Implementing flexible, extensible composition operators.

Teun van Hemert

June 17, 2013

A dissertation submitted to the University of Twente

for the degree of Master of Science.

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics and Computer Science.

Chair Software Engineering

Supervisors:

Dr. Ing. C.M. Bockisch

S. te Brinke, MSc

Dr. Ir. L.M.J. Bergmans

Abstract

The Co-op/III language lets developers implement their own composition operators. It achieves this by reifying method invocations as messages. These messages can then be manipulated by using bindings, which rewrite messages, and constraints, which let the developer order and control the application of bindings. In this thesis we investigate how we can develop a flexible and extensible execution framework for such composition operators in the Co-op/III language. To achieve this, first the exact requirements for such an implementation are defined, as well as a precise specification of bindings and constraints in Co-op/III. Two prototype implementations are implemented and described in detail, one using only Java constructs, and one using a framework for advanced dispatching. To show that these prototype implementations conform to the specification of bindings and constraints in Co-op/III, test cases are derived from the developed specification and these test cases are used to show that the prototypes have identical semantics. Finally, the prototype implementations are evaluated and compared based on software quality aspects and conformance to the developed requirements. The results of this evaluation show that while both the Java-only approach and the approach using the advanced-dispatching framework can be used to implement the execution framework, there are some shortcomings in the advanced-dispatching framework that limit the usefulness of the framework.

CONTENTS

1. INTRODUCTION

One of the most important design principles in modern software engineering is the *separation of concerns*. A concern in computer programming is a feature or behavior of a computer program. Examples of such concerns are computations, business logic, security, database access, and logging. The idea behind the separation of concerns is that a software engineer should in general only have to focus on one concern at a time, under the assumption that the other concerns 'just work'. In addition, these concerns are often separately developed and evolved . In software engineering, separation of concerns is achieved by decomposing applications into separate entities, often called modules. This concept is referred to as *modularity*.

Programming paradigms and design patterns exist to aid programmers in achieving modularity in computer programs. For example, *object-oriented* programming languages such as C++, C#, and Java support modularity through inheritance, while *aspect-oriented* [5, 12] languages such as AspectJ separate crosscutting concerns into aspects. Examples of crosscutting concerns are logging and security, which tend to scatter throughout an entire application and are hard to separate using object-oriented techniques. Design patterns aid the programmer in separating concerns as well. For example, the model-view-controller pattern [8] separates the concerns of data presentation, data manipulation, and data itself. On the architectural level, concerns can be separated through architectural patterns. For example, Service-oriented architectures [16] separate architectural concerns into services.

The mechanisms introduced in the previous paragraph are called composition operators. Typically, languages support only a limited amount of composition operators, while solutions to software problems (in particular with respect to modularity) will often benefit from multiple, flexible composition operators. While some of these shortcomings can be alleviated by using design patterns, these patterns often require repetitious boilerplate code to implement.

Ideally, composition operators can be defined by programmers themselves. By allowing this, programmers will always have access to the right composi-

tion tools. The goal of this master thesis is to implement a powerful, flexible execution framework for programmer-defined composition operators.

1.1 Background

1.1.1 Co-op

The Co-op approach attempts to address the issue of having only a limited number of composition operators in a programming language. At its core, Co-op is a dynamically typed, object-based language with its syntax derived from Java. Co-op only provides objects and function application (which is also a composition operation). It achieves flexibility in composition operators by reifying function applications as messages. Conceptually this is similar to languages such as SmallTalk. By providing programmers with primitives that let them manipulate these messages, it allows programmers to design and implement their own composition operators. The Co-op approach describes two types of primitives for manipulating these messages: *bindings* and *constraints*. A binding rewrites a message, and a constraint imposes some constraint on a pair of bindings, for example, the order in which they are executed.

There have been several prototype implementations of the Co-op approach. The first prototype implementation of Co-op – Co-op/I – was developed by Havinga [11]. Co-op/I provides expressive, first-class, composable composition operators. A second prototype – Co-op/II – was developed by Te Brinke [18]. Co-op/II builds upon the concepts of Co-op/I and improves some of the core concepts related to message selection and constraints. It also provides additional capabilities such as unifying method invocations and field lookups, access to dynamic message properties, and avoidance of infinite recursion. With these improvements, Co-op/II allows for the implementation of a wide range of composition operators.

1.1.2 ALIA4J

In software engineering, *dynamic dispatch* is the process of selecting and calling a method at run-time. Each place in the execution flow of a computer pro-

gram where such method selection and calling takes place is called a *dispatch site*. Most conventional languages support *single dispatch*. When using single dispatch, the selection of the method that will be called is performed by using one special argument of the method call. Usually this special argument is the object that is the receiver of the method call. This special argument tends to be indicated by using special syntax. Many programming languages, such as the aforementioned C++, Java, and C#, indicate the target object by placing it in front of the method to be called, separated by a dot. Other arguments of the method call are not used when selecting a method.

There are, however, languages that support other forms of dispatching. *Multiple dispatching* [13], for example, uses not only the target object, but also one or more of the other arguments of a method call. *Aspect-oriented programming* [5, 12] lets programmers include additional behavior at dispatch sites, without modifying the code at those dispatch sites. We call dispatching techniques such as multiple dispatch and aspect-oriented programming *advanced dispatching*. The concepts from the Co-op approach can also be considered as a form of advanced dispatching.

Advanced-dispatching languages share various concepts. In a related literature study, various frameworks for advanced-dispatching languages were studied. From this study we concluded that the Advanced-Dispatching Language-Implementation Architecture (ALIA) [19] is a suitable candidate for the implementation of bindings and constraints. ALIA is a language-independent architecture that comprises a meta-model [1] of the common concepts of advanced-dispatching languages, as well as execution environments for instances of the meta-model. ALIA lets developers define dispatching concepts declaratively. It allows for modularization of the implementation of the semantics of dispatching, as well as optimizations. *Advanced-dispatching Language-Implementation Architecture for Java* (ALIA4J) [2] is an implementation of the ALIA architecture for advanced-dispatching languages that extend Java.

1.2 Problem analysis

The Co-op approach, using bindings and constraints, has been shown to be suitable for implementing flexible composition operators [18]. This implementation

was done for the Co-op/II language. Currently, development on the Co-op/III language is in progress. This language does not yet support bindings and constraints, but it should do so in the future. Therefore, it should be possible to easily integrate the binding and constraint implementation we develop for this thesis into Co-op/III. As Co-op/III code is transformed to Java code, the implementation of the binding and constraint mechanism should also use Java.

The performance of the binding and constraint implementation is a relevant issue, as potentially every method invocation and member access in Co-op/III is influenced by it. In Co-op/II the evaluation of bindings and constraints are treated as entirely separate concerns. While conceptually a good idea, implementing them as such, in some cases has a significant negative performance impact. For example, Te Brinke [18] describes the evaluation of a complex function call when using multiple inheritance. Separating binding and constraint evaluation causes the processing of bindings to generate a lot of messages that will be unreachable due to constraint evaluation later on. To optimize this behavior, as well as other potential problems, a large degree of modularity is desirable. A modular system allows for the implementation of optimizations without necessitating change in other parts of the system. These optimizations should be implementable without requiring changes to the Co-op/III code generation process.

Additionally, while the work by Nagy [14] and Te Brinke [18] shows that the constraints used in Co-op/II are sufficient to implement a large range of composition operators, future research might reveal the necessity for other constraints. The implementation of constraints should be sufficiently flexible to support (relatively) easy implementation of new constraints. Similarly, the binding mechanism must be flexible as well. Supporting this flexibility requires the implementation to be sufficiently expressive.

In summary, the implementation should provide:

R1: easy integration with the existing Co-op/III code base.

R2: binding and constraint semantics compatible to those found in Co-op/II.

R3: a reasonably performing implementation of bindings and constraints.

R4: the possibility to optimize the implementation of the bindings and constraints further, without influencing code generation.

R5: the possibility to implement new types of bindings and constraints.

1.3 Approach

We will prototype two implementations of bindings and constraints. The prototypes will both be implemented in Java, but one will be built using only core Java concepts, while the second one will be developed using the ALIA4J framework. Co-op/II was implemented as an interpreter in Haskell. This decision was made because the goal of the implementation was to develop and show the semantics of Co-op/II without being concerned with performance. Functional languages such as Haskell lend themselves very well to the rapid development of interpreters. However, interpreted code in general tends to have (much) worse performance characteristics when compared to compiled code. Message reification in Java can be implemented using aspect-oriented concepts. As ALIA4J makes it easy to implement aspect-oriented concepts, we will be using ALIA4J for message reification in both prototypes.

We will need to ensure that the semantics of the two prototype implementations are the same. To do so, we will develop a precise specification of the semantics of bindings and constraints in Co-op/III. This specification is then used to derive test cases for both prototypes.

One implementation will resemble the implementation of Co-op/II [18], using almost exclusively core Java concepts (the only exception is message interception). The purpose of this implementation is to improve upon the implementation of Co-op/II by removing unnecessary message generation, and to provide a reference implementation of the semantics of Co-op/III bindings and constraints. As this implementation generates a tree of messages, in the remainder of this thesis this implementation will be referred to as the *tree-based prototype*.

For the second prototype we use ALIA4J primitives to implement the bindings and constraints. The concept of message manipulation through bindings and constraints in Co-op is a form of advanced-dispatching. ALIA4J facilitates the implementation of advanced-dispatching languages. Furthermore, it allows for highly optimized implementation of advanced-dispatching concepts. Implementing Coop/III bindings and constraints using ALIA4J might therefore

provide benefits, both when considering ease of implementation and performance. The advantage of this approach is that, since ALIA4J provides primitives for ordering and composition, it should allow for a simple implementation of bindings and constraints that is already optimized. Also, as the performance of ALIA4J improves, so will the performance of this implementation. However, the meta-model of ALIA4J is not expressive enough to completely implement the semantics of bindings and constraints in Co-op/III. We will investigate what is lacking in the ALIA4J meta-model. For the remainder of this thesis, this implementation will be referred to as the *ALIA4J-based prototype*.

The implementations will be compared based on their code quality, extensibility, and modularity. Directly comparing the performance of the implementations is unfeasible, as the implementations are executed using the interpreted execution environment of ALIA4J. This interpreted execution environment performs several extra method calls whenever the program running on top of the execution environment performs a method call. This has a significant negative performance impact on both prototypes. However, the impact is different for each prototype, as it is likely that a large part of the implementation of ALIA4J itself is not subject to this method call overhead.

1.4 Outline

The outline of the remainder of this thesis is as follows:

- Chapter 2 describes the relevant areas of the Co-op/III language. In this chapter we also develop a detailed specification of the semantics of bindings and constraints, and we show how they relate to the work performed by Nagy [14].
- In chapter 3 we discuss the implementations of the prototypes in detail, and elaborate the design decisions that were made.
- Chapter 4 discusses the differences between the the prototype implementations and the consequences of those differences.
- Finally, in chapter 5 we conclude the thesis and discuss the results and future work.

2. Co-op/III

In this chapter we describe the Co-op/III language. After giving a short overview of the language, we will focus on those language features relevant to our problem analysis and solution.

Co-op/III is a programming language that syntactically resembles languages such as C# and Java. Furthermore, it borrows some of its core concepts from such languages. The language is based around the concept of classes encapsulating data and operations (i.e. methods), and objects, which are instances of those classes.

Method calls and field accesses are treated as being the same in Co-op/III. Field accesses are considered either (1) a method call without a return value, and one parameter (setting the value of a field), or (2) a method call with a return value and without parameters (getting the value of a field). In the future when we refer to method calls we will mean both 'ordinary' method calls and field accesses.

Method calls in Co-op are represented by *message sends*. Messages are sent from a *sender* to a *receiver*. Figure 2.1 shows how a simple message is sent. Programmers do not directly access messages in Co-op, instead, messages generated by Co-op can be manipulated using *bindings* and *constraints*. Properties of messages can be made accessible in method bodies using *implicit parameters*. These messages, bindings, constraints, and implicit parameters will be elaborated on in the following sections.

Co-op/III uses classes and objects, just like e.g. Java. However, the core language is class-based, not object-oriented [21], and does not support inheritance

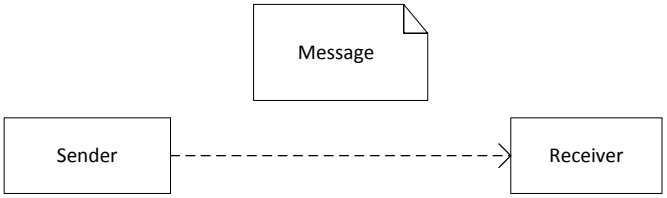

Figure 2.1: Simple message send.

(which we consider a composition operator) as a built-in concept. There are no global variables or methods in Co-op. Also, Co-op/III provides no access modifiers for classes, fields or methods (e.g. protected, private, public). Every member in a class is considered public. Classes in Co-op can have the following members:

Variables are used to define fields, similiar to languages such as Java. Variables can also be local to methods.

Methods are used to define operations on instances of classes, again similar to languages such as Java.

Bindings are used to rewrite messages.

Constraints are used to provide a binary relationship between binding classes[1]. For example, they can be used to provide an execution order between two binding classes.

Variables are used to define fields. Fields in Co-op are scoped per object, there is no concept of class fields. All variables are typed dynamically in Co-op. The reason for this is that composition operators can change the behavior of a Co-op class. For example, composition operators can be used to change which methods can be called on an instance of a class. This makes static type checking inappropriate, as information about these modified classes is not available at compile-time. Therefore, static analysis will either be unable to guarantee type safety, or it will disallow invoking methods that it can not determine to be available at compile-time. A variable definition in Co-op is shown in listing 2.1 on line 2, and assigment to a variable is shown on line 5.

Method definitions in Co-op/III are similar to those in Java. They define a method signature and body. The same scoping rules as in Java apply. Method definitions in Co-op differ slightly from those in Java, as Co-op allows for the definition of *implicit parameters*. These are parameters that are not passed explicitly to a method, but instead are assigned the values of message properties. The *this*-keyword in Java needs to be specified as an implicit parameter in Co-op/III. Lines 4-9 in listing 2.1 show method definitions in Co-op. Implicit parameters are described in more detail in section 2.4.

[1] the difference between bindings, binding classes, and binding instances will be elaborated in section 2.2

Co-op supports annotations on method calls as well. Listing 2.2 shows how method calls are annotated. These annotations become properties of messages. As all built-in operations are method calls (and therefore messages) as well, arithmetic operations such as +, - and * can be annotated as well.

Messages, bindings and constraints will be elaborated on in the following sections.

```
1  class someClass {
2    var someVariable;
3
4    method setSomeVariable[this](someVariable) {
5      this.someVariable = someVariable;
6    }
7    method getSomeVariable[this]() {
8      return this.someVariable;
9    }
10 }
```

Listing 2.1: A simple class definition in Co-op/III

```
1 someVariable.@ParameterlessAnnotation someMethod();
2 someOtherVariable.@ParametrizedAnnotation(parameter) someOtherMethod();
```

Listing 2.2: Annotated method calls

2.1 Messages

In Co-op/III messages are an abstraction of the concept of method invocation. Each method invocation in Co-op/III initially generates one message, that contains information about the sender of the message (i.e. the caller of a method), the target of the message (i.e. the receiver object, as well as the name of the called method), the passed parameters, and a *future*[2]-like object [7] for the return value of the called method. Table 2.1 gives an overview of common message properties.

There are no syntactical elements in Co-op/III just to handle messages, as there is no explicit representation of messages in the language.

[2]In short, a future (or promise) is an object that acts as a proxy for a computation that is not completed yet. Once the computation is complete, the object contains the result of the computation. Futures are commonly used in asynchronous programming.

Property	Description
name	Name of the called method.
target	Target object of the method call.
targetType	Type of the target object.
sender	Source object of the method call.
senderType	Type of the source object.
this	Target object of the method call. This property is used to set the *this* implicit parameter.
thisType	Type of the target object. This property is used when a static method call is performed.
parameters	The parameter values of the method call.
result	The return value of the method call.
message	Annotations applied to the method call.

Table 2.1: Message properties

2.2 Bindings

Bindings are used to rewrite messages. Bindings consist of a *selector* and *rewrite rules*. The selector is used to match a binding to a specific message. The rewrite rules are then used to rewrite certain properties of a message, and generate a *new* message with the rewritten properties.

```
1 binding myBinding = (mySelectorExpression) {
2   // rewrite rules
3 }
```

Listing 2.3: Binding syntax

A bindings is always a member of a class, and, as such, can access any other member of that class. It is important to note that for each instance of a class, a separate instance of a binding defined in that class exists. We call this instance the *binding instance*. When we consider the definitions in listing 2.4, we can distinguish between MyClass.myBinding, which is a *binding class*, and myObject.myBinding, which is a binding instance. The difference is that MyClass.myBinding refers to *all* instances of myBinding, while myObject .myBinding only refers to the particular instance of myBinding that is part of the myObject object.

```
1 class MyClass {
2     // other members
3   binding myBinding = (mySelectorExpression) {
4     // rewrite rules
5   }
6 }
```

```
7
8  // an instance of MyClass
9  var myObject = MyClass.new();
```

Listing 2.4: Binding classes and instances

Binding instances can be activated and deactivated programmaticaly. Listing 2.5 shows this. By default, binding instances are not active, so after definition they have to be activated to have any effect.

```
1  // binding activation
2  myBinding.activate();
3
4  // binding deactivation
5  myBinding.deactivate();
```

Listing 2.5: Binding activation and deactivation

2.2.1 Selectors

Each binding instance uses a *selector* to determine to which messages it applies. Selectors are binary expressions over the properties of messages. Selectors support the binary operations shown in table 2.2. Listing 2.6 gives an example of a selector expression.

matching type	operators	lhs	rhs
lazy	&, |	object	object
normal	==, !=, <, >, <=, >=	object	object
annotation presence	@==, @!=	*message*	annotation matching expression

Table 2.2: Selector operators

```
1  binding myBinding = (name == "a" & message @== @MyAnnotation) {
2    // rewrite rules
3  }
```

Listing 2.6: Binding classes and instances

Selectors can also invoke methods on their containing objects. This can be used to perform a field lookup on the containing object. Note that this will generate a new message as well, and can potentially cause recursive method calls.

2.2.2 Message rewriting

Message properties are rewritten using rules. The rules can be used to either assign a new value to a property, or, in the case of annotations, to add or remove annotations. Assignments always have a message property on the left-hand side and an expression on the right-hand side. The annotation operations always have the keyword *message* on the left-hand side, and an expression that returns an annotation on the right-hand side. Examples of rewrite rules are given in listing 2.7.

```
1 // assignment of a new value to a property
2 messageProperty = newValue;
3 // assignment of the value of an existing message property
4 parameters = message.result;
5
6 // adding an annotation
7 message @+= @MyAnnotation;
8 // removing an annotation
9 message @−= @MyOtherAnnotation;
```

Listing 2.7: Message rewrite rules

Similar to selectors, rewrite rules can call methods on the object containing the binding. The same caveat as with method calls in selectors applies.

2.2.3 Default binding

The default binding is used to perform the final step in delivering a message to its target object. The execution of the default binding for a given message causes the method represented by that message to be executed. The default binding is applicable whenever the current message represents a method that exists on an existing object[3]. The default binding cannot be disabled, and is always active.

[3]Te Brinke [18] elaborates on this in more detail. Summarizing, the default binding selector can be seen as (1) a selector that always is true, but can potentially fail. Essentially this means that there exists only one default binding. The other option (2) is that for each existing method a special selector exists, which only matches on messages that represent that method. There would be a default binding for every method in the program. For us, this difference does not matter, as their effect is identical.

2.3 Constraints

Constraints are used to express the relations between binding classes that apply to the same message. The constraints used in Co-op/III are based on the constraints as implemented in Co-op/II [18]. The constraints in Co-op/II are in turn based on the model of constraints as presented by Nagy [14].

As the constraints of Co-op/III are derived from Nagy's model for constraints at shared join points, we will first discuss Nagy's model. Afterwards, the model used by Co-op/III will be discussed, and differences between the models will be elaborated. It is interesting to see how these models are different, and what the consequences of these differences are.

2.3.1 Nagy's model of constraints

In Nagy's model, constraints apply to actions. Actions represent behavior that is executed at a *join point* [12]. Nagy distinguishes between *action presence* and *action execution*. An action is present at a join point whenever, before the evaluation of any constraints, its behavior will be executed. An action is said to have been executed whenever the behavior it represents has actually been executed. Using constraints, both the presence and execution of actions can be controlled.

Whenever an action in Nagy's model is executed, it may have a Boolean *result* value. These result values are independent of the return type of the action, and indicate if the execution of an action was successful. In this discussion we are only concerned with this result value, and not the return value of an action. When an action does not have a Boolean result value, Nagy considers the result to be *void*. An action that has no result (i.e. the result of the funtion is *void*) is considered to have been executed[4].

Nagy defines three types of constraints:

- Structural constraints specify what actions can be present at a shared join point.
- Ordering constraints specify a partial order of execution of actions.

[4]The absence of a result has some consequences for constraints, which will be discussed later on in this subsection.

- Control constraints specify conditional execution of actions.

Ordering and control constraints are also called *behavioral constraints*. They are similar to the constraint types in Co-op. As Co-op does not support structural constraints, we will not discuss those further.

Nagy's model defines one ordering constraint, the *pre*-constraint. This constraint is used to specify a partial ordering between actions. The semantics are as follows:

pre(x, y) – Actions x and y are executed in an order such that action y is never executed before action x. Therefore, action y must be executed after action x has been executed. This constraint is not transitive.

Control constraints in Nagy's model have the following form: *constraint (condition, constrained action)*. The *condition*-part of the constraint is represented by either a single action, or a Boolean expression composed of actions and logical connectors (*AND*, *OR* and *NOT*). When using a Boolean expression, each action in it has to have a result value of either *true* or *false*, i.e. methods without a result are not allowed in such expressions. Control constraints use the result values of the actions specified in the *condition* to control the execution of the *constrained action*. It is important to note that this automatically specifies a partial ordering for the execution of actions as well, as the result values of the actions in the *condition* must be known (and therefore the actions must be executed) before the *constrained action* can be executed.

Nagy defines two control constraints, the *cond* and the *skip*-constraint:

cond(x, y) - Action y can only be executed if the result value of expression x is true. For this constraint a result value of void is considered to be false, i.e. when the result of x is void, y will not be executed.

skip(x, y, R) - The execution of action y is skipped, and action y is marked as *executed* with result value R, if the result of expression x is true. R substitutes the original return value of y only if y is skipped. This constraint is not transitive.

The ordering and control constraints introduced by Nagy represent *hard* con-

straints. This means that, whenever an action that is part of the *condition*[5] is not *present* at the join point, the execution of the *constrained action* is not allowed. For example, consider a constraint *pre(x, y)*, where *x* and *y* are actions. Then, if action *x* is not present at the join point, the execution of action *y* is prohibited. Conversely, *soft* constraints allow for the absence of an action in the condition of a constraint. To support soft constraints, Nagy defines *soft converter functions*. These functions substitute a return value whenever an action is absent. There are three soft converter functions[6]:

%(action) - if *action* is absent, the result value *void* is substituted. Short notation: %action.

%t(action) - if *action* is absent, the result value *true* is substituted.

%f(action) - if *action* is absent, the result value *false* is substituted.

As an example, consider the constraint *pre(%x, y)*, where *x* and *y* are actions. Now, if action *x* is not present at the join point, the soft converter function will produce a *void* result and action *x* will be considered as executed anyway. Consequently, action *y* is allowed to execute as well. Note that whether a constraint is *hard* or *soft*, presence of the *constrained action* is not required.

When dealing with multiple constraints that apply to the same action, Nagy states that certain precedence rules must be followed. The constraints should be evaluated in the order *pre*, *skip*, *cond*, but it is not made explicit why this ordering is important. However, it can be partially deduced from the semantics of the constraints.

Let us assume that the constraints are *hard*. Futhermore, we know that the *cond* and *skip* constraints require the execution of actions in the condition before the execution of the constrained action, and thus they specify a partial ordering.

Consider the ordering of the *hard skip*, *pre* and *cond*-constraints. The important difference is that the *cond*-constraint can actually prevent an action from having a result value, while the *skip*-constraint ensures that there is a result value, even when the *constrained action* is not actually executed. The implication

[5]We consider the first argument of the *pre*-constraint to be the condition as well, for the purpose of this discussion.

[6]Nagy also defines a hard converter function (#), but as constraints are hard by default in Nagy's model, it is not used explicitly.

of this is that, if we evaluate *cond*-constraints before *skip*-constraints, the *cond*-constraint can 'violate' the hardness of the skip-constraint by removing one or more of the actions in its condition. For example, if we have the constraints *cond(a,b)* and *skip(b,c,F)*[7], then, if *a* evaluates to false, *b* will not be executed at all. When this happens, *b* will have no result value, and the *skip* will fail (i.e. *c* will not be executed and have no result value). When evaluating *skip* before *cond*, this will not be an issue, as a *skip* will always provide a result value for its constrained action. When a *cond*-constraint does not remove its *constrained action*, it can be executed before a *skip* constraint without any problem. The reasoning for the *pre* and *cond* constraints is the same. However, as a *skip* or *pre* can always go before *cond*, and *cond* only sometimes before a *skip* or *pre*, evaluating *skip* and *pre* before *cond* is the most convenient solution.

It is unclear why the *pre* constraint has to be evaluated first. Nagy does not provide a justification for this choice. The algorithms provided by Nagy simply evaluate the *pre*-constraints first, but there is no reason why they cannot be evaluated last.

In summary, there are two reasons for needing precedence rules for constraints in Nagy's model:

N1: Both the ordering and control constraints specify a partial ordering of actions, and the control constraints require this ordering.

N2: The fact that the constraints in the Nagy model are hard, and the cond-constraints can prevent a result value from being generated.

When we also take into consideration soft constraints, the issue with *cond* described above can be resolved by using either the *%t* or *%f* soft converter functions in the *pre* and *skip*-constraints. When *cond*-constraints then remove the presence of actions, their results will be replaced by either *true* or *false*, respectively, when used in a *condition*. Note however, that this may have consequences for the semantics of a program.

[7]We use *T* and *F* to represent *true* and *false*, respectively.

2.3.2 Co-op/III constraints

Now we first discuss the semantics of Co-op/III constraints, and relate them to Nagy's model. Then we will briefly discuss the syntax of constraints in Co-op/III.

Semantics

Each constraint in Co-op/III expresses a binary relation between *binding classes*. Co-op provides two categories of constraints: *ordering* and *control constraints*. As Co-op/III constraints apply to entire binding classes, and not individual instances, we consider the actions here to represent entire binding classes. In contrast to Nagy's model, the actions in the Co-op model do not have a *result* value. In Co-op, we do not need this special result value for actions. As discussed by Te Brinke [18], binding presence and success in Co-op are closely related, and we can consider them identical for the purpose of constraints in Co-op. As a consequence of this, control constraints influence only the application of actions, but do not impose an ordering on the actions, as they do not require the *condition* to have executed to determine a *result* value. This is in contrast to Nagy's model, where control constraints impose a partial ordering as well.

We now discuss the semantics of Co-op/III constraints in detail. We will use the Greek letters α, β, and γ to refer to binding instances, and the Roman letters A, B, and C to refer to the binding classes of these instances, respectively. Furthermore, unless we specify otherwise, we assume every binding instance of a binding class is applicable. Note that, as constraints apply to binding classes, and not to specific binding instances, there is no need for a distinction between constraint classes and instances.

Co-op/III provides the following ordering constraints:

primitive pre(A, B) (short notation: *p_pre*) orders the application of binding instances in such a way that all $\beta \in B$ are applied after all $\alpha \in A$ are applied. This constraint is not transitive. Assume we have defined the constraints *p_pre(A, B)* and *p_pre(B, C)*. Then if α, β, and γ are all applicable, the application order of the bindings would be $\alpha \rightarrow \beta \rightarrow \gamma$. However, if no binding instance $\beta \in B$ is applicable, then the application order of α and

γ is undefined. It could be either $\alpha \rightarrow \gamma$ or $\gamma \rightarrow \alpha$. The semantics of this constraint are very similar to the soft *pre*-constraint of the Nagy model.

pre(A, B) has semantics similar to *p_pre(A, B)*, but is transitive. Assume we have defined the constraints *pre(A, B)* and *pre(B, C)*. Then if α, β, and γ are all applicable, the semantics are the same as those of *p_pre*. However, if no binding β is applicable, the application order is guaranteed to be $\alpha \rightarrow \gamma$.

Control constraints in Co-op/III take the same form as control constraints in Nagy's model. They have the form *constraint(condition, constrained action)*. However, the condition in a Co-op/III is always a single action, and the condition is evaluated based on the presence of that action. This is in contrast to Nagy's control constraints, where a condition can be a Boolean expression. While this makes the constraint mechanism of Co-op/III less expressive, Te Brinke [18] has shown that a large class of well-known composition operators can be expressed using these constraints. Co-op/III provides the following control constraints:

cond(A, B) only allows the application of $\beta \in B$ when $\exists \alpha \in A$ such that α is applicable. This constraint is transitive by definition.

primitive skip(A, B) (short notation: *p_skip*) only allows the application of $\beta \in B$ when $\nexists \alpha \in A$ such that α is applicable. This constraint is not transitive. Assume we have defined constraints *p_skip(A, B)* and *p_skip(B, C)*, then α will be applied, β will be skipped (due to the first constraint), and γ will be applied, because the condition of the second constraint is no longer satisfied. The semantics of this constraint are very similar to the *skip*-constraint of the Nagy model.

skip(A, B) has semantics similar to *p_skip(A, B)*, but is transitive. Assume we have defined constraints *skip(A, B)* and *skip(B, C)*, then α will be applied, and both β and γ will be skipped.

In Co-op/III we use the soft *pre*-constraint. It is not possible to express the Co-op/III *skip* and *cond*-constraints in Nagy's terms of *hard* and *soft*, as the semantics of these constraints are different. Co-op/III constraints depend on the presence of actions, and not the execution result.

Constraints of the same type are always evaluated in topological order. For example, if we declare (in any order) constraints *p_skip(B, C)* and *p_skip(A, B)*,

then the constraint *p_skip(A, B)* will be evaluated before *p_skip(B, C)*, because the condition of the latter constraint depends on the evaluation of the former.

At the end of subsection 2.3.1 we discussed the two reasons why Nagy's model requires a certain order of precedence for its constraints. In Co-op/III, the first reason (N1) is not a factor, as execution order is only specified by the ordering constraint. The control constraints merely depend on binding presence, and not on a result of executing the binding. Similarly, the second reason (N2) is no factor either, as the conditions of control constraints in Co-op/III evaluate if a binding is present, and not its result value.

As ordering and control are entirely decoupled in Co-op/III, the evaluation of precedence constraints can take place before, in-between, and after the evaluation of control constraints, without any semantic difference. Therefore we do not explicitly define when evaluation of precedence rules has to take place.

Syntax

Constraints use the syntax as shown in listing 2.8. In this listing, *constraintType* must be one of the constraints discussed in subsection 2.3.2.

```
1 constraint myConstraint = constraintType(myBindingClass, myOtherBindingClass);
```

Listing 2.8: Constraint syntax

Constraints, analogous to bindings, can be activated and deactivated. The syntax is similar, as listing 2.9 shows. In general, constraints should be activated before the bindings they constrain are activated, as unconstrained bindings might have undesirable side-effects.

```
1 // constraint activation
2 myConstraint.activate();
3
4 // constraint deactivation
5 myConstraint.deactivate();
```

Listing 2.9: Constraint activation and deactivation

2.4 Implicit parameters

Implicit parameters are parameters that are passed implicitly to a method. An example of an implicit parameter is the *this* variable, that is commonly used in programming languages to refer to the object enclosing the current method. In Co-op/III, implicit parameters are used to access message properties. All used implicit parameters in Co-op/III must be defined in the method definition. Listing 2.10 shows the definition of a method *addition* with the implicit parameter *this*.

```
1 method addition[this](var number){
2   //method body
3 }
```

Listing 2.10: Implicit parameter definition

Implicit parameters have the same scope and use as explicit parameters. Listing 2.11 shows how the addition method uses both its explicit and implicit parameter. As can be seen, from the point of view of the method body, there is no difference between using an explicit or implicit parameter.

```
1 method addition[this](var number){
2     this.value = this.value + number.value;
3     return this.value;
4 }
```

Listing 2.11: Implicit parameter usage

The compiler will check if every implicit parameter used in a method body is actually defined. If this is not the case, a compile-time error will be generated.

When a method declares the use of an implicit parameter, the value of the message property with the same name as the implicit parameter is bound to that parameter. No explicit argument is provided for the implicit parameter. The caller of a method does not explicitly provide a value for the parameter. Listing 2.12 shows how a callsite of the method *addition* on some object could look.

```
1 var result = someInteger.addition(3);
```

Listing 2.12: Calling the addition method

When an implicit parameter is defined that does not refer to a message property, and thus no value can be assigned to that implicit parameter, a run-time exception is thrown (as this can only be determined at runtime).

3. IMPLEMENTATION

In this chapter, ALIA4J and the two prototypes of binding and constraint implementations that we developed are described in detail.

We can implement the prototypes entirely separately, or have them implement a common interface. The advantage of using separate implementations is that each implementation can be as expressive as possible. Using a common interface, it is easier to determine if the semantics of the implementations are equivalent. As the latter is one of our goals, we decide that the prototypes share a common interface. Test applications are programmed only against this interface, the specific prototype implementation is hidden from the application. Furthermore, the prototypes share some functionality, such as the classes representing messages and selector expressions. The dependencies between applications and the prototypes are shown in figure 3.1.

The two prototypes take different approaches. The tree-based prototype resembles the approach taken in Co-op/II [18] by generating a tree of messages, recursively applying bindings and constraints to an initially generated message. This leads to a tree with messages that should be deliverable to objects at its leaves. These leaves are then 'executed' by the default binding.

The ALIA4J-based protoype is very close to the conceptual model of Co-op/III, where each applied binding generates a new message dispatch. It uses ALIA4J attachments for its implementation of bindings, and uses the ordering and composition mechanisms of ALIA4J to implement constraints.

The rest of this chapter is structured as follows: in section 3.1 we give a short overview of ALIA4J. In section 3.2 we describe the modules that are shared by both prototypes, and the interface both prototypes implement. Finally, in sections **??** and **??** we discuss the tree-based and ALIA4J-based prototypes respectively.

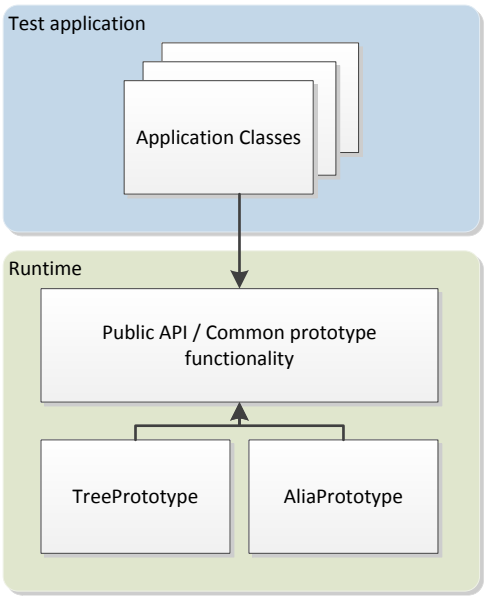

Figure 3.1: Overview of dependencies between prototypes and applications

3.1 ALIA4J

The Advanced-dispatching Language Implementation Architecture for Java (ALIA4J) approach [2] provides a meta-model for dispatching as a high-level intermediate language. It also supplies a framework for execution environments for the intermediate representation, and a visual debugger for programs that use the advanced-dispatching meta-model of ALIA4J.

ALIA4J's metamodel is captured in the Language-Independent Advanced-dispatching Meta-model (LIAM). The entities in LIAM capture the core concepts underlying various dispatching mechanisms in a fine-grained manner. Figure 3.2 gives an overview of the meta-model entities and their relations. Most of the entities in this meta-model are abstract Java classes that can be refined. Refinement of these entities is often necessary when mapping a language to LIAM. When an appropriate default is not available, refinement is required to define language-specific semantics. The only exceptions are the Attachment, Specialization and Predicate entities, since their function is only to group other meta-model entities together. The meta-model entities fulfill the following roles:

Pattern Patterns are used to quantify over the signatures of dispatch sites[1].

[1]Dispatch sites are points in the execution flow of a program where a dispatch is performed. In object-oriented languages this happens when a method on an object is invoked.

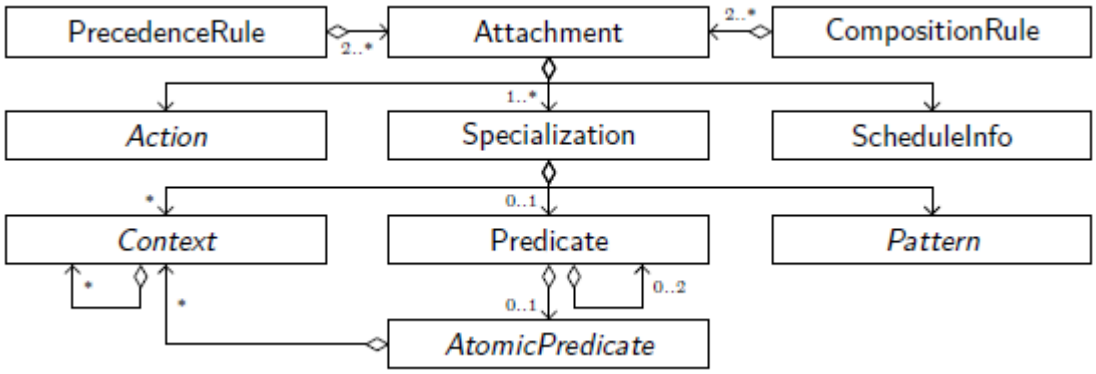

Figure 3.2: The LIAM meta-model of advanced dispatching.

Context The context entity models runtime context that is available during dispatch, such as argument values or the receiver object.

Atomic Predicate Atomic predicates can be used to test the current context during a dispatch. Atomic predicates are parametrized using Context entities.

Predicate Predicates are used to compose multiple atomic predicates together. They are trees where the inner nodes are conjunctions or disjunctions, and the leaves are (possibly negated) atomic predicates. The tree represents a Boolean formula in negation normal form.

Action After evaluating all predicates at a dispatch, the Action entity is used to define the actions that must be performed at a dispatch, such as calling a method or throwing an exception.

Specialization Specializations are used to select specific dispatches by associating Patterns with Predicates. In addition a specialization has a list of Context entities. These Context entities determine which runtime values are to be exposed to the Actions at the selected dispatches.

Attachment and Schedule Information Specializations are associated with an Action by the Attachment meta-model entity. Schedule information determines when the Action should be executed in relation to the dispatch (i.e. before, after, around).

Precedence Rule and Composition Rule The Precedence Rule is used to define a partial ordering between Attachments, to be able to control the execution order of Actions at a shared dispatch. The Composition Rule

defines how Attachments should be composed at a shared dispatch. This can be used to define for example mutual exclusion or overriding.

ALIA4J provides a framework for execution environments, Framework for Implementing Advanced-Dispatching Languages (FIAL), that defines workflows that are common to all execution environments that can execute LIAM models. Based on this framework, ALIA4J provides the following execution environments:

NOIRIn NOIRIn (Non-Optimizing Interpretation-based Reference Implementation) is an interpreter for programs using LIAM models. It replaces every dispatch site in the executing program with an invocation to a callback that evaluates the appropriate attachments.

SiRin SiRIn (Site-based Reference Implementation) is a portable compile-time execution environment that inserts dispatch logic at each dispatch site.

Steamloom^{ALIA} is an extension of the Jikes Research Virtual Machine that reifies dispatches during Just-In-Time compilation. It does not always generate bytecode at each dispatch, but instead it can directly generate (potentially optimized) machine code using the Jikes RVM JIT compiler.

The NOIRIn execution environment is usually the most up-to-date execution environment for ALIA4J. It is the execution framework we use for the development of the prototypes.

3.2 Common prototype features

While both prototypes take a different approach in their implementation, they share a common interface for the Co-op/III language elements. Also, the prototypes share some parts of the implementation. The interfaces and shared elements are grouped together in the *common* package of the prototype. This section discusses the design goals and implementation of the *common* package.

3.2.1 Design goals

While we do not implement actual code generation from Co-op/III code to Java code, we do want to make such generation relatively easy. Therefore, the first design goal for the package is to make mapping Co-op/III language elements to the prototypes easy.

The second design goal is to support testing and comparing the semantics of the implementations, to support our problem statement.

The third design goal of this package is to maximize the reuse of code when possible, while still providing flexibility in the prototype implementations.

3.2.2 Shared interfaces

To support the first two design goals, we provide a set of shared interfaces. They mainly serve to define and create the *bindings* and *constraints* as described in chapter 2, while hiding the prototype implementations. The goal is to ensure that when generating code or creating test cases, no knowledge about these implementations is needed.

This goal is achieved by developing interfaces for operating the bindings and constraints, as well as interfaces for *factories* [8] to obtain implementations of the binding and constraint interfaces. The bindings, constraints, and their factories are contained in the subpackages *common.bindings* and *common.constraints*.

Bindings and the binding factory

To define an appropriate interface for bindings, we have to consider the operations that are applied to bindings from a Co-op/III program. From chapter 2 we know that a binding definition consists of the following:

- a selector expression;
- a set of rewrite rules.

Furthermore, once defined, bindings support the following operations:

- activate;
- deactivate.

Once a binding is defined, the selector expression and rewrite rules cannot be modified anymore. Furthermore, a binding needs both a selector and a set of rewrite rules to be in a consistent state. This makes them likely candidates as parameters for the constructor of a binding. As interfaces in Java do not support the definition of constructors [10], we cannot guarantee that a selector expression is provided when directly creating a binding instance. A unified way to provide a selector expression and rewrite rules must be enabled in a different manner. For the selector expression, we decided to make this a parameter for the factory method to create a binding. We will discuss this in a moment. For the rewrite rules, we moved these to the factory method as well. The rewrite rules can be nicely expressed using a Java Map. The *keys* of the Map represent the properties of the message that are to be rewritten, and the *values* are the expressions for the new values of the properties. However, providing a method that takes a Map of rewrite rules requires the use of some unwieldy Java syntax. Modifying the rewrite rules repeatedly to develop test cases proved to be rather error-prone, so we decided to also add a method to add a single rewrite rule to our interface for bindings. The *activate* and *deactivate* operations are candidates for the interface as well. As activate and deactive are only invoked on existing instances of bindings, they are added to the binding interface. The interface for bindings can be seen in listing 3.1.

```
1 public interface Binding {
2   public Binding activate();
3   public Binding deactivate();
4   public Binding addRewriteRule(final String property,
5                         final UnaryExpression newValue);
6 }
```

Listing 3.1: The Binding interface.

Note that each operation returns an instance of *Binding*. Implementations of this interface must return *this* (i.e. the binding that was modified by the operation). This technique is called *fluent interfaces* [6]. We also use fluent interfaces for the constraint-interface and messages later on. Fluent interfaces lets programmers (or code generators) write more concise and elegant code. An example of it can be seen in listing 3.2

```
1       // arguments omitted for brevity
```

```
2       )
3       .addRewriteRule("name", new ObjectConstantExpression("after"))
4       .addRewriteRule("target", new ObjectConstantExpression(receiver))
5       .activate();
```

Listing 3.2: The fluent Binding interface in action

The implementation of the *BindingFactory* interface is responsible for the creation of new bindings. In addition, it provides access to the default binding. The BindingFactory interface is shown in listing 3.3.

```
1  public interface BindingFactory {
2    public Binding getDefaultBinding();
3    public Binding createRewriteBinding(
4        final String name,
5        final BinaryExpression selector,
6        final CoopObject containingObject);
7    public Binding createRewriteBinding(
8        final String name,
9        final BinaryExpression selector,
10       final Map<String, UnaryExpression> rewriteRules,
11       final CoopObject containingObject);
12 }
```

Listing 3.3: The BindingFactory interface

As described in chapter 2, binding instances are part of a binding class. The *name* argument that is provided is the name of this binding class. As discussed earlier, the selector expression is passed as an argument as well. Furthermore, binding instances often require access to the object that contains them. The *containingObject* argument is used to provide a reference to this containing object to the binding instance.

The *common.bindings* package also contains a class to detect[2] infinite recursion. It maintains a stack of messages currently being processed, and generates a notification whenever a message is generated that was already on the stack. Such an event indicates a likely case of possibly infinite recursion. Prototype implementations can determine what to do with such an event, for example, they can decide to warn the programmer, or stop processing of messages for the specific method call that generated the initial message.

[2]It also supports recursion avoidance.

Constraints and the constraint factory

The motivation for the *Constraint* interface is largely the same as those for bindings. Again we consider which operations we need to support. Constraints always operate on two binding classes, and these binding classes are supplied to a constraint when it is defined. As such, the creation of constraints is best facilitated by a factory. The only operations supported by constraints are similar to that of a binding, i.e. *activate* and *deactivate*. This makes the interface for constraints very simple, as listing 3.4 shows.

```
1 public interface Constraint {
2   public Constraint activate();
3   public Constraint deactivate();
4 }
```

Listing 3.4: The Constraint interface

Co-op/III supports 5 types of constraints, and each of these constraints requires a factory method. The ConstraintFactory-interface is shown in listing 3.5. Each factory method refers to the binding classes that it constrains by name.

```
1   public Constraint createCondConstraint(
2       final String prerequisiteBinding,
3       final String conditionalBinding);
4   public Constraint createPrimitivePreConstraint(
5       final String preceedingBinding,
6       final String preceededBinding);
7   public Constraint createPreConstraint(
8       final String preceedingBinding,
9       final String preceededBinding);
10  public Constraint createPrimitiveSkipConstraint(
11      final String prerequisiteBinding,
12      final String skippableBinding);
13  public Constraint createSkipConstraint(
14      final String prerequisiteBinding,
15      final String skippableBinding);
16 }
```

Listing 3.5: The ConstraintFactory interface

3.2.3 Shared functionality

Aside from the common interfaces, the prototypes share functionality as well. The CoopObject class is necessary to implement the dynamic typing in Co-op/III. The classes in the *common.expressions* package implement selector expressions. The classes in the *common.messaging* package implement the message

interception mechanism and Co-op/III messages. Finally, the *common.graph* package provides the implementation of a directed graph and some operations on such graphs that are used by both prototype implementations.

The CoopObject class

The CoopObject class (contained in the *common* package) is used to simulate dynamic typing in Co-op/III. The Java language is statically typed [10]. However, by 'abusing' the inheritance mechanism in Java, and using the fact that we use code generation[3], we can emulate dynamic typing.

As a statically typed language, Java needs to know at compile-time which methods can be called on a certain type. As discussed in chapter 2, Co-op/III lets programmers modify the methods that can be called on a type at runtime. To overcome this incompatibility, we introduce the CoopObject class. For a given Co-op/III program, this class contains a stub method for *every* method that is called by said program. These stub methods throw a runtime exception when invoked.

Now, by having all classes extend the CoopObject class, overriding just the methods they implement, we can emulate dynamic typing. Listing 3.6 gives an example of this concept. This approach cannot be used to emulate access to instance fields in a dynamic language (as they cannot be overridden). However, as we discussed in chapter 2, we always access instance fields through getters and setters, so this is not an issue. It is now possible to call any method defined in CoopObject on instances of SomeClass and SomeOtherClass, without getting compile-time errors. At runtime, when a class does not actually implement the method, a runtime exception gets thrown and the program will terminate.

```
1  // CoopObject.java
2  public class CoopObject {
3    // stub method
4    public CoopObject someMethod() {
5      throw new RuntimeException("Method not implemented");
6    }
7
8    // stub method
9    public CoopObject someOtherMethod() {
10     throw new RuntimeException("Method not implemented");
11   }
```

[3]The prototypes described in this thesis do not actually use code generation, so some manual labor is necessary to achieve the same effect.

```
}

// SomeClass.java
public class SomeClass extends CoopObject {
  public CoopObject someMethod() {
    // actual implementation
  }
  // someOtherMethod() is not implemented here!
}

// SomeOtherClass.java
public class SomeOtherClass extends CoopObject {
  // someMethod() is not implemented here!
  public CoopObject someOtherMethod() {
    // actual implementation
  }
}
```

Listing 3.6: Emulating dynamic typing using the CoopObject class

Expressions

The *common.expressions* package contains all classes necessary to define selector expressions. Selector expressions are binary expressions that are used to select messages.

We implemented the selector expressions as trees. Each inner node in the tree is a binary node (i.e. it has two children). Binary nodes represent the operators as discussed in subsection 2.2.1. The leaf nodes of the tree represent operands: either message properties or unary expressions (i.e. a constant primitive or composite value, a field lookup, or a message property lookup). Listing 3.7 gives an example of an expression definition.

```
// this represents the following Co−op/III selector expression:
// name == m && targetType == ReceiverA
Expression ex =
  new AndExpression(
    new EqualsExpression(
      new MessagePropertyLookupExpression("name"),
      new ObjectConstantExpression("m")),
    new EqualsExpression(
      new MessagePropertyLookupExpression("targetType"),
      new ObjectConstantExpression(ReceiverA.class)));
```

Listing 3.7: Selector expression definition

Selector expressions are consumed by the prototypes, and the prototypes need a way to evaluate these expressions. Each binary expression implements

the *BinaryExpression* interface (Listing 3.8). The *evaluate*-method defined by this interface requires the implementor to apply the operator represented by the binary node to its subexpressions and return the TernaryValue result.

```
1 public interface BinaryExpression extends VisitableExpression {
2   public TernaryValue evaluate(Message message);
3 }
```

Listing 3.8: The BinaryExpression interface

Similarly, each unary expression implements the *UnaryExpression* interface (Listing 3.9). The *getValue*-method of this method requires the implementor to return the value encapsulated by the node.

```
1 public interface UnaryExpression extends VisitableExpression {
2   public CoopObject getValue(Message message);
3 }
```

Listing 3.9: The UnaryExpression interface

The prototypes might need to apply certain operations (such as transformations) to the expression tree. To support this, the tree implementation supports the visitor pattern [8] through the *VisitableExpression* interface.

Most of these nodes represent the operators that can be used in expressions. Important to note are that there are two types of nodes to perform method invocations:

MethodInvocation can perform a non-reflective method invocation. To do so, an anonymous instance of this abstract class has to be created, and implement the *getValue()* method to non-reflectively call the method on the containing object. By doing so, a new message will be generated, as discussed in chapter 2.

ReflectiveMedhodInvocationExpression can perform a reflective method invocation on the object containing the binding instance. By performing a reflective method invocation, no message will be generated (as ALIA4J does not intercept reflective method calls).

*UnaryExpression*s are also used by the message rewrite rules to perform field lookups and property lookups when rewriting messages.

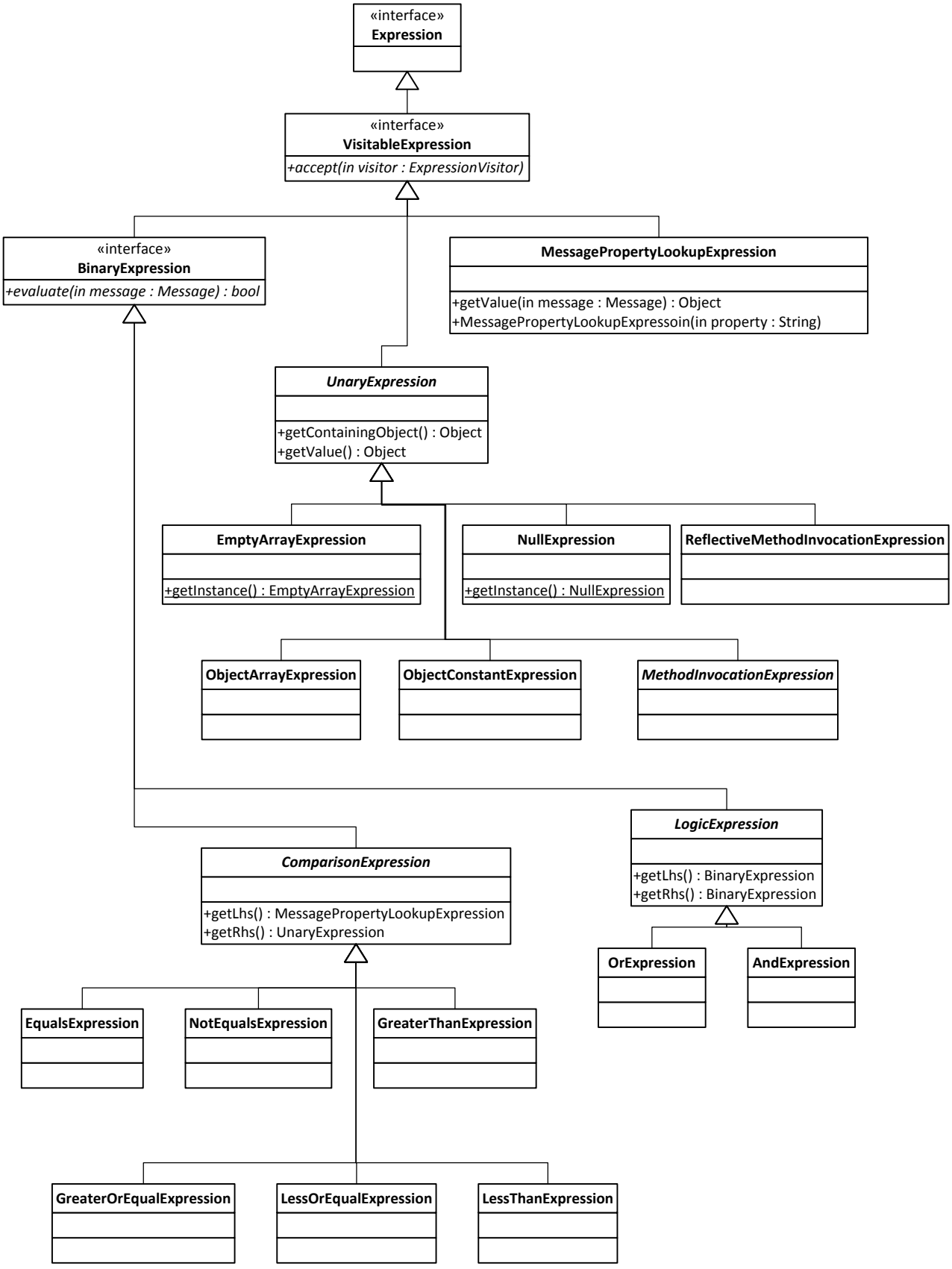

Figure 3.3: Expression class hierarchy.

Messages and message handling

The *common.messaging* package contains the *Message* class, used to represent messages, as well as the *MessageGenerator* class and *MessageHandler* interface that are used to generate the messages.

The *Message* class uses a Map to implement message properties. It provides methods to retrieve the value of a property, to rewrite properties, and to remove properties. Adding a property to a message is done by performing a rewrite on a non-existant property.

The interface for the Message class is shown in listing 3.10. Note that the interface is fluent again. Also, instances of message are immutable. The message class has a private constructor. To create a new message, the empty message is 'rewritten', resulting in a new message with the desired properties. An example of message creation is shown in listing 3.11.

```
1  public final class Message {
2    // retrieve the empty message
3    public static final Message empty();
4
5    public Object readProperty(final String name);
6    public Message rewriteProperty(final String name, final Object newValue);
7    public Message removeProperty(final String name);
8
9    public boolean hasProperty(final String name);
10   public Map<String,Object> getProperties();
11 }
```

Listing 3.10: The Message class

```
1  // create a message with the properties 'name=m' and 'targetType=SomeClass'
2  Message myMessage = Message.empty()
3    .rewriteProperty("name", "m")
4    .rewriteProperty("targetType", SomeClass.class);
```

Listing 3.11: Message creation

```
1  public interface MessageHandler {
2    public void handle(Message message);
3  }
```

Listing 3.12: Message handler interface

Message creation is only performed by an instance of the *MessageGenerator* class. The purpose of this class is to perform message reification. It intercepts every method call to instances of CoopObject and its subtypes. For each method

call, it creates an instance of the *Message* class with the properties as discussed in section 2.1.

For each method call, an empty *ResultWrapper* object is created and set as the *result* property of the message. The MessageGenerator maintains a reference to this *ResultWrapper* object. This is necessary to be able to return the result of the method call to the caller when the message has been processed.

The *MessageGenerator* passes the message on to the *handle* method of the *MessageHandler* implementation that was passed to the *MessageGenerator* when it was constructed. The *MessageHandler* interface is shown in listing 3.12. The *MessageHandler* then further processes the message. Once the *MessageHandler* is done processing the message, it will return, and the *ResultWrapper* will contain the final return value of the method call. This return value is then returned to the caller.

In the prototype this interception is done using an ALIA4J attachment (see section 3.1 for details). In this case the choice for ALIA4J for message reification was done out of convenience: ALIA4J is suitable for the task, and we are already using it for one of our prototype implementations. Message reification could also be performed using other tools or frameworks, such as AspectJ [20]. A more mature framework, that is not performing interpretation, could provide performance benefits.

Graphs

The *common.graph* package provides the implementation of a directed graph. The graph package supports a graph with vertices containing Java objects, and both labeled and unlabeled edges. It also supports the calculation of the transitive closure of a graph, using the Floyd-Warshall algorithm [4].

3.3 Tree-based prototype

The tree-based prototype exclusively uses Java constructs to implement bindings and constraints. It does so by, for every method call, generate a tree of messages. This tree is manipulated using implementations of the visitor pat-

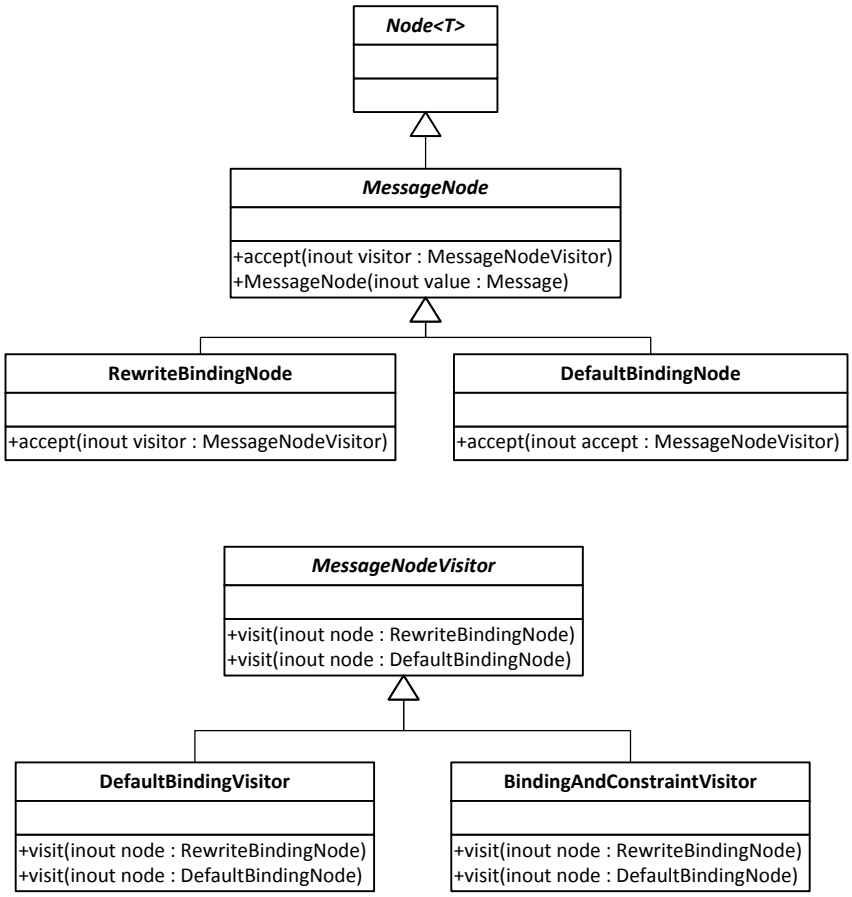

Figure 3.4: treeprototype.tree package class hierarchy

tern [8]. Using this approach, the binding and constraint semantics as described in chapter 2 are implemented. The remainder of this section discusses the implementation details of this protoype, as well as the design decisions that were made.

3.3.1 Overview of message processing in the tree-based prototype

The tree used by this prototype is implemented in the *treeprototype.tree* package. The class hierarchy for the most important classes (with their most important methods) in the tree package is shown in figure 3.4.

Every node in the tree encapsulates one message. *RewriteBindingNodes* represent messages that still have to be rewritten, and DefaultBindingNodes represent messages that can be delivered to a receiver (i.e. a subtype of CoopObject

that has an implementation of the appropriate method). The root node and intermediate nodes of a tree are always *RewriteBindingNode*s, and the leaves of the tree are usually *DefaultBindingNode*s[4].

Initially, a message tree consists of only one *RewriteBindingNode*. This initial node is generated by the *TreeMessageHandler*, which will be discussed shortly. This single node is visited by the *BindingAndConstraintVisitor*. Whenever this visitor encounters a *RewriteBindingNode*, it applies the applicable constraints and bindings, creating new child nodes. It then visits each of these child nodes depth-first, eventually generating a tree with the properties mentioned above (intermediate *RewriteBindingNode*s, and *DefaultBindingNode*s at the leaves). Once this process is completed, an instance of the *DefaultBindingVisitor* visits the tree. When this visitor encounters a *RewriteBindingNode*, it simply visits the children of said node. When it encounters a *DefaultBindingNode*, it delivers the message by reflectively invoking the method that is represented by the message. It does so for all *DefaultBindingNode*s in the tree, in depth-first order. Figure 3.5 illustrates this process.

Details about how the *BindingAndConstraintVisitor* exactly interacts with the classes of the *treeprototype.bindings* and *treeprototype.constraints* packages is discussed in subsections

It is possible to implement other visitors, for example, to generate a graphical representation of the message tree for debugging purposes. Supporting new node types is also possible, by implementing a new subclass of *MessageNode*. Furthermore, a new visitor needs to be defined for the new node type. This requires creating a new subclass of *MessageNodeVisitor*, as well as adding a new *visit(..)* method to *MessageNodeVisitor* and all its subclasses. As can be seen, adding a new node type involves modifying several existing classes, besides adding new subclasses. This is a known downside of the visitor pattern [8]. However, adding new node types is expected to happen only on very rare occasions, and therefore we consider this trade-off acceptable.

[4]The exception being when a message ends up matching no selector expressions of a rewrite binding, and cannot be delivered by the default binding.

Figure 3.5: The tree visiting process.

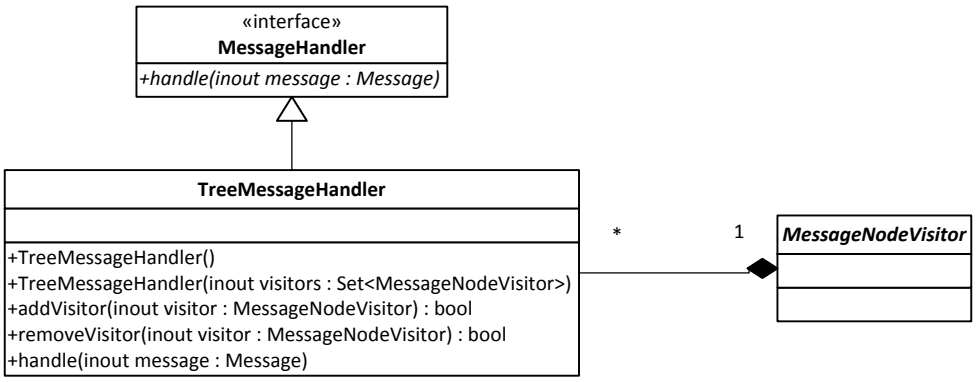

Figure 3.6: treeprototype.messaging package class hierarchy.

3.3.2 Message handler

As discussed in section 3.2, each prototype implementation must implement the *MessageHandler* interface. In this prototype the *TreeMessageHandler* class in the *treeprototype.messaging* package implements this interface. The implementation of this class is very simple, and the relevant methods are shown in figure 3.6. The *TreeMessageHandler* class encapsulates an ordered set of *MessageNodeVisitors*, and implements the *handle(Message message)* method from the *MessageHandler* interface.

Whenever a message is handed off to the *TreeMessageHandler* by the message interception mechanism, the *TreeMessageHandler* creates a new *RewriteBindingNode* representing the message. It then proceeds by having each visitor it encapsulates visit this root node.

By default the *TreeMessageHandler* creates a set containing a *BindingAndConstraintVisitor* and a *DefaultBindingVisitor*, in that order. By doing so, it supports the semantics of bindings and constraints as discussed in chapter 2. However, it is possible to add additional visitors, for example, in order to support new node types, or to support visualisation of the message tree. As there is no reason to have more than one instance of the *TreeMessageHandler*, it is implemented as a singleton.

3.3.3 Bindings

Bindings are implemented in the package *treeprototype.bindings*. This package contains implementations for the binding types that are described in chapter 2. It also provides an implementation for the *BindingFactory* interface that was discussed in section 3.2, and it contains the *BindingManager* class, which is responsible for keeping track of binding classes, binding instances, and whether or not a binding instance is active. Figure 3.7 shows the class hierarchy for this package.

Binding instances are implemented in three classes: *AbstractBinding*, *DefaultBinding* and *RewriteBinding*. The abstract *AbstractBinding* class implements functionality that is common to both binding instance types: activation and deactivation. Furthermore, it provides functionality to determine whether a binding instance is active, to which binding class a particular binding instance belongs, and which CoopObject instance encapsulates the binding. It implements the *Binding* interface of the *common.bindings* package, and provides a dummy implementation for the *addRewriteRule* method. The class defines two abstract methods: *boolean isApplicableTo(Message message)* and *MessageNode applyTo(Message message)*. These methods are necessary to determine if a binding instance is applicable to a given message, and to actually apply a binding instance to a given message. These two methods are invoked by the *BindingAndConstraintVisitor* discussed earlier. Finally, the class has a protected constructor that handles the registration of the binding instance with the *BindingManager*, discussed later.

The *DefaultBinding* class implements the *isApplicableTo* method and the *applyTo* method of the *AbstractBinding* class. The *applyTo* method simply returns a new *DefaultBindingNode* representing the message that is passed to it. The *isApplicableTo* method determines if the default binding is applicable to this message. It does so by inspecting the *target*, *targetType*, *parameters*, and *name* properties of the message, and looking for an object that matches these properties. If it finds such an object, it returns *true*, otherwise it returns *false*.

The *RewriteBinding* class keeps track of the selector expression and rewrite rules of the binding instance it represents. It supports the addition of new rewrite rules through the *addRewriteRule* method. It also implements the *isApplicableTo* and *applyTo* methods of the *AbstractBinding* class. The *isApplicableTo* method passes the message that is provided to it to the selector the *Rewrite-*

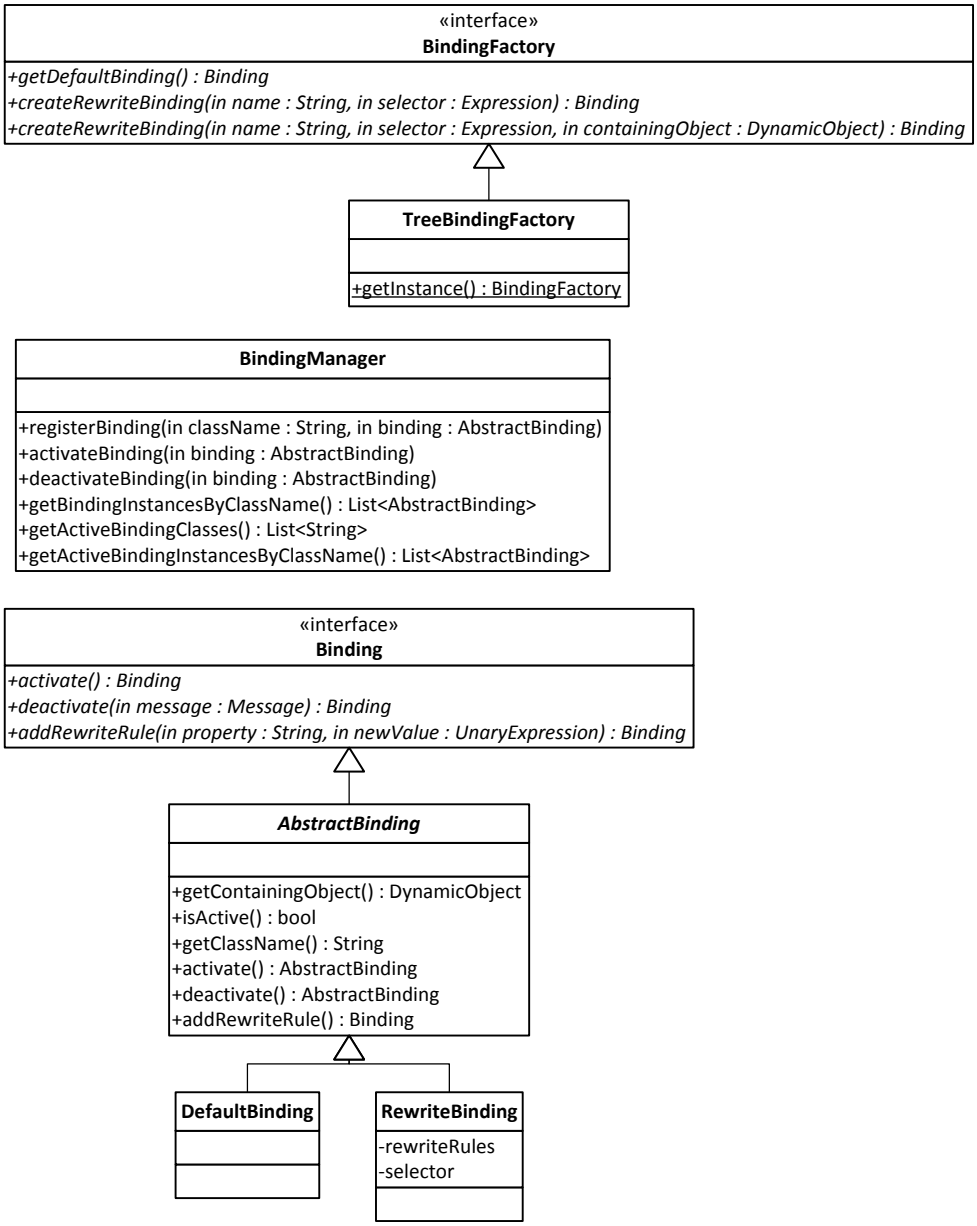

Figure 3.7: treeprototype.bindings package class hierarchy.

Binding instance contains. The result of the selector evaluation determines if the binding instance is applicable to the message. The *applyTo* method takes the message that is passed to it, and applies each rewrite rule that is contained in the *RewriteBinding* instance. After applying all the rewrite rules, it returns a new *RewriteBindingNode* containing the rewritten message.

The *TreeBindingFactory* is a singleton [8], that implements the methods defined by the *BindingFactory* interface. When asked for the default instance, it returns the single instance of the *DefaultBinding* it contains. When asked for a *RewriteBinding*, it creates a new instance of a *RewriteBinding* with the specified binding class, selector, and containing object.

The *BindingManager* singleton class keeps track of all binding classes and their binding instances. On construction, each instance of an *AbstractBinding* subclass registers itself with the *BindingManager* instance. The *BindingManager* is also aware of which bindings are active. Finally it provides methods to retrieve (active) binding instances by binding class name.

3.3.4 Constraints

Constraints are implemented in the package *treeprototype.constraints*. The package contains implementations for the constraints as described in chapter 2. It also provides an implementation for the *ConstraintFactory* interface that was discussed in section 3.2, and it contains the *ConstraintManager* class, which is responsible for keeping track of constraints, and whether or not a constraint is active. The *ConstraintManager* is also responsible for the actual evaluation of constraints. This is supported by using one or more instances of one of the *ConstraintProcessor* subclasses. Figure 3.8 shows the class hierarchy for this package.

Similarly to the binding instances discussed in the previous section, the abstract *AbstractConstraint* class implements some common functionality for constraints. It encapsulates information about the name of the constraint, and the names of the binding classes that are constrained by a particular instance of an *AbstractConstraint* subclass. It also implements methods to activate and deactivate a constraint instance. This activity state is registered with the *ConstraintManager*.

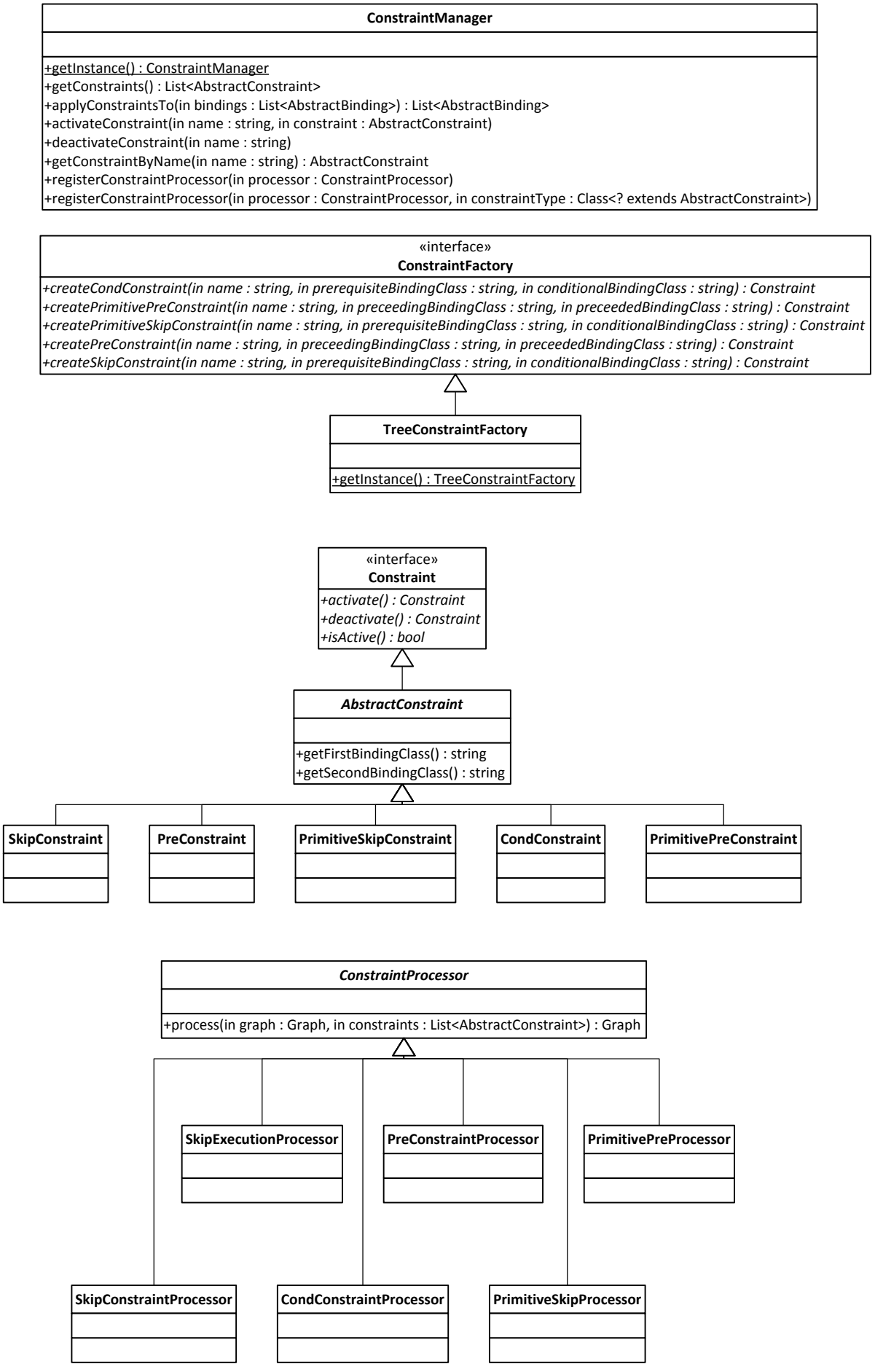

Figure 3.8: treeprototype.constraints package class hierarchy.

The subclasses of *AbstractConstraint*, *PreConstraint*, *PrimitivePreConstraint*, *SkipConstraint*, *PrimitiveSkipConstraint*, and *CondConstraint* merely serve as 'markers'. They are used to determine what kind of constraint the system is dealing with. Alternatively this could have been expressed by not making *AbstractConstraint* abstract, and having it keep track of a constraint type identifier, for example an enumeration representing the constraint types. Both approaches are modular enough (i.e. they allow for definition of new constraint types without modification of existing classes), and there is no clear advantage of one over the other.

The *TreeConstraintFactory* is a singleton that implements the methods defined by the *ConstraintFactory* interface. When asked for a subtype of *AbstractConstraint*, it creates a new instance of a subtype of *AbstractConstraint*, with the provided name, constraining the provided binding classes.

The *ConstraintManager* singleton performs administrative tasks similar to the *BindingManager*. The *ConstraintManager* is not aware of what types of constraints exist, it maintains a mapping of 'subtype of *AbstractConstraint*' to a list of actual instances of that subtype. In addition, it is responsible for applying the constraints that it manages. It does so through instances of *ConstraintProcessor* subclasses. As discussed in 2, every constraint expresses a binary relation between two binding classes. Graphs are a natural way to express a collection of such relationships. Therefore, each subclass of *ConstraintProcessor* transforms a graph[5] of applicable binding instances that is provided to it by the *ConstraintManager*. The *ConstraintManager* creates this graph based on information about active binding classes passed to it by the *BindingAndConstraintVisitor*.

Each *ConstraintProcessor* is executed in turn, being passed the graph that was transformed by the previous *ConstraintProcessor*. This is an implementation of the 'Pipes and Filters' architectural pattern [3]. Figure 3.9 illustrates this process.

The *ConstraintProcessor* instances each perform (part of) the evaluation of a certain constraint type. The *ConstraintManager* is aware of which types of constraints are active, and *ConstraintProcessor* instances register with the *ConstraintManager* which subtype of *AbstractConstraint* (if any) they can process. The *ConstraintProcessor* instances are executed in the order in which they register with the *ConstraintManager*. This system is easily extensible, as adding new types

[5]Using the graph implementation in *common.graph*

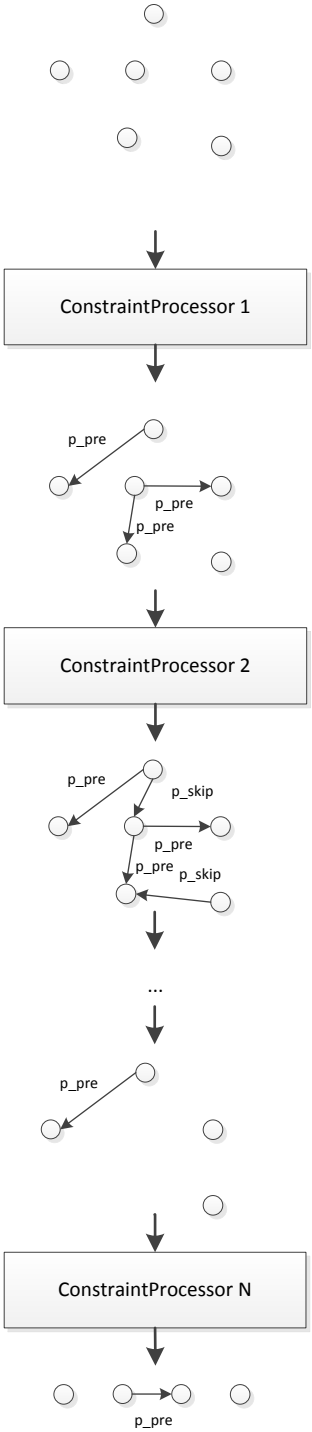

Figure 3.9: Processing constraints.

of constraints is a matter of implementing a new subclass of *AbstractConstraint*, implementing one or more[6] new subclasses of *ConstraintProcessor* for the new constraint type, and registering these *ConstraintProcessors* with the *ConstraintManager*. No modification of the *ConstraintManager* itself is necessary, or any other part of the system.

The current prototype implements seven *ConstraintProcessor*s. They are registered in the following order, and perform the following transformations:

PreConstraintProcessor This processor adds an edge labeled 'p_pre' to the (initially edgeless) graph for each *PreConstraint* that is active and applicable. It then calculates the transitive closure for the graph with these edges and labels any added edges with 'p_pre' as well.

PrimitivePreConstraintProcessor This processor takes the graph created by the previous processor and adds an edge labeled 'p_pre' for each *PrimitivePreConstraint*[7].

SkipConstraintProcessor This processor adds an edge labeled 'p_skip' to the graph, created by the previous processor, for each *SkipConstraint* that is active and applicable. It then calculates the transitive closure of the graph, considering only the edges labeled 'p_skip', and labels any added edges with 'p_skip' as well.

PrimitiveSkipConstraintProcessor This processor takes the graph created by the previous processor and adds an edge labeled 'p_skip' for each *PrimitiveSkipConstraint*. Note that both the *SkipConstraintProcessor* and the *PrimitiveSkipConstraintProcessor* do not remove any vertices yet. This is done later in the process by the *SkipExecutionConstraintProcerssor*.

CondConstraintProcessor This processor adds an edge labeled 'cond' to the graph, created by the previous processor, for each *CondConstraint* that is active and applicable. Note that this does not add edges from inapplicable binding instances to applicable binding instances (as inapplicable binding instances are not in the initial graph). It is possible for a *CondConstraint* to have a relation between an inapplicable binding class and an applicable

[6]Depending on the complexity of the constraint.

[7]The graph implementation in *common.graph* does not support multiple edges between the same vertices with the same label. Adding such duplicate edges to a graph is conveniently ignored.

binding class however. We call this a *dangling cond-constraint*. They will be processed later. Note that, again, this constraint processor does not actually remove any vertices. The cond-constraints are evaluated later on in the process by the *CondExecutionConstraintProcessor*.

SkipExecutionConstraintProcessor This processor 'evaluates' each edge labeled 'p_skip' in topological order. For each 'p_skip' edge it encounters, it removes the vertex at the head of the edge from the graph.

CondExecutionConstraintProcessor This processor 'evaluates' each edge labeled 'cond' in topological order. However, it first checks if any vertices remaining should be eliminated due to a *dangling cond-constraint*. Each vertex that should be removed by this process is added to the *dangling victims* list. The *CondExecutionProcessor* then proceeds by topologically evaluating each vertex and constraint. If the *condition* of a constraint is contained in either the *dangling victims* list, or it does not exist in the graph, the *constrained binding* is removed from the graph.

After this processing is done, the graph contains only vertices for binding instances that should be executed, and edges labeled 'p_pre'. The binding instances are returned to the *BindingAndConstraintVisitor* in topologically sorted order[8]. Figure 3.9 also illustrates this transformation process.

3.4 ALIA4J-based prototype

The ALIA4J-based prototype is built upon the ALIA4J-framework, of which an overview was given in section 3.1. While both prototypes use ALIA4J for message reification through the common framework, with this prototype we attempt to use ALIA4J to ease the implementation of bindings and constraints as well. In chapter 4 we discuss how well this succeeded, and we compare the implementations of the prototypes. In this section we discuss the implementation details and design decisions of the ALIA4J-based prototype.

With the ALIA4J-prototype we to use ALIA4J primitives to implement bindings and constraints. We chose to implement bindings using ALIA4J attachments. This choice was made because these attachments support the use of

[8] if a cycle exists in the graph, this will fail, and the running program will terminate.

predicates to test the current context (including the current message) to determine their applicability. When putting this in Co-op terms, these predicates can be used as selector expressions. Furthermore, ALIA defines constructs (PrecedenceRule and CompositionRule) to provide ordering between attachments, as well as control over their execution. This is similar to constraints in Co-op.

3.4.1 Overview of message processing in the ALIA4J prototype

The processing of messages in the ALIA4J prototype is more complex than in the tree-based prototype. To improve understanding of this process we first give a short overview before we proceed with the detailed discussion of the system.

Figure 3.10 shows the flow of a message through the *MessageHandler* and constraint and binding system. Initially, a message is generated by the *MessageGenerator* in the *common.messaging* package. The message is handed off to the *MessageHandler.handle(...)* method. This method simply passes the message on to the *prepareConstraints(Message message)* method.

The *prepareConstraints* method is responsible for evaluating control constraints. The reason for this will be discussed in subsection 3.4.4. After processing the control constraints, the static *dispatch(Message message, DisabledBindingInstances instances)* method is called. However, this method will never be actually executed, as the ALIA4J-attachment contained by subtypes of the *AbstractBinding* class will intercept the call. The reason for this 'dummy' invocation of the *dispatch* method is that ALIA4J, in Co-op terms, cannot generate new messages. While ALIA4J does support the invocation of a target method using the *proceed*-mechanism, these invocations cannot be intercepted again by ALIA4J. Therefore, we do not use this mechanism, but instead generate a dummy method call. As ALIA4J is not aware of such calls, it is not able to optimize for this situation.

The subtypes of *AbstractBinding* use a *predicate* to determine if they should actually do so. The message is handed off to the appropriate implementation of *AbstractBinding*, and if the binding was not made inapplicable by a control constraint, the binding is executed. In the case of a *RewriteBinding*, the message is rewritten, and sent to the *prepareConstraints* method of the *MessageHandler*

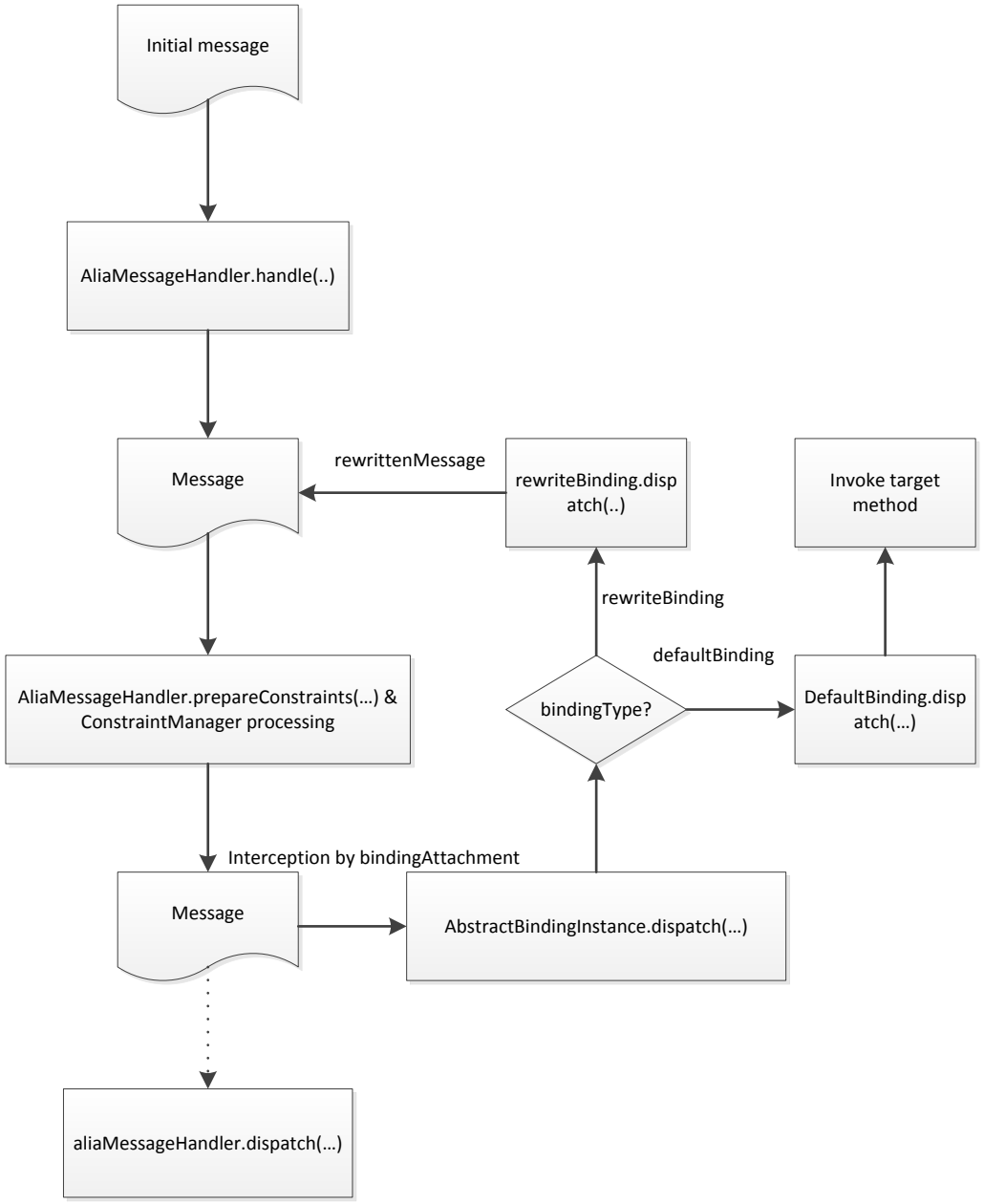

Figure 3.10: Message processing in the ALIA4J-based prototype.

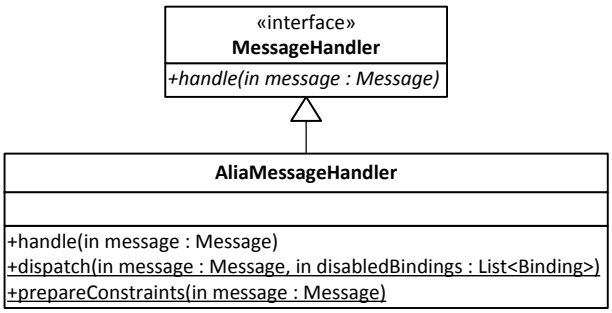

Figure 3.11: The *aliaprototype.messaging* package.

again, restarting the process. The *DefaultBinding* performs an actual method call, using the Java reflection mechanism.

3.4.2 Message handler

This prototype, as is necessary for the proper handling of messages, also has an implementation of the *common.messaging.MessageHandler* interface. The interface is implementated in the *aliaprototype.messaging* package, by the *AliaMessageHandler* class. Figure 3.11 shows the class hierarchy of the *aliaprototype.messaging* package.

As discussed above, the *AliaMessageHandler* implements three methods: *handle(...)*, *dispatch(...)*, and *processConstraints(...)*. The simplest of these methods is *dispatch*. This *static* method merely functions as a dummy method for bindings to intercept.

The *handle* method is necessary for the implementation of the *MessageHandler* interface. In the case of *AliaMessageHandler*, it simply hands off the message it is passed to the *prepareConstraints* method. This method initiates the evaluation of control constraints for each message. Constraint processing is discussed in detail in subsection 3.4.4. The implementation keeps track of which bindings should not be executed due to the evaluation of control constraints.

3.4.3 Bindings

Bindings are implemented in the package *aliaprototype.bindings*. This package contains implementations for the binding types that are described in chapter

2. It also provides an implementation for the *BindingFactory* interface that was discussed in section 3.2, and it contains the *BindingManager* class, which is responsible for keeping track of binding classes, binding instances, and whether or not a binding instance is active. The package also contains the *DisabledBindingInstances* class, which encapsulates a list of bindings, as well as the implementations of certain selector expression operators as ALIA4J predicates, and an implementation of a *common.messaging.ExpressionVisitor*, *SelectorExpressionTransformationVisitor*, to transform selector expressions from *common.expressions* to ALIA4J predicates. Finally, the class *BindingClassAttachmentReference* implements an ALIA4J *AttachmentReference* that refers to the attachments for a given binding class. Figure 3.12 shows the class hierarchy for this package.

Binding instances are again implemented in three classes: *AbstractBinding*, *DefaultBinding* and *RewriteBinding*. The abstract *AbstractBinding* class implements functionality that is common to both binding instance types: activation and deactivation. Furthermore, it provides functionality to determine whether a binding instance is active, to which binding class a particular binding instance belongs, and which CoopObject instance encapsulates the binding. It implements the *Binding* interface of the *common.bindings* package, and provides a dummy implementation for the *addRewriteRule* method. The class defines two abstract methods: *boolean isApplicableTo(Message message)* and *MessageNode applyTo(Message message)*. These methods are necessary to determine if a binding instance is applicable to a given message, and to actually apply a binding instance to a given message. Furthermore, the class implements a method to retrieve the *attachment* that it encapsulates. Finally, the class has a protected constructor that handles the creation of its *attachment*[9], and registration of the binding instance with the *BindingManager*, discussed later.

Setting up the ALIA4J *attachment* is done by transforming the selector to an ALIA4J *predicate* by applying the *SelectorExpressionTransformationVisitor* to the *SelectorExpression* that is passed to the constructor. Theoretically, by doing so, the binding does not have to determine by itself if it is applicable to a message, since the *attachment* will do so. If we would only implement *pre* and *primitive pre* constraints in this prototype, this would be the case. However, as we also implement the control constraints of Co-op/III, this is no longer sufficient.

[9]In ALIA4J, attachments (and constraint rules such as the PrecedenceRule), need to be *deployed* before they function. Deploying the encapsulated *attachment* is done by the activate method, and conversely, undeployment is performed by the deactivate method.

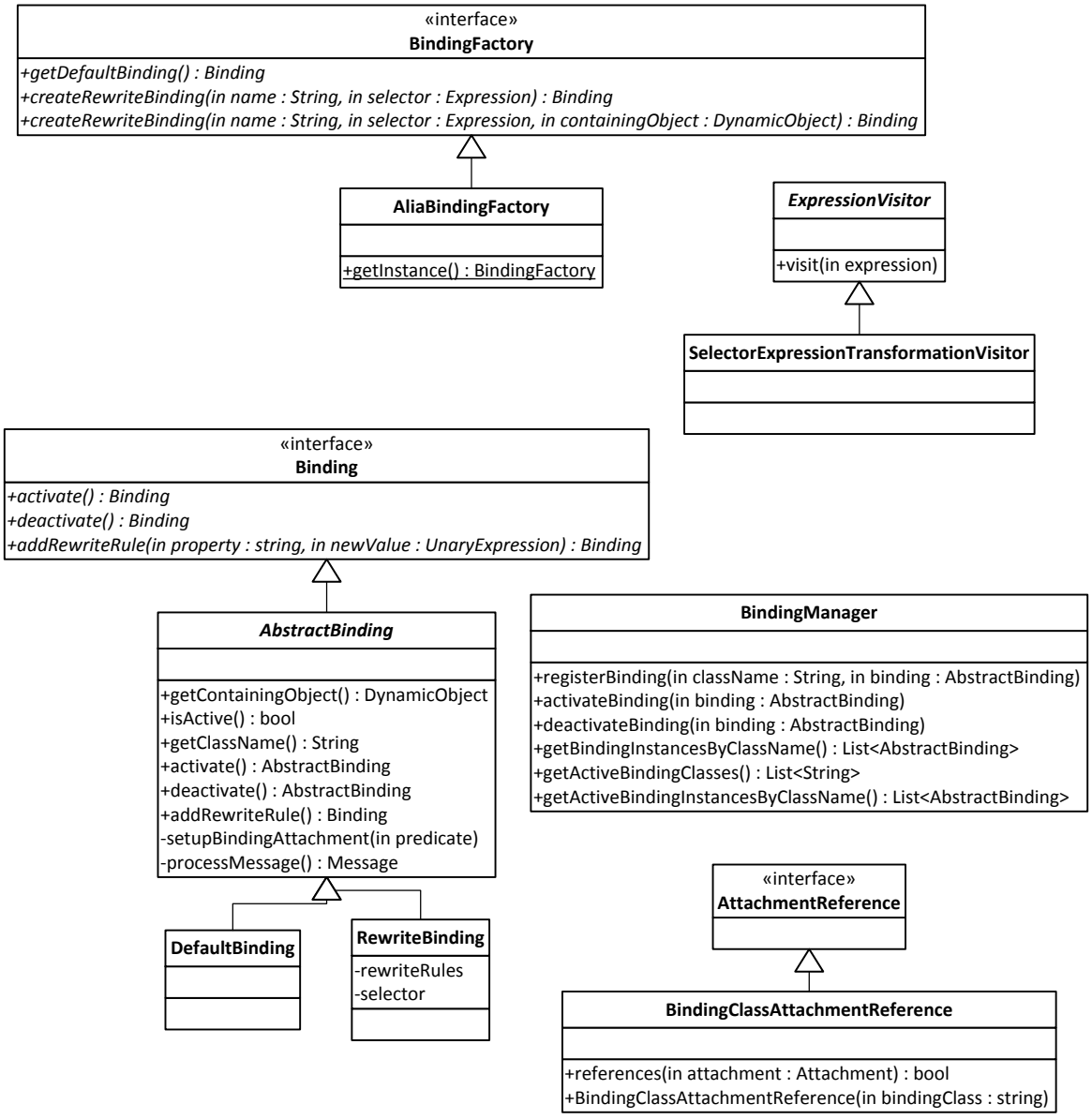

Figure 3.12: aliaprototype.bindings package class hierarchy.

For these control constraints we need to be able to evaluate the selector expressions separately. Subsection 3.4.4 on constraints will elaborate why exactly this is necessary. However, as ALIA4J predicates cannot be evaluated separately, bindings in this prototype still need the *isApplicableTo(...)* method. Furthermore, as a binding needs to know, after evaluation of the control constraints, if it is still applicable, the predicate of the *attachment* contained by the binding also validates if it is not in the list of binding instances contained in the *DisabledBindingInstances* instance that is also passed to the *dispatch* method of the *AliaMessageHandler*. If the binding instance encapsulating the attachment is in the list, the predicate will be false, and the binding will no longer match.

The *DefaultBinding* class implements the *isApplicableTo* method and the *applyTo* method of the *AbstractBinding* class. The *applyTo* method invokes the target of the message. The *isApplicableTo* method determines if the default binding is applicable to this message. It does so by inspecting the *target*, *targetType*, *parameters*, and *name* properties of the message, and looking for an object that matches these properties. If it finds such an object, it returns *true*, otherwise it returns *false*.

The *RewriteBinding* class keeps track of the selector expression and rewrite rules of the binding instance it represents. It supports the addition of new rewrite rules through the *addRewriteRule* method. It also implements the *isApplicableTo* and *applyTo* methods of the *AbstractBinding* class. The *isApplicableTo* method passes the message that is provided to it to the selector the *RewriteBinding* instance contains. The result of the selector evaluation determines if the binding instance is applicable to the message. The *applyTo* method takes the message that is passed to it, and applies each rewrite rule that is contained in the *RewriteBinding* instance. After that, it invokes *prepareConstraints*, as discussed in 3.4.1.

The *AliaBindingFactory* is a singleton [8], that implements the methods defined by the *BindingFactory* interface. When asked for the default instance, it returns the single instance of the *DefaultBinding* it contains. When asked for a *RewriteBinding*, it creates a new instance of a *RewriteBinding* with the specified binding class, selector, and containing object.

The *BindingManager* singleton class keeps track of all binding classes and their binding instances. On construction, each instance of an *AbstractBinding*

subclass registers itself with the *BindingManager* instance. The *BindingManager* is also aware of which bindings are active. Finally it provides methods to retrieve (active) binding instances by binding class name.

The *BindingClassAttachmentReference* implements an ALIA4J *AttachmentReference*, and is used by the ordering constraints discussed in the next section to implement precedence constraints between binding classes. It works by implementing the *boolean references(Attachment attachment)* method of the *AttachmentReference* class. This method returns true if the attachment passed to this method is contained by a binding instance that is part of the binding class the *BindingClassAttachmentReference* instance represents.

3.4.4 Constraints

Constraints are implemented in the package *aliaprototype.constraints*. The package contains implementations for the constraints as described in chapter 2. It also provides an implementation for the *ConstraintFactory* interface that was discussed in section 3.2, and it contains the *ConstraintManager* class, which is responsible for keeping track of constraints, and whether or not a constraint is active. The *ConstraintManager* is also responsible for the actual evaluation of control constraints. Figure 3.13 shows the class hierarchy for this package.

Similarly to the binding instances discussed in the previous section, the abstract *AbstractConstraint* class implements some common functionality for constraints. It encapsulates information about the name of the constraint, and the names of the binding classes that are constrained by a particular instance of an *AbstractConstraint* subclass. It also implements methods to activate and deactivate a constraint instance. This activity state is registered with the *ConstraintManager*.

Similar to the tree-based prototype, subclasses of *AbstractConstraint*, *SkipConstraint*, *PrimitiveSkipConstraint*, and *CondConstraint* merely serve as 'markers'. They are used to determine what kind of constraint the system is dealing with. The *PreConstraint* and *PrimitivePreConstraint*, in addition, encapsulate an ALIA4J *PrecedenceRule*. This precendence rule imposes a partial ordering between two *BindingClassAttachmentReferences*, discussed in the previous subsection. In the case of the (transitive) *PreConstraint*, activation of the constraint

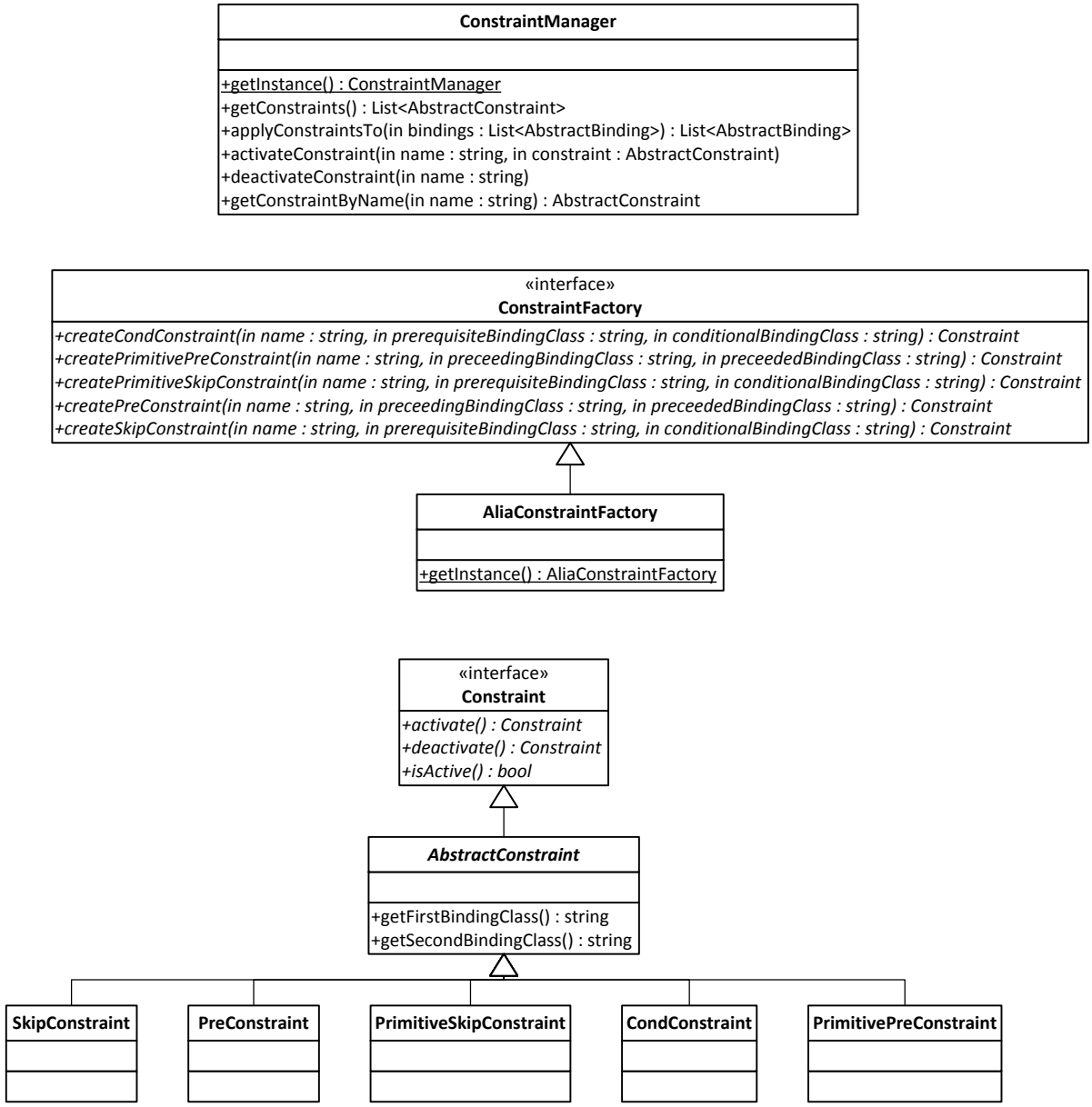

Figure 3.13: aliaprototype.constraints package class hierarchy.

involves calculating the transitive closure over the constrained binding classes as well, and creating a *BindingClassAttachmentReference* for each binding class that is closed over by the constraint. This entire set of *BindingClassAttachmentReference* instances can then be passed to a single *PrecedenceRule*. Using *PrecedenceRule* instances and the *BindingClassAttachmentReference* makes implementing the ordering constraints of Co-op/III very simple.

ALIA also defines a *CompositionRule*, which could potentially be used to implement control constraints. However, due to a problem with the semantics of this *CompositionRule*, at least in the release of ALIA4J used for the development of this prototype[10], the *CompositionRule* is disabled. The consequences of this are that (1) we are unable to determine if control constraints can be implemented at all using *CompositionRule*s, and (2) implementing control constraints is difficult using the chosen approach.

On a theoretical level, the former should be, at least partially, possible. *CompositionRule*s can be used to define overriding relationsships and constraints for jointly executed Actions according to the documentation [2]. However, as in practice the *CompositionRule* cannot be used, this cannot be validated. Furthermore, we are unsure if we can express the precedence rule for control constraints[11] that was discussed in 2.3.2. Finally, it appears that the evaluation of *CompositionRule*s depends on static analysis. As the semantics of Co-op/III constraints are mostly dependant on dynamic (runtime) information, it might not be possible at all. However, as it is not possible to actually test or investigate this, it is unclear what the precise impact is.

The latter can be solved by, before the bindings actually get to intercept the message (through the dispatch method), evaluating the control constraints. However, this solution has some negative consequences:

- We need to know, before the attachment of a binding actually matches, if a binding should be allowed to apply or not. This requires us to separately evaluate the predicate of an attachment. As this is not possible in ALIA4J, we need to use the original selector implementation, as is discussed in subsection 3.4.3. Therefore, it is no longer truly useful to use predicates to determine which bindings to apply.

[10] Version ALIA4J-0.1.0.

[11] *Skip* and *primitive skip* should be evaluated before *cond*.

- Using the *CompositionRule* of ALIA would nicely modularize the behavior of control constraints. This is no longer possible, as now the prototype itself needs to keep track of which bindings are enabled and disabled due to the application of constraints.
- Being able to use *AttachmentReference*s would simplify the implementation of the transitive *skip* constraint. However, this is no longer possible.
- We can no longer take advantage of improvements to the ALIA4J framework, at least when it comes to control constraint implementation.

We did however still choose to implement control constraints in the manner mentionend. If/when the *CompositionRule* (with the proper semantics) becomes available again, it should be relatively easy to implement the control constraints using the *CompositionRule*. Furthermore, we still benefit from the use of the *PrecedenceRule* and optimizations to other parts of ALIA, so the prototype is useful.

The implementation of control constraints is now done by having each *RewriteBinding* send their rewritten message to the *prepareConstraints* method of the *AliaMessageHandler*. The *prepareConstraints* method then delegates the processing of the control constraints to the *ConstraintManager* class. This class evaluates the constraints in a manner similar to the tree-based prototype, using a graph, albeit in a less flexible manner[12]. The result of this is a list of bindings that should no longer be applicable due to them being constrained by the control constraints. This list of bindings is passed on to the *dispatch* method of the *AliaMessageHandler*, and available as context to the binding instances. The predicate of the attachment encapsulated by the binding instance can now use this list to determine if it is actually applicable. If it is not, the predicate will be false, and the binding will not be executed.

[12]Note, however, that it is possible to reuse the entire constraint handling mechanism of the tree-based prototype for this.

4. Evaluation

In this chapter we evaluate both prototype implementations. We compare them based on traditional software quality metrics, as well as on modularity and extensibily.

However, first we determine if, and to what extent, the prototypes support the requirements presented in section 1.2. For convenience, we repeat those requirements here:

R1: easy integration with the existing Co-op/III code base.

R2: binding and constraint semantics compatible to those found in Co-op/II.

R3: a reasonably performing implementation of bindings and constraints.

R4: the possibility to optimize the implementation of the bindings and constraints further, without influencing code generation.

R5: the possibility to implement new types of bindings and constraints.

Requirement **R1** is supported by both prototypes, as they both implement the interfaces described in subsection 3.2.3. These interfaces were designed to support easy integration with Co-op/III.

Requirement **R2** is also supported by both prototypes. Test cases were derived from the definition of binding and constraint semantics as defined in chapter 2. As both prototypes succeed in executing all the tests, they both conform to those semantics.

Requirement **R3** is difficult to quantify, as the prototypes both execute on the NOIRIn interpreting execution environment of ALIA4J. This has a significant negative performance impact. As such, it is difficult to say if the prototype implementations are viable for 'real-world' usage. We can conclude that, in relation to eachother, the prototypes have similar performance characteristics (they run their tests withing about 30ms of eachother). However, the prototypes

do both improve on the binding and constraint handling process of Co-op/II[1], as both prototypes avoid the generation of unnecessary messages.

Requirement **R4** is supported by both prototypes, as their implementations are hidden behind the interfaces described in subsection 3.2.3. As the prototypes *must* conform to those interfaces, and the code generation process *must* only use those interfaces, the implementations can change without affecting code generation.

Requirement **R5** is the most difficult requirement to evaluate. As both prototypes are fundamentally based on Java, if it is possible to implement extra bindings and constraints at all, it can be done in both prototypes. Therefore, we can state that both prototypes support this requirement. However, when considering how easy it is to implement such extensions, the discussion becomes more complex.

The tree-based prototype always requires the implementation of the entire binding or constraint without support of the underlying framework. That makes it initially more complex to develop new constraints. However, as the prototype is designed to support extensibility when it comes to handling constraints, as shown in section 3.3, constraints can be implemented modularly.

As shown in section 3.4, when ALIA4J primitives can be used to support the development of a binding or constraint, it can be very easy to do so. See for example the *pre* constraint. However, when ALIA4J primitives cannot be used, it can become quite complex to implement additional constraints. As is shown by the implementations of the *skip* and *cond* constraints, it can require some workarounds and might make the use of some ALIA4J features inconvenient or even impossible.

The mechanism for controlling action execution in ALIA4J (i.e. the Precedence and Composition rules) are not extensible. The consequence is that it is not possible to define new rules. Thus, defining Co-op constraints that require primitives other than the Precedence rule or Composition Rule will likely require a lot of effort.

Concerning debuggability, the tree-based prototype is currently easier to de-

[1]Co-op/II treated bindings and constraints as entirely separate concerns, as discussed in the introduction.

bug. As this prototype only uses conventional Java constructs[2], conventional Java debuggers such as the one in the Eclipse IDE can be used. Debugging the ALIA-based prototype is more difficult. Applications using ALIA4J in general are difficult to debug, as it is often difficult to see what parts of a program are affected by an attachment. There are currently no tools to support debugging such applications. However, a visual debugger for ALIA4J is currently in development [22], but is not yet available.

The tree-based prototype is easier to understand by simple inspection of the code. The ALIA4J-based prototype is difficult to understand as it may be unclear how attachments are affecting the messages.

Software metrics were measured[3] and analysed as well. We are in particular interested in modularity, and therefore in the metrics for *coupling* and *cohesion*. We observed the following:

- Only the ConstraintManager in the tree-based prototype shows a high fan out (i.e. it 'knows' about more than 25 other types). For most classes the fan out is well below 15.
- None of the prototypes exhibit significant fan in, for most classes the afferent coupling is 10 or lower.
- When considering the Chidamber and Kemerer definition for Lack of Cohesion, none of the classes in both prototypes exhibited significant lack of cohesion.
- When considering the Henderson-Sellers definition for Lack of Cohesion, none of the classes in both prototypes exhibit significant lack of cohesion.
- None of the prototypes use very deep inheritance, the deepest inheritance hierarchy is 4 levels. This depth of inheritance is exhibited by two classes that extend ALIA4J predicates, and part of this inheritance occurs inside the ALIA4J framework.

The above shows that both prototypes are of good quality, when considering traditional software metrics. Note, however, that ALIA4J 'hides' some depen-

[2]Except for the interception of messages. However, this is a very modularized and small part of the prototype.

[3]The Eclipse Metrics [15] and Eclipse Metrics (continued) [9] Eclipse plugins were used for this purpose.

dencies, as it does not always use traditional mechanisms to invoke methods. Software metric tools are not able to see those dependencies. Similarly, as both prototypes depend on reflection to implement the 'default binding', and reflective method calls are also invisible to metric tools, dependencies created there are also not considered.

5. Conclusion

In this chapter we discuss the conclusion of this thesis, as well as future work.

5.1 Conclusion

The main purpose of this thesis was to investigate how to flexibly and extensibly implement composition operators. The idea of having flexible, developer-defined composition operators is already embraced by the Co-op language. Therefore, the latest iteration of the Co-op language, Co-op/III, was used as a foundation for this work. In this thesis we developed two implementations for bindings and constraints in Co-op/III.

Co-op/II, the predecessor of Co-op/III, already supports the definition of composition operators by using *bindings* and *constraints*. However, for Co-op/III there existed no specification for the semantics of bindings and constraints. As we wanted to ensure that our implementations provided the same semantics, first we developed a precise specification for the semantics of bindings, and in particular constraints, in Co-op/III. This specification was used to develop test cases for our implementations, to show that they were semantically equivalent. Furthermore, we also show how the semantics of Co-op/III constraints compare to the constraints described by Nagy [14]. While the constraints of Co-op are inspired by Nagy's constraints, there are some significant differences. The most important of these differences is that certain types of constraints (the control constraints) also necessitate a partial order of the constrained actions. Furthermore, the constraints of Nagy support complex condition expresions, when compared to Co-op. However, as Te Brinke has shown [18], it is still possible to implement a large class of composition operators using only the relatively simple conditions that are available in Co-op. Finally, Nagy's constraints need to be evaluated in a particular order. We have partially shown that this is the result of the semantics of the condition expressions used by those constraints, and we also show that such an evaluation order is not necessary for Co-op constraints.

Based on the specification for bindings and constraints, we developed two implementations. These implementations implement a common interface. This supports the development of test cases and provides a convenient target for code generation. Furthermore, the implementations are hidden from the code generator. We have shown that both a plain Java implementation[1] and an implementation based on ALIA4J [2] can be used. We evaluated both approaches and can summarize the results as follows. Both approaches improve on handling of bindings and constraints when compared to Co-op/II, as the generation of unnecessary messages is eliminated.

The Java-based approach, which works by generating a tree of applicable bindings, has the following advantages:

- The implementation is not limited by any underlying frameworks. The entire implementation is designed with the goal of implementing bindings and constraints in Co-op/III. An important consequence is that it is not necessary to implement (sometimes complex) workarounds to overcome limitations of such frameworks.
- Due to being independent of an underlying framework, the implementation is very flexible. The pipe-and-filter approach used makes it easy to implement additional semantics, as long as these semantics depend on the manipulation of a directed graph.
- The implementation is relatively easy to understand and debug. As the implementation does not depend on any specific framework, anyone with reasonable Java experience can understand the implementation. Furthermore, the implementation can be debugged using common Java debugging facilities, such as those found in the Eclipse IDE.

The Java-based approach also has some disadvantages:

- As Java and the Java Class Library do not provide any support for developer-defined composition operators or advanced programming language constructs, such as aspects or advanced dispatching, all types of bindings and constraints had to be implemented manually.

[1] With the exception of message reification.

- Consequently, the implementation of bindings and constraints has to be optimized manually, there are no optimizations in the underlying framework that can be taken advantage of.
- The implementation cannot use advanced modularization mechanisms such as aspect-orientation, impairing its modularity.

The ALIA4J-based approach uses the primitives provided by ALIA4J to implement bindings and constraints. This approach has the following advantages:

- When an ALIA4J primitive closely matches a concept from Co-op/III (i.e. a binding or constraint), it is easy to implement such a concept. A good example of this are bindings, which can easily be mapped to ALIA4J Attachments, and selector expressions, which can be implemented using predicates. Pre-constraints can also easily be implemented using the Precedence Rule in ALIA4J.
- As ALIA4J is developed separately from the implementation of bindings and constraints, but provides a relatively stable interface, optimization of ALIA4J results in better performance of the binding and constraint implementation. These performance improvements come 'for free', i.e. without having to modify the implementation of the bindings and constraints.
- Due to its advanced-dispatching nature, when depending only on ALIA4J primitives the ALIA4J-based implementation is more modular than the Java-based implementation.

However, the ALIA4J-based implementation has some disadvantages as well:

- ALIA4J does not natively support intercepting a method call that was generated by ALIA4J. As we model bindings as attachments, and bindings can apply to messages generated by other bindings, this is a significant shortcoming. We developed a workaround for this, but this may have an impact on the ability of ALIA4J to optimize.
- While the work on ALIA4J describes a Composition Rule that might be used to implement skip and cond constraints, in the current release of ALIA4J this Composition Rule is not implemented. The most important

consequence of this omission is that complex workarounds were necessary to make these skip and cond constraints work.

- The mechanism for controlling the execution order actions cannot be extended. In other words, it is not possible to add constructs such as the Precedence Rule or Composition Rule without modifying the source of ALIA4J itself. This might be an issue when trying to implement new types of constraints.
- Debuggability is currently an issue, as there are no debugging tools specifically for ALIA4J, and the common debugging tools are not always usable when dealing with ALIA4J-specific issues. There is ongoing work to develop a debugger for ALIA4J, but it was not available at the time of this work.
- The execution flow of programs using advanced-dispatching mechanisms and aspect-orientation is generally harder to understand, as it is often unclear where specific behavior applies. This is also the case when using ALIA4J. This makes extending the implementation of this framework more complex.

The two most important disadvantages of the ALIA4J approach are the lack of debugger support, which makes resolving some bugs very difficult, and the lack of primitives to support the implementation of control constraints. The latter of these disadvantages is a clear indication that ALIA4J is not (yet) expressive enough. Currently we feel that the potential advantages of using ALIA4J do not outweight the disadvantages.

5.2 Future work

For future work based on this thesis we can consider two directions. The created prototypes can be improved on and one of them can be added to Co-op/III, or other implementation strategies can be explored.

Some improvements have to be made to both prototypes before they can be used in Co-op/III. The ability for selectors and bindings to generate new messages can cause (potentially infinite) recursion, as discussed by Te Brinke

[18]. The common infrastructure used by both implementations does support a rudimentary form of recursion detection that can possibly also be used to avoid (some forms of) recursion, but this has not been extensively tested. Furthermore, currently there is no way to bind message properties to implicit parameters in Co-op/III. However, implementing this is likely easy, as implicit parameters can trivially be implemented using a stack on which the message properties can be placed.

If the ALIA4J-based prototype is to be used, it might be necessary to investigate if that framework can be extended. While it is promising, as discussed in the previous section it currently lacks some features to make it useful for implementing Co-op/III bindings and constraints. In particular, a mechanism to support the execution of actions based on the presence of other actions (i.e. the ALIA4J-equivalent of control constraints) should be (re)introduced.

A final option is not using ALIA4J alltogether, and explore other frameworks, such as Reflex [17]. However, there does not seem to be a lot of active development on these frameworks. For example, development on Reflex has been stagnant for several years now. Still, it might be worth investigating if there are other options available.

BIBLIOGRAPHY

[1] BÉZIVIN, J. On the unification power of models. *Software and Systems Modeling 4*, 2 (2005), 171–188.

[2] BOCKISCH, C., SEWE, A., YIN, H., MEZINI, M., AND AKŞIT, M. An in-depth look at alia4j. *Journal of Object Technology 11*, 1 (Apr. 2012), 1–28.

[3] BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. *Pattern-oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.

[4] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. *Introduction to Algorithms*, 3 ed. The MIT Press, 2009.

[5] FILMAN, R. E., ELRAD, T., CLARKE, S., AND AKŞIT, M., Eds. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[6] FOWLER, M. Fluent interface. `http://www.martinfowler.com/bliki/FluentInterface.html`, 2005. [Online, accessed 24th of February 2013].

[7] FRIEDMAN, D., AND WISE, D. Aspects of applicative programming for parallel processing. *Computers, IEEE Transactions on C-27*, 4 (april 1978), 289 –296.

[8] GAMMA, E., HELM, R., JOHNSON, R., AND VLISIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] GBOISSIER. Eclipse metrics plugin (continued). Eclipse Marketplace, 2009. [Online, Eclipse IDE plugin, version 1.3.8.

[10] GOSLING, J., JOY, B., STEELE, G., BRACHA, G., AND BUCKLEY, A. The java language specification - java se 7 edition. `http://docs.oracle.com/javase/specs/jls/se7/html/index.html`, 2012. [Online, accessed 24th of February 2013].

[11] HAVINGA, W. K., BOCKISCH, C. M., AND BERGMANS, L. M. J. A case for custom, composable composition operators. In *Proceedings of the 1st International Workshop on Composition: Objects, Aspects, Components, Services and Product Lines, Rennes, France* (March 2010), vol. 564 of *Workshop Proceedings*, CEUR-WS, pp. 45–50.

[12] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., MARC LOINGTIER, J., AND IRWIN, J. Aspect-oriented programming. In *ECOOP* (1997), SpringerVerlag.

[13] MUSCHEVICI, R., POTANIN, A., TEMPERO, E., AND NOBLE, J. Multiple dispatch in practice. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications* (New York, NY, USA, 2008), OOPSLA '08, ACM, pp. 563–582.

[14] NAGY, I. *On the Design of Aspect-Oriented Composition Models for Software Evolution*. PhD thesis, University of Twente, Enschede, June 2006.

[15] OF FLOW, S. Eclipse metrics. Eclipse Marketplace, 2010. [Online, Eclipse IDE plugin, version 3.12.0].

[16] PAPAZOGLOU, M. P. *Web Services: Principles and Practice*. Pearson Education Limited, 2008.

[17] TANTER, E., AND NOYÉ, J. A versatile kernel for multi-language aop. In *Generative Programming and Component Engineering*, R. Glück and M. Lowry, Eds., vol. 3676 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, pp. 173–188.

[18] TE BRINKE, S. First-order function dispatch in a java-like programming language. Master's thesis, University of Twente, January 2011.

[19] THE ALIA4J PROJECT. Alia4j overview. `http://www.alia4j.org/alia4j/`, 2008-2012. [Online; accessed 1st of February 2013].

[20] THE ECLIPSE FOUNDATION. The aspectj project. `http://www.eclipse.org/aspectj/`, 2013. [Online; accessed 24th of February 2013].

[21] WEGNER, P. Concepts and paradigms of object-oriented programming. *SIGPLAN OOPS Mess. 1*, 1 (Aug. 1990), 7–87.

[22] YIN, H., BOCKISCH, C. M., AND AKŞIT, M. A fine-grained debugger for aspect-oriented programming. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development, AOSD 2012, Potsdam, Germany* (New York, March 2012), ACM, pp. 59–70.

A. Software Metrics

This appendix contains the results of the software metrics tools, split by prototype. Legend:

Abbreviation	Meaning
LINE	Lines of code
WMC	Weighted Methods per Class
Ce	Efferent Coupling (fan out)
NOF	Number of Fields
LCOM	Lack of Cohesion of Methods
LCOM-PFI	LCOM - Pairwise Field Irrelation
LCOM-TC	LCOM - Total Correlation
LCOM-HS	LCOM - Henderson-Sellers
LCOM-CK	LCOM - Chidader & Kemerer

Common prototype features

TYPE	LINE	WMC	Ce	NOF	LCOM-PFI	LCOM-TC	LCOM-HS	LCOM-CK
DynamicClass	3	21	3	0				
RecursionAvoider	7	3	4	0				
AndExpression	5	3	5	0				
ComparisonExpression	3	3	4	2	100	100	100	1
EmptyArrayExpression	3	4	3	0				
EqualsExpression	5	4	7	0				
ExpressionVisitor	3	15	16	0				
GreaterOrEqualExpression	5	3	6	0				
GreaterThanExpression	5	3	6	0				
LessOrEqualExpression	5	3	6	0				
LessThanExpression	5	3	6	0				
LogicExpression	3	3	2	2	100	100	100	1
MessagePropertyLookupExpression	5	8	7	1	0	0	0	0
MethodInvocationExpression	5	3	3	1	0	0	0	0
NotEqualsExpression	5	4	7	0				
NullExpression	3	4	3	0				
ObjectArrayExpression	3	4	3	1	0	0	0	0
ObjectConstantExpression	3	4	3	1	0	0	0	0
OrExpression	5	3	5	0				
ReflectiveMethodInvocationExpression	6	8	9	1	0	0	0	0
TernaryValue	3	0	1	0				
TernaryValue.FALSE	5	0	1	0				
TernaryValue.TRUE	4	0	1	0				
TernaryValue.UNKNOWN	6	0	1	0				
UnaryExpression	3	3	2	1	0	0	0	0
Edge	4	8	5	2	50	12	33	0
Graph	9	80	25	2	74	33	39	0
LabeledEdge	3	6	8	1	0	0	0	0
Vertex	3	7	7	1	0	0	0	0
Message	7	15	7	1	0	0	0	0
MessageGenerator	34	11	20	1	0	0	0	0
MessageGenerator$(anonymous)$	36	1	4	0				
ResultWrapper	3	4	2	1	0	0	0	0

Tree-based prototype

TYPE	LINE	NOF	WMC	Ce	LCOM-PFI	LCOM-TC	LCOM-HS	LCOM-CK
AbstractBinding	10	3	10	9	100	112	89	4
BindingManager	9	2	12	9	100	92	60	1
DefaultBinding	10	0	10	10				
RewriteBinding	16	2	13	15	71	29	42	0
TreeBindingFactory	8	1	5	8	0	0	0	0
AbstractConstraint	9	4	7	4	100	116	90	9
CondConstraint	10	0	1	3				
CondConstraintProcessor	13	0	11	13				
ConstraintManager	15	3	19	26	79	99	67	1
ConstraintProcessor	11	0	7	8				
PreConstraint	9	0	1	3				
PreConstraintProcessor	13	0	12	17				
PrimitivePreConstraint	9	0	1	3				
PrimitivePreConstraintProcessor	12	0	5	12				
PrimitiveSkipConstraint	9	0	1	3				
PrimitiveSkipConstraintProcessor	12	0	6	12				
SkipConstraint	9	0	1	3				
SkipExecutionConstraintProcessor	8	0	4	7				
TransitiveSkipConstraintProcessor	13	0	12	17				
TreeConstraintFactory	6	0	8	9				
TreeMessageHandler	15	1	6	11	0	0	0	0
BindingAndConstraintVisitor	15	0	9	12				
DefaultBindingNode	5	0	4	4				
DefaultBindingVisitor	12	0	13	17				
GraphvizDotVisitor	5	1	6	7	0	0	0	0
MessageNode	10	0	5	5				
MessageNodeVisitor	3	0	2	3				
Node	6	3	5	4	100	117	100	3
RewriteBindingNode	11	5	14	7	100	129	100	10
TextVisitor	5	2	4	8	50	0	50	0

ALIA4J-based prototype

TYPE	LINE	WMC	Ce	NOF	LCOM-PFI	LCOM-TC	LCOM-HS	LCOM-CK
AbstractBinding	33	13	21	4	100	112	89	4
AliaBindingFactory	8	5	8	0				
AttachmentCollectionReference	9	5	6	1	0	0	0	0
BindingManager	9	19	11	4	95	92	70	13
DefaultBinding	12	14	12	0				
EqualsSelectorPredicate	9	2	7	0				
NotEqualsSelectorPredicate	9	2	7	0				
RewriteBinding	14	10	12	2	100	92	75	1
SelectorExpressionTransformationVisitor	26	17	21	3	67	88	70	0
SelectorPredicate	14	3	11	2				
AbstractConstraint	9	7	6	4	100	116	90	9
AliaConstraintFactory	6	8	7	0				
CondConstraint	9	2	5	0				
ConstraintManager	9	7	7	1	0	0	0	0
PreConstraint	11	4	6	1	0	0	0	0
PrimitivePreConstraint	7	2	4	0				
PrimitiveSkipConstraint	7	2	4	0				
SkipConstraint	9	2	5	0				
AliaMessageHandler	13	5	8	0				