior of many selectors is similar in the filter set, the resulting abstract instruction models are also similar. Actually, the only difference lies in the selectors of

the current message and substitution message of the filter action instructions. These selectors differ because they are the name of the method or call corresponding with the abstract instruction model. So, if all these selectors that have the name of the method or call are replaced by a placeholder 'generalizing selector', all the abstract instruction models will be exactly the same. Note that selectors that are not equal to the name of the method or call must not be replaced. These selectors are explicitly mentioned in some substitution part and will be the same for all similar abstract instruction models.

The weave specification can now be compressed by not including the abstract instruction model for every method and call. Instead, each unique generalized abstract instruction models is included once. For each method and call it is indicated to which generalized abstract instruction model that method or call corresponds. This can be done by using integer id's, for example. During weaving, the generalizing selectors in the messages in the filter action instructions are replaced by the name of the method or call.

The benefit of this technique varies per application. Applications that have only a few classes with filter modules superimposed do not benefit much from this technique. The more classes there are with filter modules superimposed, the greater the benefit. Especially, applications that add for example logging or tracing to the entire application benefit from this technique. In such a case, almost the same filter code is generated for thousands or millions of methods.

10.2.4.4 GZip Compression

On top of the abstract instruction model compression, GZIP compression is used during the write operation of the xml file. This reduces the amount of disk I/O operations during the communication process, which makes the communication faster.

10.2.5 Weaving the Abstract Instruction Model into Common Intermediate Language Assemblies

This section describes how filter code is woven. It explains how inner call checking is handled and how join point context information is maintained. Next, it describes how the abstract instruction model is translated to IL code, giving special attention to filter actions. Then, it explains how filter actions on return are executed. Finally, it describes an efficiency improvement in combination with conditional superimposition.

10.2.5.1 Inner Call Checking

Inner call checking is handled with an *if* structure, as explained in Section 10.1.6. For this checking an integer id is used that uniquely represents the method within the assembly. An integer is used instead of the method signature because integer comparison is more efficient. Also, the instance object is used in the checking algorithm, if the method is not a static method or within a value type.

If a call is done to inner, it is checked whether the called method has filters woven. If this is the case a `SetInnerCall` operation is woven, using the integer id corresponding with the called method.

The inner call checking is made thread dependent by storing an inner call context object for each thread in a hashmap, using the thread object as the key.

10.2.5.2 Maintaining Join Point Context Information

Certain actions need information about the join point. Therefore, a `JoinPointContext` object is supplied to the action. This `JoinPointContext` object contains the following information:

Method signature
: The method signature of the called method. Initialized at the beginning of the filter code.

Start selector and target
: The selector and target of the original call. Initialized at the beginning of the filter code.

Sender of the message
: For output filters the `JoinPointContext` contains the sender of the message. For input filters the `JoinPointContext` object does not contain the sender, because reflection is needed to obtain it: in .NET the sender cannot be retrieved from the program stack. To obtain the sender, the call stack must be retrieved.

Current selector and target
: The selector and target at the current position in the filter set. Because this can change during the execution of the filter set, it is updated for every filter action. If they do not change in a filter set, this might be optimized by setting them once at the beginning of the filter code.

Substitution selector and target
: If the current filter action is reached through the acceptance of a matching part, these properties contain the result of applying the substitution part on the current selector and target.

Parameter value
: The values of the parameters of the method are maintained. They are initialized at the beginning of the filter code. The value might be changed by the filter action.

Return value
: The return value might be get and set by a filter action.

A `JoinPointContext` object is maintained during one filter set execution. The lifecycle of the `JoinPointContext` object within a filter set execution is as follows:

- Start filter code:
  - Create `JoinPointContext` object;
  - Initialize method signature;
  - Initialize start selector and target;
  - Initialize parameter values.
- Before a filter action:
  - Initialize current/substitution selector and target.

```
condition expression
brtrue Label1
br Label2
Label1:
trueblock
br Label3
Label2:
falseblock
Label3:
```

Listing 10.14: IL structure of the Branch instruction

- During a filter action:
  - Use the `JoinPointContext` object to obtain join point information and to change the value of parameters and return value.
- End filter code:
  - Load the values of the `ref` and `out` parameters from the `JoinPointContext` into the parameters
  - Put the return value on the stack

10.2.5.3 Translating the Abstract Instruction Model to IL Code

To translate the abstract instruction model to code, a visitor is used that traverses the abstract instruction object model. This visitor has methods to translate each type of abstract instruction to code.

Block
: The `Block` instruction is translated by translating each instruction within the block and weave the results sequentially.

Branch
: The `Branch` instruction is translated to a branching structure in IL, as shown in Listing 10.14.

Jump
: A `Jump` instruction is translated to an unconditional branch statement that branches to the code corresponding with the target instruction of the jump.

FilterAction
: The translation of a `FilterAction` instruction depends on the type of the filter action. How this is implemented is explained in Section 10.2.5.4.

10.2.5.4 Translating Filter Actions: A Strategy Based Approach

Filter actions are translated to code by creating an instance of the corresponding `FilterAction` class and calling its `execute` method. For many filter actions this is an appropriate solution. But for certain filter actions it is much more efficient to weave specific IL-instructions than to weave a call to the `execute` method. An example of such a filter action is the `Dispatch` action. If this filter action is implemented using the

`execute` method, the implementation first has to find the method to dispatch to, using reflection and the target, selector and parameter types out of the join point context. Then the actual call is done. But at compile time we can already determine the method to dispatch to. Therefore, we can directly weave the call to the appropriate method instead of weaving a call to the execute method and let the execute method resolve the method to dispatch to, using reflection. This makes the implementation of the `Dispatch` action much more efficient.

So we need a mechanism that gives the ability to define and use a specialized weave function for certain filter actions, while using the default weave function for the other filter actions. To do this we use the `Strategy` pattern. The developer can provide a specific weave strategy for a certain filter action by making an implementation of the `FilterActionWeaveStrategy` interface. This specific strategy is registered by the `Weaver`. When the `Weaver` has to weave a filter action, it searches its weave strategy registry for a specific weave strategy. If such a strategy is present, that strategy is used to execute the weaving. Otherwise, the default weave strategy is used, which just weaves a call to the execute method.

Weave Strategies for Build-in Filter Actions

Continue action
The `Continue` action is the simplest of all filter actions. No code needs to be generated for this action.

Substitution action
If a `Substitution` action occurs, a jump needs to be done, according to the flow based inlining with jump instructions approach. This jump is already present in a separate instruction.

Error action
An `Error` action is translated to throwing an exception.

Dispatch action
A `Dispatch` action is translated as a call to the target method.

Advice action
An `Advice` action is translated as a call to the advice method, specified by the substitution part, with the `JoinPointContext` object as argument.

Skip action
If a `Skip` action occurs, the flow through the filter set is returned. During this returning flow, the filter actions that need to be executed on return are executed. Therefore, the flow is returned by weaving a jump to the 'returning' part of the filter code, instead of weaving a `return` instruction.

10.2.5.5 Storing Filter Actions for Execution on Return

The acceptance or rejection of filters leads to two different filter actions. One filter action that should be executed on call and another filter action that should be executed on return. Because evaluation of the filters happens on the calling flow, the filter action on call can be immediately executed. The execution of the filter action on return, however, should be postponed until the flow returns. Therefore, we need to store this information somewhere.

To do this, we use an action store. This action store is created at the start of the filter code. When a filter action on return is encountered in the filter code, not the corresponding execution code is woven, but a store operation, to store the filter action in the action store for execution on return. This store operation uses an integer id that uniquely indicates the stored filter action.

After weaving the abstract instruction model, extra code is woven to execute the filter actions on return. This code consists of a `while` loop to iterate backward over the stored filter actions and a `switch` operation within the `while` loop to map the integer id to the right filter action. Each `case` in the `switch` contains the execution code of the corresponding filter action. Listing 10.15 shows this in pseudo code.

```
while (actionstore.HasNext())
{
  switch (actionstore.Next())
  {
    case 0:
      //filter action 0
      break;
    case 1:
      //filter action 1
      break;

      ...

    case n:
      //filter action n
      break;
  }
}
```

Listing 10.15: Executing actions on return

When in the normal filter code a filter action is encountered that returns the flow, for example the `Dispatch` action, a jump to this `while` structure is woven to execute the filter actions on return before returning from the method.

10.2.5.6 Conditional Superimposition Efficiency Improvement

If all superimposed filter modules on a concern are conditional (except the default inner dispatcher), then the filter code does nothing but dispatch to `inner` if all conditions are `false`. Therefore, executing the filter code is useless if all conditions are `false`. Efficiency is improved if the filter code is skipped in such a case. Therefore, if all filter modules are conditionally superimposed, an `if` statement is put around the entire filter code. This `if` statement checks whether there is a filter module condition that is `true`. This is shown in Listing 10.16. If there is a `true` filter module condition, the filter code is executed. If there is no `true` filter module condition, the filter code is skipped. The reset of the inner call flag is still executed, to reset a potential inner call. This might also be implemented by doing a reset of the inner call flag in the filter code from which the inner call originated.

```
If(fmCond1 || fmCond2 || ... || fmCondN){
//filtercode
}
```

```
error : Error    = { !doError => [*.*] };
disp  : Dispatch = { true => [*.*] }
```

Listing 10.17: Filter module used for the performance measurements

```
reset innercall flag
//original code
```

Listing 10.16: Executing actions on return

10.2.6 Filter Inlining vs Runtime

This chapter explained filter inlining. Filter inlining is an alternative to using a runtime. This section explores differences between filter inlining and using a runtime with respect to performance, ease of implementation and what functionality can be provided.

Comparing performance To measure the differences in performance between inlining and using a runtime we did an experiment. In this experiment the filter module in Listing 10.17 was superimposed on a test method. This test method implements a sorting routine to sort an array of integers. This method is called 100,000 times, with a list of 100 items to be sorted. The condition in the `Error` filter is designed to produce `false` in half of the occurrences. The experiment is executed five times. Table 10.1 shows the average results of these tests.

	Time (s)	Overhead (%)
Without filters	3.3160	
Inlining	3.8825	17.1
Runtime	128.1306	3764.0

Table 10.1: Performance differences between filter inlining and using a runtime

From these results it can be concluded that filter inlining is much more efficient than using a runtime.

Implementing New Concepts A big advantage of using a runtime is that implementing new concepts is easier. When a developer wants to implement a new concept he only has to change the runtime interpreter. This is written in *J#* which makes it easier to modify. If new concepts are to be implemented using filter inlining, the developer has to think about how these new concepts change the way a filter set is translated to a procedural language structure. He then has to implement this translation, targeting IL, when using .NET. Ensuring that correct IL code is generated is generally harder than writing *J#* code.

Also, debugging of the *J#* code in the runtime is generally easier than debugging the generated IL code.

Increases in Code Size If a runtime is used, the increase in code size is small, compared to filter inlining. With a runtime, only calls to the runtime are woven in the base code. With filter inlining, an entire translation of the filter set execution is woven at each method or call where a filter set is executed. This results in significant increases in code size, especially when filter sets are superimposed on many classes and methods.

Functionality There is no difference in functionality that can be provided between filter inlining and using a runtime interpreter. In both cases we have a Turing complete language to implement the functionality and a weaver that can access any part of the IL code.

Conclusion From this comparison we can conclude that a runtime is preferred in an experimental setting. It provides an environment where new concepts can be easily implemented and tested. In a production environment, filter inlining is the preferred choice, because of its efficiency.

10.2.7 Inlining problems

A number of unresolved issues occurred during the implementation. This section discusses these issues.

10.2.7.1 ref/out arguments and output filters

`Ref` and `out` arguments form a problem when they are used in combination with output filters. At the beginning of the filter code, the argument values are normally stored in the `JoinPointContext` object. For output filters these values are present on the stack and so are obtained one by one from the stack. But for `ref` and `out` arguments not the value is on the stack but a pointer to the value. This forms a problem because we cannot store a pointer in the join point context object, as it is not an object, without making the code unsafe. We also cannot just ignore the pointer or access its value and store this in the `JoinPointContext` object, because we need the pointer for the eventual dispatch and at the end we need to put the resulting value back to the location referenced by the pointer. Because we can only obtain the pointer once from the stack, it will be lost after it has been used. There is a duplicate operation that duplicates the top value on the stack, but because we can only access the top of the stack the pointer will eventually need to be removed to access the entries underneath it.

What about input filters? For input filters this was no problem, because the `ref` and `out` arguments are parameters of the method and so can be accessed multiple times in any order. At the start of the filter code we store the value of the `out` parameters in the `JoinPointContext` object (`ref` parameters do not have an initial value). Before the return we put the values in the `JoinPointContext` object back to the corresponding `out` and `ref` parameters.

10.2.7.2 Meta Filters

The `Meta` action has various multithreading functionality build in. A `Meta` action can, for example, proceed the evaluation of the filter set, where after the control flow is returned to the `Meta` action. This multithreading functionality forms a problem in our filter inlining approach. Because the filter code is woven within the method, doing a `proceed` in a `Meta` action would mean that jumps need to be done into the middle of another methods instructions. This is not possible. In the inlining approach an alternative lightweight variant of the `Meta` action is provided, the `Advice` action. This filter action does not have the multithreading functionality. As with the `Meta` action, the `Advice` action can be used to call an advice method, specified by the substitution part. After the advice method has executed, evaluation of the filter set continues. If we want to implement the `Meta` action functionality of proceeding the execution of the filter set and thereafter returning to the meta method for further proceeding, we can do this by using two `Advice` actions, one on call and one on return.

The `reply` operation in a `Meta` action can be implemented by doing a jump the 'returning' part of the filter code. The `resume` in a `Meta` action can just be implemented as a 'continue' action.

Part IV

Conclusion

Chapter 11

Conclusion

"Whoever in discussion adduces authority
uses not intellect but memory."
Leonardo da Vinci
Italian engineer, painter, & sculptor (1452 - 1519)

This chapter discusses the results of the research presented in this thesis. It also discusses related work and describes future work.

11.1 Discussion

This section discusses the message flow analysis approach presented in this thesis and its applications.

11.1.1 Message Flow Analysis

Message flow analysis provides information at compile time about how specific messages behave in the filter set. There already existed a few approaches to message flow analysis, but they had a number of problems with traceability and efficiency, as was explained in Section 4.4. This thesis presented a new approach to message flow analysis, the message flow simulation approach. We will now discuss to what extend this new approach meets the requirements from Section 4.3.

Efficient Algorithm One of the requirements is that the algorithm has polynomial time complexity. Section 5.4 explained that the reasoning algorithm is worst case $O(\#ASTelements^3)$, where $\#ASTelements$ is the number of elements in the abstract syntax tree, so the size of the filter set. That section also explained that the time complexity is usually more closer to linear in the size of the filter set. In addition, it is linear with the number of classes in the system.

The implementation with GROOVE, however, is less efficient. This is mainly due to the generality of GROOVE's isomorphism checking algorithm. Because the reasoning is only done on a filter module level and the size of a filter module remains usually small, around 4 filters, this is no problem; the combination of the results of different filter modules is close to the theoretical time complexity.

Traceability Another requirement for message flow analysis is traceability: it should be possible to link the reasoning results back to the corresponding parts of the filter set. The existing approaches had problems with the traceability. The presented message flow simulation approach does provide this traceability. Because the complete AST of the filter set is transformed to a flowchart and this flowchart is used in the simulation, each state in the execution model can be linked back to the corresponding AST element.

Fine-Grained Analysis Related to the traceability requirement is the fine-grained analysis requirement. The message flow simulation approach provides fine-grained analysis, because each AST element is used in the simulation algorithm.

Issues As explained in Section 5.5, there are some issues with message flow analysis that this new approach did not solve:

Meta filter uncertainty
: The `Meta` filter can change the message and the flow behavior of the message in a way that cannot be predicted by message flow analysis. On possible solution is to do a semantic analysis of the `Meta` advice.

Target matching on a name level
: Target matching can at compile time only be done on a name level. This might give different results than the instance matching at run time, as explained in.

Message list: exponential state space
: The message list functionality in the reasoning algorithm might make the resulting execution model exponential in size.

11.1.2 Consistency Reasoning

In Chapter 7, filter reasoning is successfully applied to consistency reasoning. The fine-grained message flow analysis provided by filter reasoning makes it possible to do a detailed reachability analysis of the filter set.

11.1.3 Signature Generation

Filter reasoning is also successfully applied in signature generation in Chapter 8. Signature generation itself has also been improved. The old approach to signature generation had a number of problems:

- Certain methods that should be in the signature were not included. The old signature generation algorithm used an approximation technique to generate the signatures. This approximation technique simplified the algorithm, but might lead to certain methods erroneously not being added to the signature.
- Type safety was not checked. Both in the new approach as in the old approach it is possible that methods are included in the signature that might sometimes be dispatched to non-existent target methods. They are included, because they can also be dispatched to an existing target method. The old approach did not check whether all possible dispatch methods are existing.

These problems have been solved in the new signature generation algorithm. This algorithm does not use an approximation of the superset, but generates it by looking at the dispatch structures in the filter set. In this way, the start signatures always contains the methods that should be in the final signatures. Also, type checking has been incorporated in the algorithm and the conflict detection mechanism has been extended.

11.1.4 Behavioral Reasoning

Chapter 9 explained how behavioral reasoning benefits from message flow analysis, because message flow analysis only provides the execution paths that can really occur in practice. In this way, it is prevented that behavioral reasoning finds conflicts on impossible execution paths. Message flow analysis also makes behavioral reasoning more efficient. Originally, behavioral reasoning tried to match a regular expression on each execution path. The problem with this is that there might be exponentially many execution paths. Therefore, we do not try to match the regular expression on each single execution path, but on the entire execution model at once. By transforming the regular expression to a non-deterministic finite automaton and also representing the execution model as a non-deterministic finite automaton, the matching can be done in polynomial time. This gives an answer to the question whether there is an execution path that matches the regular expression, and so violates the constraint.

There is still a problem with checking assertions. Assertions need to match on all execution paths, not just on one. Therefore, the assertion regular expression needs to be complemented. This creates a constraint that can be matched using the given algorithm. The problem is that the constraint might be exponentially larger than the assertion. This means that matching assertions is still worst case exponential in the size of the regular expression but not anymore in the size of the filter set.

11.1.5 Filter Inlining

Filter reasoning makes it possible to inline filters, as explained in Chapter 10. Message flow analysis provides exact information about the behavior of a specific message in the filter set. We can use this information to generate program code that corresponds to this behavior and weave this code in the corresponding method.

An inlining engine has been constructed that translates the filter behavior of a specific method to an abstract instruction model. This abstract instruction model is a platform independent model that resembles procedural programming constructs. This model can be easily translated to code for a specific language. An implementation of a weaver that translates the abstract instruction model to code and weaves this code in the target program has been made for the .NET platform.

The advantage of using filter inlining instead of using an interpreter is performance: filter inlining proved much faster than using a runtime. Disadvantages are that implementing new concepts is harder using filter inlining: there is no framework anymore to implement these constructs. Therefore, filter inlining is preferred in an industrial environment. An interpreter provides advantages in an experimental environment.

11.2 Related Work

This section presents work related to filter reasoning.

11.2.1 Message Flow Analysis

Existing Approaches Bosman developed two approaches to message flow analysis. As Section 4.4 explained, these approaches have a number of problems. More information about these approaches can be read in [5].

Graphs for Modeling Aspect Semantics The message flow simulation approach presented in this thesis is inspired by the work of Tom Staijen on using graph transformations for modeling aspect semantics [48]. Staijen uses graph transformations to generate a model of the execution of an aspect oriented program. This model can be used to analyze the aspect interference in an aspect oriented program.

Control/Data Flow Analysis Message flow analysis is a combination of control flow analysis and data flow analysis. Extensive research has been done on control flow analysis and data flow analysis. A good overview of both control flow analysis and data flow analysis can be found in [35]. Most research on these subjects is targeted at optimizations in procedural languages. Many data flow analysis approaches also use the flowchart as the basis for the analysis, just as our approach.

Semantic Analysis To cope with the `Meta` filter uncertainty, described in Section 5.5, we can do a semantic analysis of the implementation code. The semantic analysis of advice code is just starting to be explored. Work in this area has been done by Van Oudheusen [53].

11.2.2 Consistency Reasoning

Unreachable Code Analysis Consistency reasoning tries to detect reachability conflicts. In this way, consistency reasoning is similar to unreachable code analysis and dead code analysis in procedural languages. Unreachable code analysis tries to find code that is never executed. This is normally done using only control flow analysis [35]. Consistency reasoning uses message flow analysis to find unreachable code. This leads to the detection of unreachable code that would not have been detected with pure control flow analysis.

Dead Code Analysis Dead code analysis tries to find code that is executed but that computes results that are not used anywhere along the execution paths from that code. An example of dead code is the assignment of a variable that is never used. Another example is a statement that computes a value that is never used. Dead code analysis uses data flow analysis [35]. Dead code analysis has similarities with finding redundant parts in a filter set by consistency reasoning. Consistency reasoning finds, for example, redundant filters. Redundant filters are filters that are executed but that not lead to

a filter action, other than the `Continue` action (flow only continues to the next filter). Redundant filters can be left out.

11.2.3 Signature Generation

Signature Generation The signature generation algorithm presented in this thesis is an improvement upon the signature generation algorithms described by Holljen [19] and Bosman [5].

Type Checking Signature generation also does type checking on the dispatch methods. Extensive research has been done on type checking in programming languages. General information about type checking can be found in [38] and [39]. Information about type safety in delegation based languages can be found in [25, 26]. More specific information about type safety in Compose* can be found in [19].

11.2.4 Behavioral Reasoning

Behavioral Reasoning In this thesis we applied filter reasoning to behavioral reasoning. We gave a short introduction to behavioral reasoning. More information about behavioral reasoning about composition filters can be found in [9, 10, 11].

Automata Theory We used automata theory to apply filter reasoning in behavioral reasoning. More information about automata theory can be found in [50].

11.2.5 Filter Inlining

Existing Filter Inlining Approaches Filter inlining has already been implemented earlier. Wichman implemented filter inlining in the ComposeJ tool [57] for Java. Wichman did not use message flow analysis on the complete filter set to translate the filter set to program code. Instead, he translates the filters in the filter set one by one, bottom up. Each filter element in the filter set is translated one by one, from right to left. The code for each next filter element is placed on top of the code of the previous filter element. Also, the code for each next filter is placed on top of the code of the previous filter. To translate a filter element, some sort of message flow analysis is used to evaluate whether the filter element matches for the specific message or not. Although this approach generates code specific for a given message, it might also generate unreachable code. For example, if a message is always dispatched by a `Dispatch` filter early in the filter set, still code is generated for the next filters, although they are unreachable.

The inner call checking implemented in this approach is similar to the inner call checking in our approach. It is also made thread independent. However, only a Boolean value is used to indicate an inner call. As Section 10.1.6 explained, this leads to problems under certain circumstances.

Another implementation of filter inlining has been made for the C platform. More about this implementation can be read in [52].

Weaving in Other Aspect Oriented Languages Many other AOP approaches also use weaving to incorporate aspects into a program. Information about weaving in *AspectJ* can be found in [18]. A powerful framework for load time weaving is provided by JMangler [27].

11.3 Future Work

This section presents suggestions for future work on filter reasoning and its applications.

11.3.1 Message Flow Analysis

More Efficient Implementation using GROOVE to implement the reasoning algorithm made the Filter Reasoning Engine inefficient compared to its theoretical efficiency. The performance is improved by doing the GROOVE analysis only on separate filter modules and combining the results of the different filter modules in a filter set. There is, however, still room for more improvement. It is recommended to improve this efficiency, either by creating a domain specific rule matching algorithm for GROOVE or by using another implementation technique that has the advantages of GROOVE but not the disadvantages.

Message Lists Currently, the Filter Reasoning Engine does not support the new *message list* feature. Incorporating this feature in FIRE might be some future improvement. However, Section 5.5.4 already explained that this might make the execution model exponential in size.

Analyzing Meta Advice Further research might also be done on analyzing `Meta` advice, especially how it influences the message and the flow of the message. This information can be used for filter reasoning. The analysis might be done using the semantic analysis developed by Van Oudheusden [53].

11.3.2 Consistency Reasoning

Primary Conflict Analysis For consistency reasoning a reachability analysis is performed that gives detailed information about the unreachable parts. These unreachable parts indicate possible issues. Also, a cause and effect relationship is identified between the conflicts: all conflicts are caused by a number of primary conflicts. Reporting is done based on these reachability conflicts and the cause and effect relationship between them. But the primary conflicts also have a cause. This cause is not a reachability conflict, but lies in the construction of earlier elements in the filter set.

```
error : Error    = { True ~> [*.a] };
disp  : Dispatch = { [*.a] }
```

Listing 11.1: Filter set that leads to a reachability conflicts in the second filter

Listing 11.1 shows a filter set that leads to consistency conflicts. The consistency reasoning engine detects the primary conflict that the matching part in the `Dispatch` filter never matches. This causes the `Dispatch` filter to always reject. No information is

given about the cause of the primary conflict. The primary conflict is, however, caused by the fact that the `Error` filter always gives an error for messages `a`. This prevents messages `a` from reaching the `Dispatch` filter. Further research might be done about the analysis of the causes of the primary conflicts. Reporting these causes might give better insight to the user about the problems in the filter set.

Error Reporting Section 7.2.4 presented some possibilities to report the consistency conflicts to the user. No extensive research has been done whether the chosen option is really the best option or whether there are other, better options. This might be future work.

11.3.3 Signature Generation

Extended Type Checking Currently, the Signature Generation Engine performs type checking on the target method of a `Dispatch` action. It checks whether the method exists and has the same typing as the entrance method. But type checking might also be done on other actions. For example, for a `Meta` action or an `Advice` action it can be checked whether the advice method exists and has the correct typing. For the StarLight implementation, this is done in the weaver, so type problems are notified to the user. But this type checking might be done earlier in the process, in the Signature Generation Engine.

A possible approach is to make the type checking more general. The type checking of the `Dispatch` action is built-in behavior of the signature generation engine. This can be made more general by not hard coding it, but by specifying it in the properties of filter actions, just as with the flow behavior and the substitution behavior. Possible options for this new property are:

Message typing The typing of the target method must be the same as the originally called method.

Specific typing The target method should have a specific type. This specific type should also be specified in the property. Examples of actions that might use this option are the `Meta` action and the `Advice` action.

No typing No type checking needs to be performed for the filter actions that use this option. An example of an action that uses this option is the `Substitution` action.

11.3.4 Behavioral Reasoning

Other Checking Mechanisms Behavioral reasoning currently uses regular expressions to check whether constraints and assertions are violated. As a future improvement is mentioned that other mechanisms might also be used [11]. When other mechanisms are going to be used, it should be investigated how they can make best use of filter reasoning.

11.3.5 Filter Inlining

More Efficient Join Point Context Currently, the join point context is always completely created at the beginning of the filter code. Often, not every information

provided by the join point context is used. Therefore, the join point context might be optimized by including only the information that is needed at the time that it is needed. Specific semantic analysis of the implementations of the filter actions is needed to identify the necessary information. This might improve the efficiency of the filter code.

11.4 Contributions

This section describes the contributions made by this thesis.

11.4.1 Message Flow Analysis

This thesis presents a new approach to message flow analysis in Chapter 5. Message flow analysis can be used to analyze how a specific message behaves in the filter set. This approach has the following improvements compared to the old approaches:

- The algorithm is more efficient algorithm. This makes the approach more scalable.
- The results of the message flow analysis can be traced back to the corresponding elements in the abstract syntax tree.
- The results of the message flow analysis are fine-grained. The analysis includes each element in the abstract syntax tree.

11.4.2 Consistency Reasoning

Chapter 7 describes how the new message flow analysis approach is applied to consistency reasoning. This makes it possible to do a more fine-grained consistency analysis. Also, a cause and effect relationship between consistency conflicts has been identified.

11.4.3 Signature Generation

Chapter 8 presents a new approach to signature generation. This new approach solves several issues of the old approach to signature generation:

- The signature generation is precise. The old approach made an approximation of the signatures. This leaded to the erroneous exclusion of methods under certain circumstances. The new approach includes all methods in the signature that should be included; no methods are erroneously excluded anymore.
- Type safety is checked: It is possible that methods are included in the signature that might in certain cases be dispatched to a non-existent target method. The new approach checks for all methods in the signature whether all dispatch methods exist. The old approach did not perform this checking.
- A better conflict detection mechanism is provided, that finds the following conflicts:
  - Complete cyclic dependency conflict check: all cyclic dependency conflicts are found, instead of a only a subset of the cyclic dependency conflicts
  - Cyclic dispatch conflicts are detected
  - Infinite signatures are detected

11.4.4 Behavioral Reasoning

Chapter 9 describes how the new message flow analysis approach is applied to behavioral reasoning. This resulted in the following improvements for behavioral reasoning:

- Only the possible execution paths through the filter set are checked for conflicts, instead of all execution paths through the filter set. This leads to less false positives.
- The time complexity of the algorithm is made polynomial: the regular expression representing a conflict is not matched on each individual path, but on the entire execution model (non-deterministic finite automaton) at once. This makes the algorithm linear in the size of the execution model and linear in the size of the regular expression.

11.4.5 Filter Inlining

Chapter 10 describes how message flow analysis is applied to filter inlining. This leads to the following contributions:

- The filter set is translated to code specific for a given message. This code can be woven in the method corresponding to the given message.
- The generated code is a new platform independent abstract instruction model: it can be used for any (procedural) platform.

Part V

Appendices

Appendix A

Motivation Example

A.1 Base system

```
public class MailSystem {
  private final static MailSystem INSTANCE = new MailSystem();

  Connection conn = Connection.getInstance();

  private MailSystem() {}

  public void sendMail(Mail mail) {
    conn.send(mail.getData());
  }

  public Mail receiveMail() {
    return new Mail(conn.receive());
  }
}
```

Listing A.1: The `MailSystem` class

```
public class Connection {
  private final static Connection INSTANCE = new Connection();

  private boolean connected = false;

  private Connection() {}

  public static Connection getInstance() {
    return INSTANCE;
  }

  public boolean isConnected() {
    return connected
  }

  public void connect() {
    ....
    connected = true;
  }

```

```
  public void disconnect() {
    ....
    connected = false;
  }


  public void send(byte[] data) {
    ....
  }

  public byte[] receive() {
    ....
  }
}
```

Listing A.2: The `Connection` class

A.2 LogMail Concern

```
concern LogMail {
  filtermodule LogFM {
    internals
      logger : Logger;
    inputfilters
      before : Before = { [*.sendMail] logger.logSend };
      after  : After  = { [*.receiveMail] logger.logReceipt }
  }

  superimposition {
    selectors
      mailsystem = {C | isClassWithName(C, 'MailSystem')};
    filtermodules
      mailsystem <- logFM;
  }
}
```

Listing A.3: The `LogMail` concern

```
public class Logger {
  public Logger() {...}

  public void logSend(JoinPointContext context) {...}
  public void logReceipt(JoinPointContext context) {...}
}
```

Listing A.4: The `Logger` class

A.3 BufferMail Concern

```
concern BufferMail {
  filtermodule BufferFM {
    externals
      connection : Connection = Connection.getInstance();
      buffer     : MailBuffer = MailBuffer.getInstance();
    conditions
      connected  : connection.isConnected();
```

```
    inputfilters
      disp : Dispatch = { !connected => [*.sendMail] buffer.storeMail,
                          True => [*.sendMail] inner.sendMail}
  }

  filtermodule CheckConnectionFM {
    externals
      buffer : MailBuffer = MailBuffer.getInstance();
    inputfilters
      after : After = { [*.connect] buffer.sendMail}
  }

  superimposition {
    selectors
      mailsystem = {C | isClassWithName(C, 'MailSystem')};
      connection = {C | isClassWithName(C, 'Connection')};
    filtermodules
      mailsystem <- BufferFM;
      connection <- CheckConnectionFM;
  }
}
```

Listing A.5: The `BufferMail` concern

```
public class MailBuffer {
  private final static MailBuffer INSTANCE = new MailBuffer();

  private List buffer = new List();

  private MailBuffer() {}

  public static MailBuffer getInstance() {
    return INSTANCE;
  }

  public void sendMail(Mail mail) {
    List sendBuffer = buffer;
    buffer = new List();
    for (Mail mail : sendBuffer) {
      Connection.getInstance().send(mail.getData());
    }
  }

  public void storeMail(Mail mail) {
    buffer.add(mail);
  }
}
```

Listing A.6: The `MailBuffer` class

A.4 SecureConnection Concern

```
concern SecureConnection {
  filtermodule SecureFM {
    internals
      encryption : Encryption
```

```
    inputfilters
      before : Before   = { [*.send] encryption.encrypt };
      after  : After    = { [*.receive] encryption.decrypt };
      disp   : Dispatch = { [*.setEncryptionKey] encryption.
          setEncryptionKey }
  }

  superimposition {
    selectors
      connection = {C | isClassWithName(C, 'Connection')};
    filtermodules
      connection <- SecureFM;
  }
}
```

Listing A.7: The `SecureConnection` concern

```
public class Encryption {
  public Encryption() {...}

  public void encrypt(JoinPointContext context) {...}
  public void decrypt(JoinPointContext context) {...}
  public void setEncryptionKey(String key) {...}
}
```

Listing A.8: The `Encryption` class

Bibliography

[1] Ada. Ada for the web, 1996. URL `http://www.acm.org/sigada/wg/web_ada/`.

[2] L. Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994. URL `http://trese.cs.utwente.nl/publications/paperinfo/bergmans.phd.pi.top.htm`.

[3] L. Bergmans and M. Akşit. Composing crosscutting concerns using composition filters. *Comm. ACM*, 44(10):51–57, Oct. 2001.

[4] S. R. Boschman. Performing transformations on .NET intermediate language code. Master's thesis, University of Twente, The Netherlands, Aug. 2006.

[5] R. Bosman. Automated reasoning about Composition Filters. Master's thesis, University of Twente, The Netherlands, Nov. 2004.

[6] O. Conradi. Fine-grained join point model in Compose*. Master's thesis, University of Twente, The Netherlands, Aug. 2006.

[7] A. J. de Roo. Towards more robust advice: Message flow analysis in composition filters and its application. Master's thesis, University of Twente, The Netherlands, Mar. 2007.

[8] D. Doornenbal. Analysis and redesign of the Compose* language. Master's thesis, University of Twente, The Netherlands, Oct. 2006.

[9] P. Durr, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In R. Chitchyan, J. Fabry, L. Bergmans, A. Nedos, and A. Rensink, editors, *Proceedings of ADI'06 Aspect, Dependencies, and Interactions Workshop*, pages 10–18. Lancaster University, Lancaster University, Jul 2006.

[10] P. Durr, T. Staijen, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *EIWAS 2005: 2nd European Interactive Workshop on Aspects in Software*, 2005.

[11] P. E. A. Dürr. Detecting semantic conflicts between aspects (in Compose*). Master's thesis, University of Twente, The Netherlands, Apr. 2004.

[12] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Comm. ACM*, 44(10):29–32, Oct. 2001.

[13] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison Wesley, 1995.

[15] M. Glandrup. Extending C++ using the concepts of composition filters. Master's thesis, University of Twente, 1995. URL `http://trese.cs.utwente.nl/publications/paperinfo/glandrup.thesis.pi.top.htm`.

[16] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley and Sons, 2003. ISBN 0471431044.

[17] W. Havinga. Designating join points in Compose* - a predicate-based superimposition language for Compose*. Master's thesis, University of Twente, The Netherlands, May 2005.

[18] E. Hilsdale and J. Hugunin. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-842-3. doi: http://doi.acm.org/10.1145/976270.976276.

[19] F. J. B. Holljen. Compilation and type-safety in the Compose* .NET environment. Master's thesis, University of Twente, The Netherlands, May 2004.

[20] R. L. R. Huisman. Debugging Composition Filters. Master's thesis, University of Twente, The Netherlands, Feb. 2007.

[21] S. H. G. Huttenhuis. Using aspect-oriented programming to solve problems of design pattern implementations. Master's thesis, University of Twente, The Netherlands, 2006. Nov.

[22] E. International. Common language infrastructure (CLI). Standard ECMA-335, ECMA International, 2002. URL `http://www.ecma-international.org/publications/files/ecma-st/Ecma-335.pdf`.

[23] Jython. Jython homepage. URL `http://www.jython.org/`.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[25] G. Kniesel. Type-safe delegation for run-time componen adaptation. In R. Guerraoui, editor, *ECOOP '99—Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 351–366. Springer, jun 1999. ISBN 3-540-66156-5. URL `http://link.springer.de/link/service/series/0558/papers/1628/16280351.pdf`.

[26] G. Kniesel. *Darwin – Dynamic Object-Based Inheritance with Subtyping*. Phd thesis, CS Dept. III, University of Bonn, Germany, 2000. URL `http://roots.iai.uni-bonn.de/research/darwin/downloads/papers/darwinDiss.pdf`.

[27] G. Kniesel, P. Costanza, and M. Austermann. JMangler—A powerful back-end for aspect-oriented programming. In Filman et al. [13], pages 311–342. ISBN 0-321-21976-7.

[28] P. Koopmans. Sina user's guide and reference manual. Technical report, Dept. of Computer Science, University of Twente, 1995. URL `http://trese.cs.utwente.nl/publications/paperinfo/sinaUserguide.pi.top.htm`.

[29] E. J. McCluskey. Minimization of boolean functions. *Bell System Tech. Journal*, 35:1417–1444, nov 1956.

[30] Microsoft Corporation. .NET compact framework - technology overview. Technical report, Microsoft Corporation, 2003. URL `http://msdn.microsoft.com/mobility/prodtechinfo/devtools/netcf/overview/default.aspx`.

[31] Microsoft Corporation. Overview of the .NET framework. Technical report, Microsoft Corporation, 2003. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpovrintroductiontonetframeworksdk.asp`.

[32] Microsoft Corporation. What is the common language specification. Technical report, Microsoft Corporation, 2003. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconwhatiscommonlanguagespecification.asp`.

[33] Microsoft Corporation. What's is .NET? Technical report, Microsoft Corporation, 2005. URL `http://www.microsoft.com/net/basics.mspx`.

[34] W. Minnen. Design of new filter types for data abstraction. Master's thesis, University of Twente, The Netherlands, 2007. To be released.

[35] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN 1-55860-320-4.

[36] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution*. PhD thesis, University of Twente, The Netherlands, June 2006.

[37] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the Hyperspace approach. In M. Akşit, editor, *Software Architectures and Component Technology*. Kluwer Academic Publishers, 2001. ISBN 0-7923-7576-9.

[38] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[39] B. C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[40] A. Popovici, G. Alonso, and T. Gross. Just in time aspects. In M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 100–109. ACM Press, Mar. 2003.

[41] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In G. Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 141–147. ACM Press, Apr. 2002.

[42] J. Prosise. *Programming Microsoft .NET*. Microsoft Press, Redmond, WA, USA, 2002. ISBN 0-7356-1376-1.

[43] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer-Verlag, 2004.

[44] A. Rensink. Time and space issues in the generation of graph transition systems. In *International Workshop on Graph-Based Tools (GraBaTs)*, volume 127 of *Electronic Notes in Theoretical Computer Science*, pages 127–139. Elsevier Science Publishers, 2005.

[45] P. Salinas. Adding systemic crosscutting and super-imposition to Composition Filters. Master's thesis, Vrije Universiteit Brussel, Aug. 2001.

[46] D. R. Spenkelink. Compose* incremental. Master's thesis, University of Twente, The Netherlands, Dec. 2006.

[47] T. Staijen. Towards safe advice: Semantic analysis of advice types in Compose*. Master's thesis, University of Twente, Apr. 2005.

[48] T. Staijen and A. Rensink. A graph-transformation-based semantics for analysing aspect interference. In *Workshop on Graph Computation Models*, Natal, Brazil, 2006.

[49] D. Stutz. The Microsoft shared source CLI implementation. *MSDN Magazine*, 17 (7), July 2002.

[50] T. A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison-Wesley, second edition edition, 1997.

[51] P. Tarr, H. Ossher, S. M. Sutton, Jr., and W. Harrison. N degrees of separation: Multi-dimensional separation of concerns. In Filman et al. [13], pages 37–61. ISBN 0-321-21976-7.

[52] J. W. te Winkel. Bringing Composition Filters to C. Master's thesis, University of Twente, The Netherlands, Dec. 2006.

[53] M. D. W. van Oudheusden. Automatic derivation of semantic properties in .NET. Master's thesis, University of Twente, The Netherlands, Aug. 2006.

[54] C. Vinkes. Superimposition in the Composition Filters model. Master's thesis, University of Twente, The Netherlands, Oct. 2004.

[55] D. Watkins. Handling language interoperability with the Microsoft .NET framework. Technical report, Monash Univeristy, Oct. 2000. URL `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/interopdotnet.asp`.

[56] D. A. Watt. *Programming language concepts and paradigms*. Prentice Hall, 1990.

[57] J. C. Wichman. The development of a preprocessor to facilitate composition filters in the Java language. Master's thesis, University of Twente, 1999. URL `http://trese.cs.utwente.nl/publications/paperinfo/wichman.thesis.pi.top.htm`.