

Vrije Universiteit Brussel
Faculty of Sciences
Department of Computer Science and
Applied Computer Science



A Symbiotic Approach to Aspect-Oriented Logic
Meta Programming In a Prototype-based Language

Dissertation submitted in partial fulfillment of the requirements for the
degree of Licentiaat in de Informatica

Tom Leuse

Promotor:
Prof. Dr. Theo D'Hondt

Advisors:
Tom Van Cutsem
Stijn Mostinckx

June 6, 2005

Abstract

To date, most aspect-oriented programming technology is founded on class-based programming. Class-based object-orientation, however, covers only half of the object-oriented paradigm. The prototype-based subparadigm structures programs using objects alone, without requiring class abstractions. It is mostly this lack of classes why prototype-based languages have been left untouched by aspect technology: they define the structure required by conventional pointcut languages to describe crosscutting concerns.

The solution proposed in this dissertation is to augment the pointcut language such that it can describe crosscutting concerns based on the dynamic structure and behaviour of the program. Pointcuts are expressed in terms of the properties of objects and messages sent at run-time. However, the description of such pointcuts in terms of boolean expressions in the base language itself would nullify the benefits of e.g. a logic-based pointcut language, which enables a more declarative programming style known as logic metaprogramming to reason about the base program.

This dissertation reconciles the need for accessing dynamic information with a logic-based pointcut language through the use of a language symbiosis between the prototype-based base language and the logic-based pointcut language. The result is a pointcut language which fosters a declarative programming style while still being able to inspect the necessary dynamic information through symbiosis.

Abstract

Tot op heden zijn de meeste bestaande technologieën voor aspectgeoriënteerd programmeren gebouwd op klassegebaseerd programmeren. Klassegebaseerd objectgeoriënteerd programmeren omvat echter slechts de helft van het objectgeoriënteerde paradigma. Het prototype-gebaseerde subparadigma structureert programma's door enkel gebruik te maken van objecten, zonder dat er nood is aan klasse-abstracties. Het is vooral dit gebrek aan klassen dat ertoe geleid heeft dat prototype-gebaseerde talen onaangeraakt zijn gebleven door aspect-technologie: klassen definiëren structuur die vereist is om crosscutting concerns te beschrijven in conventionele pointcut-talen.

De oplossing die voorgesteld wordt in deze thesis is om de pointcut-taal te versterken zodat ze crosscutting concerns kan beschrijven aan de hand van de dynamische structuur en het gedrag van een programma. Pointcuts worden uitgedrukt in termen van de eigenschappen van objecten en berichten die gestuurd worden tijdens de uitvoering van het programma zelf. De beschrijving van zulke pointcuts in termen van booleaanse expressies in de basistaal zelf zou echter de voordelen van bvb. het gebruik van een logische pointcut-taal ongedaan maken, die toelaat om een meer declaratieve programmeerstijl, gekend als logisch metaprogrammeren, te gebruiken om te kunnen redeneren over het basisprogramma.

Deze thesis verzoent de nood aan dynamische informatie met een logische pointcut-taal door het gebruik van taalsymbiose tussen de prototype-gebaseerde basistaal en de logische pointcut-taal. Het resultaat is een pointcut-taal die een declaratieve programmeerstijl toelaat, en toch de mogelijkheid behoudt de nodige dynamische informatie te verkrijgen via symbiose.

Acknowledgements

I would like to start by thanking Prof. Dr. Theo D'Hondt for promoting this research. A lot of thanks also go to my advisors, Tom Van Cutsem and Stijn Mostinckx, for all their ideas, the numerous discussions we had, and all the support they gave me when I had problems writing or implementing. Even more thanks to them for proofreading this dissertation, sometimes even on very short notice.

I would also like to thank Kris Gybels for helping me with my implementation and sending me interesting papers, and all other members of the Programming Technology lab for their interesting feedback during the various presentations.

Other thanks go to the *Vrije Universiteit Brussel*, and especially the Department of Computer Science, for the education and for showing me the numerous facets of computer science.

Thanks also to my girlfriend Vicky for supporting me and keeping me motivated to keep on writing, and to all my friends for giving me the opportunity to think about other stuff than aspect-oriented logic meta programming every now and then.

Last, but certainly not least, I would like to thank my parents for giving me the opportunity to get an excellent education, and for providing an excellent environment for me to work in.

Contents

1	Introduction	1
1.1	Document structure	2
2	Prototype-Based Languages	4
2.1	Prototypes	4
2.2	Class-based versus prototype-based languages	5
2.2.1	Delegation	5
2.2.2	Object creation	6
2.2.3	Advantages and disadvantages of prototype-based languages	6
2.2.4	Summary	8
2.3	Self	8
2.3.1	Concepts	9
2.3.2	Prototype-based idioms in Self	10
2.3.3	Summary	12
2.4	The Pic% Programming Language	12
2.4.1	Pico	12
2.4.2	Pic%	18
2.5	Conclusion	23
3	Aspect-Oriented Software Development	24
3.1	Crosscutting Concerns	24
3.2	Solutions to the crosscutting problem	27
3.2.1	Inheritance	27

3.2.2	Aspects	28
3.2.3	Composition Filters	28
3.3	Aspect-Oriented Programming	31
3.3.1	AspectJ	33
3.3.2	JAsCo	37
3.4	Aspects and prototype-based languages	39
3.5	Conclusion	40
4	Language Symbiosis	41
4.1	Symbiosis	41
4.1.1	Definition	41
4.1.2	Language Symbiosis vs. Integration	42
4.1.3	Use of symbiosis	42
4.2	Linguistic Symbiosis	42
4.2.1	Approaches to linguistic symbiosis	43
4.2.2	Summary	44
4.3	Case studies in Language Symbiosis	45
4.3.1	The Agora Framework	45
4.3.2	Smalltalk and SOUL	48
4.3.3	Lillambi	50
4.4	Conclusion	51
5	Symbioco	52
5.1	Loco	52
5.1.1	The declarative programming paradigm	52
5.1.2	The Loco programming language	54
5.1.3	Summary	56
5.2	Symbioco	57
5.2.1	Goal	57
5.2.2	Approach	57
5.2.3	Translating expressions	58
5.2.4	Summary	62

5.3	Validation	62
5.3.1	The Rijmenam Example	62
5.3.2	Implementation	62
5.3.3	Summary	65
5.4	Technical Aspects of Symbioco	66
5.4.1	Combining multiple interpreters	66
5.4.2	Using errors as symbiosis joinpoints	67
5.4.3	Switching between processes	68
5.5	Conclusion	70
6	Prototype-Based AOLMP	71
6.1	Aspect-Oriented Logic Meta Programming	71
6.1.1	Using a declarative language to describe pointcuts	72
6.1.2	Example	74
6.1.3	Summary	74
6.2	Aspects in Symbioco	75
6.2.1	Introducing aspects in Pic% and Loco	75
6.2.2	The Symbioco Aspect Weaver	76
6.2.3	Loco as a pointcut language	77
6.3	Validation	77
6.4	Symbiotic Pointcut Predicates	79
6.4.1	Joinpoint representation in Pic%	79
6.4.2	The Pic% joinpoint predicate library	79
6.4.3	The use of language symbiosis	80
6.5	AOLMP with Prototypes	81
6.6	Technical aspects of Aspects in Symbioco	81
6.6.1	The Symbioco Aspect Weaver Revisited	81
6.6.2	Weaving around advice	82
6.7	Conclusion	87

7	Conclusions	89
7.1	Evaluation	90
7.1.1	Contributions	90
7.1.2	Limitations	92
7.2	Future research	93
7.2.1	Weaver improvements	93
7.2.2	Prototype-based Idioms	93
7.2.3	Representing Pic% programs as Loco facts	93
	Bibliography	94

List of Figures

2.1	Late binding of self (taken from [40])	6
2.2	Pico's environment model [16]	18
2.3	Pic%'s environment model [16]	21
2.4	Shared constants [16]	22
3.1	XML Parsing in org.apache.tomcat	25
3.2	URL Pattern Matching in org.apache.tomcat	25
3.3	Logging in org.apache.tomcat	26
3.4	Graphical representation of the Composition Filters model [5]	29
3.5	Message filtering graphically represented [4]	30
3.6	Graphical representation of the weaving process [18]	32
3.7	Informal depiction of dynamic joinpoints	34
3.8	Point Synchronization example with delegation [8]	39
4.1	Language Symbiosis vs. Language Integration	42
4.2	Symbiosis between a language and its implementation language (left) versus symbiosis between two languages in a common implementation language (right)	44
4.3	Language Symbiosis between Agora and its implementation language [31]	46
4.4	Graphical representation of the lillambi setup [12]	50
5.1	The London Underground [19]	54
5.2	Representation of the Symbioco-layer	58
5.3	Conceptual representation of a Pic% (left) and a Loco (right) parsetree	60

5.4	Graphical representation of Rijmenam domain knowledge [14]	63
6.1	Java code for a Stack versus TyRuBa representation of this Java code [13]	73
6.2	Weaving Pic% advice	82
6.3	Preparing install of extended dictionary	83
6.4	Evaluate advice code	84
6.5	Calling <code>proceed</code>	85
6.6	Restoring original dictionary	85
6.7	Proceeding with original call	86

Listings

2.1	Stack implementation in Self [37]	10
2.2	Traits in Self [40]	11
2.3	Pico-implementation of the bubblesort algorithm	13
2.4	Some operator examples	15
2.5	Example of apply-operator	16
2.6	Pico Booleans	17
2.7	Pic% object creation	19
2.8	Dot-operator in use	19
2.9	Nested mixin methods	19
3.1	Example code for Dynamic Joinpoints	33
3.2	Example of a pointcut	36
3.3	Example of advice	36
3.4	Example of an aspect	37
3.5	JAsCo aspect bean	38
3.6	JAsCo connector	38
4.1	Invoking Smalltalk from within SOUL	48
4.2	Invoking SOUL from within Smalltalk	49
4.3	Old versus new SOUL syntax[23]	49
5.1	Connections	54
5.2	Nearby	55
5.3	Nearby rules	55
5.4	Querying the system	55
5.5	Representing knowledge in Loco (1)	62

5.6	Representing knowledge in Loco (2)	63
5.7	Finding paths in Loco	64
5.8	Finding a path's length in Pic%	64
5.9	Finding the shortest path in Pic%	64
5.10	Rijmenam examples	65
6.1	A simple Pic% login object	78
6.2	Example of a logging aspect	78
6.3	Subset of the Pic% joinpoint library	79
6.4	<code>stateChanging</code> rule	80

List of Tables

2.1	Basic Pico Syntax [10]	14
4.1	A catalog of objects in Agora's symbiosis model [31]	47

Chapter 1

Introduction

In this dissertation we will discuss the benefits of a symbiosis between the pointcut language and the base language using aspect-oriented logic meta programming in a prototype-based language. The thesis statement can be broken down into two parts: the use of aspect-oriented logic meta programming to allow aspects in a prototype-based language, and the use of a symbiosis between the crosscut language and the prototype-based language we wish to use aspects in.

Aspect-oriented programming [27] is based on the concept of crosscutting concerns. These are concerns in a program that cannot be addressed in a single class (or a hierarchy of classes), but are in fact spread over a numerous amount of classes. A typical example of a crosscutting concern is the logging concern: in regular object-oriented programming, logging code is usually spread over all classes that want to log something.

Aspects address this matter by allowing the programmer to specify code that needs to be executed at a certain point, called a *joinpoint*, during program execution outside classes. The joinpoints at which additional code needs to be executed are described using a so-called pointcut language.

To date, most aspect-oriented programming technology is founded on class-based programming. The main reason for this is the structure offered by classes: most pointcut languages are based on this static structure.

Prototype-based languages [30] are a subset of object-oriented languages, that do not feature classes. Instead, objects are used to both build abstractions, and execute programs. Prototype-based languages are more dynamic and flexible than their class-based siblings, but provide less static structure. It is this lack of static structure that poses a problem for aspect-oriented programming, since this structure is used to describe joinpoints.

The solution proposed in this dissertation is to augment the pointcut language such that it can describe crosscutting concerns based on the dynamic structure and behaviour of the program. A logic programming language is chosen as pointcut language, because this makes reasoning about the base program easier. Using a logic programming language to reason about programs is called logic metaprogramming [41].

Logic metaprogramming has already been applied to aspect-oriented programming, which resulted in aspect-oriented logic metaprogramming [13]. However, instead of reasoning about a static representation of a program using facts, in our approach we will use AOLMP at runtime. Additionally, a symbiosis will be constructed between the logic pointcut language and the base language.

Language symbiosis between two languages [23, 31] means that both languages can use each others functionality. For instance, this can enable two object-oriented languages to send messages to one another's objects, or enable an object-oriented programming language to execute queries in a declarative language.

In our approach a symbiosis between a logic pointcut language and a prototype-based base language is constructed, in order to provide the pointcut language with the ability to inspect dynamic information in the base language.

Pic%, a prototype-based programming language developed at the Vrije Universiteit Brussel, will be used as base language in our experiment. Pic% is based on Pico, a very simple and expressive language originally intended to teach programming concepts to non-computer science students. Pic% manages to hold on to the expressiveness and simple syntax featured in Pico.

Loco is the logic programming language that will be used as pointcut language to describe pointcuts in Pic%. Like Pic%, Loco is also developed at the Vrije Universiteit Brussel. Loco syntactically resembles Pico (and Pic%), which will prove to be very useful when constructing the symbiosis.

In our experiment, Loco will be triggered if interesting events occur in Pic%. Loco will reason about the Pic% program, and will decide (possibly using Pic% symbiotically) whether advice needs to be woven in.

1.1 Document structure

This dissertation is structured as follows:

In the next chapter prototype-based languages will be discussed. We will explain what a prototype-based language is and where these languages come from. At the end of the chapter two prototype-based languages are discussed in more detail, Self and Pic%.

Chapter three will describe aspect-oriented software development. The problem of crosscutting concerns will be discussed, and some approaches to aspect-oriented programming will be shown.

In chapter four a brief look will be taken at language symbiosis. This concept will be defined, and the distinction with language integration will be shown. The chapter will then be concluded with some examples of language symbiosis.

The fifth chapter revolves around the symbiosis between Pic% and Loco, called Symbioco, a proof-of-concept implementation supporting the thesis. In the chapter we will take a closer look at Loco, the logic programming language that is used in this symbiosis. Then the actual symbiosis will be discussed, after which a validating example will be given. The chapter will be concluded with a more technically detailed discussion of the symbiosis.

The sixth chapter deals with the addition of aspect-oriented logic meta programming in the symbiosis from chapter five, and uses roughly the same structure as chapter five. First aspect-oriented logic meta programming in general will be discussed, after which a closer look will be taken at how this was added in Symbioco. After this a validating example will be given and the chapter will be concluded with a more technically detailed discussion of the aspect weaver.

The seventh and final chapter presents our conclusions and identifies areas for future work.

Chapter 2

Prototype-Based Languages

As said in the introduction, the goal of this dissertation is to weave aspects into a prototype-based language. Therefore this chapter will briefly introduce prototype-based languages. The chapter will start by explaining what constitutes a prototype-based language, and where these languages come from. Then we will compare this prototype-based approach with more known class-based systems. In the end two examples of prototype-based languages will be discussed. First we shall take a brief look at Self, which is the most evolved prototype-based language to date. Then we will thoroughly discuss Pic%, the prototype-based language that will be used throughout this dissertation. In order to properly discuss Pic%, we will also introduce Pico, the language upon which Pic% is based.

2.1 Prototypes

Although object-oriented programming is currently very popular, only one branch of object-oriented programming is widely used today, namely class-based programming as in Java or Smalltalk. A completely different paradigm within the boundaries of object-oriented programming, namely prototype-based programming, is therefore completely ignored.

Prototype-based programming is based on the idea that objects can be thought of in terms of a few representatives (with some differences). These representatives are called prototypes. In order to create a new object, a prototype can be selected and cloned. The use of prototypes instead of classes is easily defensible: when reasoning about concepts people naturally tend to think in prototypes [36]. For instance, when thinking about an object oriented language most people think about a typical example of such a language (e.g. Java), instead of thinking about an enumeration of classifying descriptions of an object oriented language. In the same way it can be said

that class-based programming is a 'prototype' of object-oriented programming.

2.2 Class-based versus prototype-based languages

Now that we know what prototypes are, and where they come from, we will compare prototype based languages with their class-based siblings. This comparison will be made based on the concepts that were introduced in the Treaty of Orlando [29]. The Treaty introduced 2 basic concepts: *Empathy* and *Templates*. Empathy describes how it is expressed that an object resembles another object, and thus reuses behaviour of this "parent" entity. Templates generate objects from their own image, allowing us to generate several objects that look like each other, possibly with the exception of a small amount of state.

We will conclude this section with an overview of the advantages and disadvantages of the prototype-based approach.

2.2.1 Delegation

The first concept of the Treaty we will discuss, it empathy. Empathy is implemented in prototype-based and class-based languages using respectively the delegation [28] and inheritance. Delegation, as opposed to class inheritance which merges the structural description of all classes in the inheritance chain into a single object, is a relation between two objects where either can be addressed in its own right. When a message is not understood (i.e. the corresponding method is not implemented) by the child object *c* (the one that is lowest in the delegation hierarchy), it delegates the call to its parent *p* (messages can also be delegated to multiple parent objects), together with a reference to itself. This self-reference can then be used to send future self-sends to *c* instead of *p*. This is achieved due to the late binding of *self*, and distincts delegation from ordinary message forwarding. We will illustrate this with an example. Suppose a message *m1* is sent to *c*, which doesn't understand *m1*. The object will delegate this message to its parent, *p*. Suppose *p* implements *m1* as sending a message *m2* to *self*. With simple message forwarding, *m2* would be invoked on *p*. Using delegation, however, *self* will be bound to *c*, the original object where *m1* was sent to, and *m2* will be invoked on *c*. Figure 2.1 illustrates the principle of late binding, and shows the difference with simple message forwarding (which is what the Delegation Design Pattern [20] explains, unfortunately causing some confusion about delegation).

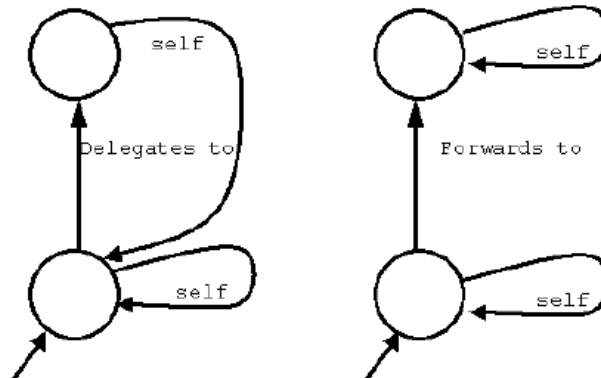


Figure 2.1: Late binding of self (taken from [40])

2.2.2 Object creation

The second concept introduced in the treaty is templates. Recall from the beginning of this section that templates allow us to generate several objects that look like each other. In class-based environments the role of templates is filled in by classes: objects are generated by instantiating classes. A prototype-based language enables object creation by cloning prototypes. It is also possible to create objects *ex nihilo* (out of nothing), in order to make new prototypes. Using delegation these new prototypes can be variations on existing ones.

2.2.3 Advantages and disadvantages of prototype-based languages

Now that we have discussed the biggest differences between prototype-based and class-based languages, we will give an overview of the advantages and disadvantages of prototype-based languages. A similar overview of these advantages and disadvantages can be found in [9].

2.2.3.1 Advantages

Using prototypes instead of classes gives us a lot of advantages, the most important of which will be listed here.

Simplicity Prototypes simplify designing programs, since only objects need to be taken into account, related only through delegation and regular

composition. The entire system of classes linked through inheritance does not need to be specified.

It is also simpler to do meta-programming in a prototype-based language. This is due to the fact that an object contains all the information necessary to do meta-programming, where in class-based systems the class of the object also needs to be specified. When representing this class as an object to work with, what is this class-object's class? As can be seen this leads to an infinite amount of meta-classes, a problem that prototype-based languages don't have.

Per-object behaviour The behaviour of objects can be specified separately for every object. This makes it possible to change the behaviour of a single object (e.g. debug-code in a faulty object), where in class-based programming all objects of a class would be affected at once. It also allows us to easily express a singleton pattern, since unlike in class-based systems we do not need to define an additional class that is generally of no use except to create the singleton.

Initialization Thanks to the cloning process, there is no need to worry about partially filled out fields in objects. Since a complete object is cloned, all its fields are taken over, ensuring that no variables are left uninitialized.

Dynamism Prototype-based languages are usually more dynamic than their class-based counterparts. In most prototype-based languages it is allowed to dynamically add and modify fields (both variables and methods) of an object, and even to dynamically restructure the delegation hierarchy between objects (which gives us the state pattern "for free", as will be shown later on).

2.2.3.2 Disadvantages

Given all the advantages described above, why doesn't everybody use prototype-based languages? Why didn't they become a grand success like their class-based siblings? The reason is that they also have some significant shortcomings. Therefore we will now list some "features" from class-based languages that seem to be missing in some prototype-based languages.

Relations between objects Class-based languages offer a very natural way of classifying objects. It is usually very important to speak about a set of related objects, sharing a representation or having a common interface. In the prototype-based model using just delegation and cloning, it is nigh impossible to keep track of these relations, not in the

least because there is usually no system-maintained relation between objects and their clones.

Abstract concepts Another problem of prototype-based languages is that they make it hard to represent inherently abstract concepts. Though representing concrete things as abstract classes can be very difficult at some times, prototypes simply enforce one to describe *everything* in a concrete way. To illustrate this, consider a stack. Everyone knows what a stack is and how it functions, but is a stack really a clone of a certain prototype, of some sort of ideal stack? This way of thinking can be awkward sometimes.

Distinction between structure and behaviour Finally, in class-based languages there is a clear distinction between the structure of an object (which is the object itself) and the behaviour (which is defined in the object's class), making it easier to protect the behaviour. Prototypes do not enforce this distinction. Though this is not always a benefit (as is shown in section 2.2.3), we can also foresee problems when a set of objects depending on a prototype is declared, and this prototype suddenly gets modified¹. Therefore some protection on such prototypes is required. Not all changes to the parent should be forbidden, though, as these changes are considered to be essential to programming with prototypes.

2.2.4 Summary

We have now compared prototype-based languages with class-based languages, and seen the strengths and weaknesses of the prototype-based approach. In the following sections we will discuss two prototype-based languages: Self and Pic%. Self is the most evolved prototype-based language to date, so it is hard to write anything about prototype-based languages without at least mentioning Self. Pic% is the language that will be used throughout this dissertation, so it will be discussed elaborately.

Another prototype-based language, Agora, will be discussed later, in chapter 4.

2.3 Self

Self [39] was mainly developed by Sun Microsystems, and is the most evolved prototype-based language to date. Self can be seen as a transposition of

¹This problem was already observed in [28], and has been dubbed the *prototype corruption problem* in [6]

Smalltalk into the the prototype-based world, although it is significantly more minimal than its class-based counterpart.

Self strives for *uniformity* in all of its language features. This devotion gives rise to Self's expressive power [39].

In this section a brief overview of the Self programming language is given. First the core language concepts will be described, after which some interesting programming idioms which first emerged through programming experience in Self will be discussed.

2.3.1 Concepts

In this section, the core language concepts of Self will be described. As mentioned before, Self strives for minimality. Therefore it is not surprising that there are only two entities in Self: objects and messages. Other language features, like for instance variables, have been repolished to fit this minimalistic approach.

Replacing variables by methods

In Self, all computation is achieved by sending messages among objects. In Self, message passing is the fundamental operation [39]. Self does not contain variables, instead objects are stored in so-called *slots* which can be manipulated through message passing. When a slot is declared in Self, the Self system will automatically create appropriate accessor methods. This implies that variable access is completely replaced by message passing: objects simply send the accessor method for a certain slot to themselves.

Self eliminates the distinction between variable access and message passing by unifying them, thereby making message passing and inheritance more powerful [39]. The reason for this is that children of a certain object can *refine* accessor methods, providing those “variables” with a different behaviour. Note that this also means that variables can be overridden, simply because variables are replaced by methods. In many other object-oriented languages the difference between methods and variables is the reason that variable overriding is impossible.

Objects

Self features three types of objects: ordinary objects, method objects and block objects. Objects can be created ex-nihilo by listing a number of slots between bars, separated by dots (this is called a *slot list*). A point object for instance can be represented by `|x . y|`. If such a construct is followed by

an expression, the result is a method object. The slots declared in the slot list preceding the expression will act as local “variables” for the method. Local slots can also be preceded by a colon, which makes the slot a parameter of the method.

Objects are able to *delegate* messages to their parent objects. Parents of an object can be defined by annotating a slot with an asterisk. Messages not found in the receiver will automatically be forwarded to the parent, with late binding of self (also see section 2.2.1). Self also features multiple inheritance, meaning that multiple parents can be defined for a single object. In this case method lookup becomes more complex because of the need to disambiguate slots in multiple parents.

An example of a stack implementation taken from [37] is shown below.

Listing 2.1: Stack implementation in Self [37]

```
aStack <- (|
  stack = array clone .
  top = 0.
  push: obj = (top: (top+1). stack at: top Put: obj).
  pop = (top: (top-1)).
  getTop = (stack at: top)
|)
```

The third type of objects, *block* objects are the equivalent of a so-called closure, which encapsulates a method together with a pointer to its enclosing environment. This is useful for passing around code together with its environment of definition.

2.3.2 Prototype-based idioms in Self

This section will describe a number of techniques or idioms on how to structure prototype-based programs. These idioms show that it is possible to organize programs just as well without classes in a prototype-based language like Self [38].

Traits

In order to attain structured and reusable software, it is necessary that state and behaviour can be *shared* among instances of the same type (or clones of the same prototype). Sharing allows the programmer to change the behaviour of an entire set of objects with just a few strokes [39]. If no sharing is used, code will be duplicated which might lead to inconsistent objects.

This problem is solved in Self through *parent sharing*. Code is factored out by placing it in a separate object, and all objects that wish to share this

code use this objects as parent. Such an object is called a *traits object*. Its role is similar to that of a class in class-based languages.

However, it is possible to argue that traits in prototype-based languages actually fulfill the role of classes, thus that classes are necessary to structure programs. It can also be interpreted the other way around: it is possible to express class-like features in prototype-based languages, yet this structure is not imposed and the programmer is free to choose [40]. An example of how traits can be used is shown below.

Listing 2.2: Traits in Self [40]

```
stackTraits ← (|
  push: obj = (top: (top+1). stack at: top Put: obj).
  pop = (top: (top-1)).
  getTop = (stack at: top)
|)

stackPrototype ← (|
  parent* = stackTraits.
  stack = array clone.
  top = 0.
|)
```

In this example all behaviour is factored out into the traits object, which will be shared by all concrete stacks through the parent pointer. Concrete classes hold the actual state.

The traits technique brings some advantages with it. Behaviour of a certain object is specified only once, leading to improved reusability and consistency. Additionally, by explicitly defining a prototypical stack (as in the example), this prototype can always be used to create new stacks. Without traits it would become confusing which stack would be a good (i.e. empty) stack to clone from.

However, even with the traits technique, there is nothing that prevents users from changing the prototypical stack, which means that this scheme is sensitive to the prototype corruption problem [6].

Refinement through delegation

Traits, as seen in the previous section, are just objects. This means that there is no reason why they should not be allowed to inherit behaviour from other objects. Traits inheriting functionality from other traits can be regarded as the equivalent of code sharing via subclassing in class-based languages [40]. Generally new data types in classless languages can easily be derived from existing data types by refining the corresponding traits objects [38].

However, when both state and behaviour need to be inherited, this poses some problems. State and behaviour are kept in separate objects in Self, implying that an object that wants to inherit both must do so by inheriting from two different objects. Thanks to Self's support for multiple inheritance, however, this poses no real problem. An object can simply declare a "data parent" and a "traits parent" [39].

Changing object behaviour

Another example of Self's dynamic object model is the implementation of dynamic behaviour. It frequently occurs that a certain object needs to respond to a message differently, depending on the state the object is in. In Self this can easily be done by changing the object's parent (supposing this parent implements the behaviour). This is possible since in Self, parent slots can be assigned to. This is actually a very expressive way to implement the State design pattern [20].

2.3.3 Summary

Self is a very pure object-oriented language, much in the spirit of Smalltalk [40]. The fact that Self is very minimalistic makes it very simple. Almost everything, including flow control, variable scoping, method invocation and primitive data types, consists of just objects and messages [39]. This simplicity is augmented even more by the use of prototypes instead of classes. Self is one of the most mature prototype-based languages to date, and has been used to prove that it is possible to organize programs without classes [38].

2.4 The Pic% Programming Language

Pic% (which is pronounced Pico-o) is an object-oriented, prototype-based extension of Pico [16]. In order to properly discuss Pic%, we will first take a closer look at Pico.

2.4.1 Pico

Pico is a language developed at the Vrije Universiteit Brussel by Prof. Dr. Theo D'Hondt. It was originally intended to teach programming concepts to students in other sciences than computer science. Therefore, it has been designed to be very simple, both conceptually and syntactically.

Pico borrows a great many concepts from Scheme conceptually [10]. Due to the representation of programs as lists [1] in Scheme however, its syntax

can be rather confusing to comprehend. Therefore Pico’s syntax has been based on calculus, rather than on Scheme, and can in fact be seen as “a non-trivial marriage between the power of languages like Scheme, and the standard infix notation as it is used in calculus” [15].

Ever since it was developed, Pico has been the target of various modifications and extensions, one of which is Pic%.

We will start this discussion by giving an overview of the basic syntax of Pico using a small example, after which we will briefly discuss the extended syntax. We will then take a look at Pico’s lazy argument evaluation, to conclude the Pico discussion with a short section on meta-programming.

2.4.1.1 Basic syntax

The syntax of Pico will be discussed using the code example shown below, which is an implementation of the bubblesort algorithm.

Listing 2.3: Pico-implementation of the bubblesort algorithm

```

1 bubblesort ( table ) : {
2   changed : true ;
3   swap : void ;
4   while ( changed ,
5     { changed := false ;
6       i : 2 ;
7       while ( i <= size ( table ) ,
8         { if ( table [ i - 1 ] > table [ i ] ,
9           { swap := table [ i - 1 ] ;
10            table [ i - 1 ] := table [ i ] ;
11             table [ i ] := swap ;
12             changed := true
13           } ) ;
14            i := i + 1 }
15         )
16     } ) ;
17   table
18 }
```

This example defines a function named `bubblesort`, taking a single parameter `table`. Afterwards, in the same fashion some variables are defined. Apart from functions and variables, there is a third kind of values used in the above example, namely tables (line 8 till 11). Tables are used in the same way as Scheme’s cons-cells, as data containers and to represent programs (as shown in section 2.4.1.4).

Each of these values has a syntactical representation, called invocation. All these invocations can be defined using a colon (as can be seen in the first

three lines of the example). Defining an invocation means that its name is added and bound to a value in the Pico-dictionary (comparable to a Scheme environment). In fact, all of Pico's syntax is built around the Pico-dictionary. Apart from being able to add names to the dictionary, we can also retrieve (or reference) names and modify (or assign to) them. This is shown in table 2.1.

	Reference	Tabulation	Application
Reference	v <i>Variable reference</i>	$t[exp_1]$ <i>Table indexing</i>	$f(exp_1, \dots, exp_n)$ <i>Function call</i>
Definition	$v : exp_2$ <i>Variable definition</i>	$t[exp_1] : exp_2$ <i>Table definition</i>	$f(exp_1, \dots, exp_n) : exp_{n+1}$ <i>Function definition</i>
Assignment	$v := exp_2$ <i>Variable assignment</i>	$t[exp_1] := exp_2$ <i>Table modification</i>	$f(exp_1, \dots, exp_n) := exp_{n+1}$ <i>Function redefinition</i>

Table 2.1: Basic Pico Syntax [10]

As can be seen in the example, these basic operations make up for the biggest part of the code. The only apparent exceptions in our example are the use of infix-notation (e.g. `i+1`), and the use of curly braces. However, the next section will illustrate how this extended syntax is treated by Pico and show that it coincides with the table shown above.

When examining the example, one might also notice that the semicolon is used as a separator between different expressions, unlike Java or C++ where expressions are terminated by semicolons. This means that between 2 expressions there is always a semicolon, but there is no semicolon after the last statement (as can for example be seen at line 12). This may seem odd to people that are used to languages like Java, but it is actually very natural and clean syntax (to illustrate this note that most programming languages don't require a comma after the last argument passed to a function either).

2.4.1.2 Extended syntax

In the previous section it was shown that the nine operations from table 2.1 made up for the biggest part of most Pico code. Besides these operations, there was also some extended syntax. In this section we will discuss this extended Pico-syntax, and show that it is not very different from the syntax in table 2.1. First we will discuss operators, the extended syntax that is responsible for infix notation. We will then briefly talk about some notational shorthands, like for instance the curly braces we've seen in the previous section. Finally, the apply-operator will be introduced.

Operators In the previous section there was already an example of infix notation ($i+1$). This notation is a shorthand for the application of the $+$ function to arguments i and 1 . This means that there is no difference between $i+1$ and $+(i, 1)$. Since the Pico-parser parses these 2 expressions in the exact same way, it is also possible to define infix operations. However, this infix notation doesn't work for every function. Consider the function `sum`, taking 2 arguments and returning the sum of those arguments (which is comparable to what the $+$ function does). When calling this function using `arg1 sum arg2`, an error will be thrown by the parser. The only way to call this function is using prefix notation: `sum(arg1, arg2)`. The reason for this is that infix notation is only supported for operators.

Operators are a special kind of function names, containing only symbols from the set $\{\$, \%, +, -, |, \sim, \&, *, \backslash, /, !, ?, \wedge, \#, <, =, >\}$. Some examples of operators and how they are used can be seen in code listing 2.4.

Listing 2.4: Some operator examples

```

1 |(x,y): (x + y) / 2
2 !n : if(n=0,1,n * !(n-1))
3 x <%#!+-*> y: x^2 + y^2
4
5 4 | 6          -> 5
6 |(4,6)        -> 5
7 !5           -> 120
8 3 <%#!+-*> 4  -> 25

```

The first 3 lines are definitions of operators. The first one calculates the average of its arguments, the second one is a recursive implementation of a factorial, and the last one returns the sum of the squares of its arguments. On the first line it can be seen that it is possible to define an operator in the same fashion as a regular function would be defined. The third definition shows that the infix notation can also be used for this. In this definition it is also shown that there can be any amount of operator-characters in the operator's name. On the second line an example of a unary operator is shown. The four lower lines are calls to these definitions. The first two calls illustrate that you can choose freely between infix or prefix notation, as was said above.

Notational shorthands Another part of Pico's extended syntax consists of 2 notational shorthands. In the example from the previous section we already came across the curly braces, so we will first elaborate on those. In Pico, just like in Scheme, compound expressions can be constructed using

a call to the native function² `begin`. `begin` accepts one or more arguments, evaluates them one by one, and returns the last value. Constantly using `begin` can become rather cumbersome however, which is why Pico provides syntactic sugar to simplify constructing compound expressions in the form of curly braces.

$$\text{begin}(exp_1, exp_2, \dots, exp_n) \equiv \{ exp_1; exp_2; \dots; exp_n \}$$

In the same fashion there is a notational shorthand for creating tables. Using nothing but basic syntax, a table can be created using the `table` native. Pico allows the use of square brackets to create tables more easily.

$$\text{table}(exp_1, exp_2, \dots, exp_n) \equiv [exp_1, exp_2, \dots, exp_n]$$

Apply-operator The last syntax extension we will discuss is the use of the apply-operator. In a Pico implementation, function argument lists are represented as tables. This enables the programmer to write functions with variable sized argument lists at no extra cost [10]. To do this, the symbol `@` is used as shown in the following code example. The function `sum` takes an arbitrary number of arguments, and returns their sum.

Listing 2.5: Example of apply-operator

```

1 sum@args: {
2   res: 0;
3   for(i:1, i<=size(args), i:=i+1,
4     res := res + args[i]);
5   res
6 }
```

The apply-operator can also be used to define the native function `table` from the previous paragraph.

```
table@list: list
```

As mentioned above, the `table` function takes a list of arguments, and returns a table containing those arguments. Since the argument list of the `table` function is represented as a table of values, all that needs to be done is returning this argument list.

Another useful application of the apply-operator is the implementation of the native function `begin` [10, 11]:

```
begin@list: list[size(list)]
```

²A native function is a function that is directly implemented in the interpreter.

Recall that `begin` takes a list of expressions as arguments, evaluates them all, and returns the result of the last expression. This can be simulated through the use of `@`, since Pico uses eager parameter binding, and since argument evaluation always happens from left to right. Note that this also means that arguments passed to the `table` function, and by consequence expressions between the square brackets syntactic sugar, are evaluated before an actual table is returned.

These examples show the ease and expressiveness of Pico: even basic language constructs can be expressed in a single line of code.

2.4.1.3 Lazy Argument Evaluation

As said before, Pico uses eager argument evaluation. However, in some cases we want to be able to use lazy argument evaluation. This is made possible in Pico using functional parameters. When the formal parameters of a function are variables, the semantics is *call-by-value* (as in Scheme). However, if the formal parameters are function applications, the resulting semantics are *call-by-expression* [11]. We will illustrate this using the following example:

```
f(g(x,y), z): g(1,2) + z
```

In this example a function `f` is defined, taking two arguments. The second parameter, `z`, is a normal variable reference. The first one, however, is a functional parameter. When calling `f` its first argument will be treated as being the body of a function, and bound to `g`. This means that it will *not* be evaluated, and thus we obtain lazy evaluation. Suppose we call

```
f(x+y, 3)
```

a new local function is defined in the scope of `f`: `g(x,y): x+y`. When executing the body of `f`, `g(1,2)` will be evaluated using this function and return 3, `f` will thus return 6. Because `g` is defined at the time `f` is called, the scope of `g` is the scope of `f`'s caller. This implies that the caller can use variables visible in his scope as free variables in the body of `g`.

In order to demonstrate the power and expressiveness of this mechanism, note that Pico's boolean system and its control structures are defined inside Pico themselves (as opposed to Scheme where "special forms", i.e. functions with a dedicated interpretation, are needed). Booleans are defined as Church booleans, as in the λ -calculus.

Listing 2.6: Pico Booleans

```
true(t(), f()): t()
false(t(), f()): f()
if(cond, then(), else()): cond(then(), else())
```

Without the functional parameters to delay evaluation, we would not be able to express such a definition of `if` in Pico.

2.4.1.4 Meta-programming and reflection in Pico

In general, all programs deal with data relating to a certain domain. This domain can be a bank, a physical system, and any other human endeavour where computers are applied. Meta programs are then defined as programs where the domain is another program. Reflection is a special case of meta-programming where the domain of a program is the program itself. [24] Pico offers some powerful features for meta-programming. It introduces the three core evaluator functions (*read*, *eval*, *print*) for use within the language.

However, the most beautiful meta-programming technique is the unification of Pico programs with Pico data structures (tables), resembling how in Scheme programs are represented using Scheme data structures (lists). An example of this is the internal representation of a function: it is in fact a table of size 4, containing a name, argument list, body and a dictionary representing the lexical environment. This dictionary is then represented as another table, this time of size 3, containing a reference, the actual value, and a pointer to the next dictionary. This means that Pico is able to access these functions, dictionaries and all other compound values as tables, allowing for reflection.

2.4.2 Pic%

2.4.2.1 Objects

The first step to making an object-oriented language out of Pico is obviously to add objects. This was done in a very simple way, by using Pico's environment model. Pico environments (or dictionaries) are first-class, so they can be used in order to represent objects as a collection of slots, which bind a name to a certain value. This is graphically represented in figure 2.2. In

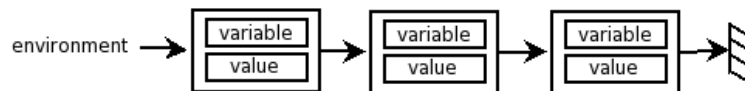


Figure 2.2: Pico's environment model [16]

order to easily use this model a single native function, *capture*, was added. This function simply returns a clone of the current environment. [16] An object can now be defined as is shown in the following listing.

Listing 2.7: Pic% object creation

```

counter(n): {
    up(): n := n+1;
    down(): n := n-1;
    capture()
}

```

An object can now be created using:

```
c: counter(0).
```

This will lead to `c` being bound to a `counter` object, which is simply the scope at the moment of the call to `capture` in the object definition. When the `counter` function gets executed, an extension of the function's scope will be created, in which the `n` parameter will be bound, as is done for each function application. All definitions that occur while executing the function will also be added in this extended scope, so in our example the extended scope for a call to `counter` will contain the variable `n`, and the `up` and `down` functions. It is this scope that the call to `capture` will return.

In order to access methods and variables inside objects, a dot-operator is used for name lookup in a Pico environment, resulting in a simple message passing semantics.

Listing 2.8: Dot-operator in use

```

c1: counter(10);
c2: counter(100);
(c1.up()) + (c2.down())

```

2.4.2.2 Inheritance & Cloning

In order to support inheritance, nested mixin methods were adopted from the language Agora [31]. An example of these mixin methods is shown below.

Listing 2.9: Nested mixin methods

```

counter(n): {
    up(): n := n+1;
    down(): n := n-1;
    protect(limit): {
        up():
            if(n = limit ,
                error(" overflow " ),
                .up());
    }
}

```

```

    down():
      if (n = -limit ,
          error(" underflow"),
          .down());
      capture() };
  capture()
}

```

```
p: c.protect(2)
```

The last line in this example creates a 'protected' `counter` that limits its state to the interval $[-2,2]$, and binds it to `p`. Note that in the overridden `up` and `down` messages there are message sends that do not specify a target object. These message sends are parsed as super sends, and will respectively invoke `up` and `down` in the 'standard' `counter`-object.

It can easily be seen that methods in `Pic%` are actually regular Pico-functions. The object containing these methods is simply the scope in which they exist. This means that `Pic%` has first-class methods, which is very uncommon for programming languages.

Since it could become cumbersome to apply the same mixin over and over again in order to attain multiple versions of an object, cloning functionality was added. Cloning an object returns an exact replica of the cloned object. However, by introducing cloning, the model that has proven to be so simple up till now runs into some difficulties. We will illustrate this using an example.

Problem First of all, recall that Pico is lexically scoped. This means that all functions (and therefore all methods) keep a pointer to the environment they were created in (for methods this is the object containing them). Now suppose we clone `c`, the `counter`-object we created before, and call the resulting clone `c1`. Since we want our methods to be shared among several clones, `c1` needs to be a shallow copy of `c`. This gives us 2 objects that share their methods, each one having a variable representing its current state. However, because of the lexical scoping, for instance all `up` methods in all clones of `c` carry the same environment (their original lexical environment) with them. Even though each clone has its own variable representing state, when executing the `up` function in any clone, the free variable `n` in the method body is looked up and changed in the scope of the original object, which is obviously not what we want to achieve. [16]

One solution would be to duplicate each method and adjust the environment link for every method in each clone. This would however nullify sharing, something we don't want to happen either.

Solution This problem can be solved in a very elegant way, though. The main cause of the problem is Pico's lexical scope. By abandoning this lexical scope, and using its alternative, dynamic scoping, the problem is solved. Dynamic scoping means that free variables are looked up in the current (dynamic) environment rather than the environment saved at definition-time. This means that a function (or method) no longer keeps track of the environment it was defined in.

If we go back to the example with `c` and `c1`, we can now see that when `up` gets executed in the `c1` object, `n` will be looked up in the environment of execution (which is simply `c1`), and will point to the `n` that represents the correct state [16]. Changing the scoping of `Pic%` is a simple and clean solution that allows us to share methods, and it also allows us to do variable overriding. This is possible because no identifier is statically bound, and all lookup happens at runtime. It is this deferral to runtime that gives us a disadvantage: method lookup is less efficient in a dynamically scoped language. [16]

Now that methods are no longer attached to their scope, we can implement cloning with shared methods. However, if we use a shallow copy of our environment, not only methods, but also state will be shared among objects, which is far from desirable. Another problem is that shared methods should never be changed, because these changes would affect all objects. To solve these problems, the environment model needs to be altered, in order to support immutable variables, or constants. These constants can then easily be shared between multiple objects, whereas normal variables will be unique for every object. When considering the implementation of cloning this means that constants will be copied in a shallow way, where normal variables will be copied deeply. [16]

To support these constants, the environment will now have to keep track of two different lists, one for variables, and one for constants. This is shown in figure 2.3. Constants can be added to an environment using declaration (as

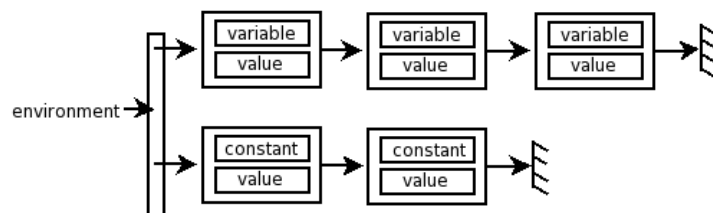


Figure 2.3: `Pic%`'s environment model [16]

opposed to definition for variables). The main difference between these two syntactically is that for declaration we use double colons (`::`) where we use

a single colon (`:`) for definition, as shown below.

Definition of a variable:

```
x: 3
```

Declaration of a constant:

```
pi:: 3.14
```

Note that these constants are shared among objects regardless of whether they are bound to a method or a value. A visual representation of this constant-sharing is shown in figure 2.4

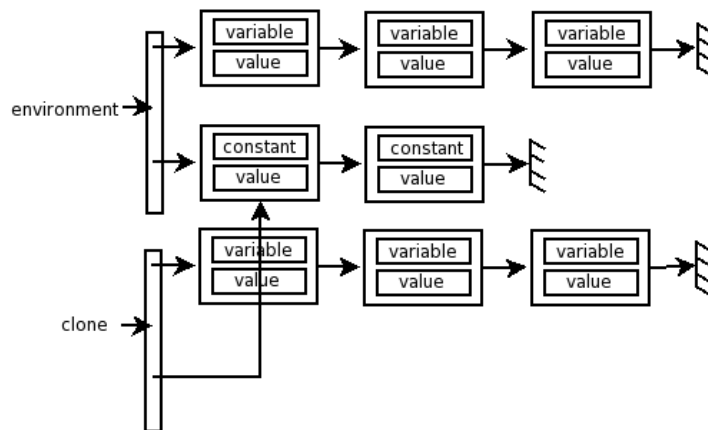


Figure 2.4: Shared constants [16]

The visibility of object-members is also arranged by declarations and definitions. Basically, all constants are always accessible by external objects, while all variables are always invisible. The rationale behind this is that constants usually describe behaviour and should therefore be accessible, while the variables usually constitute the state of the object, which should be encapsulated. Message lookup therefore implies a lookup in the constant part of the environment.

2.4.2.3 Closures

Though using dynamic scope to solve the cloning problem was a clean solution, it introduces a new problem. Consider the following piece of code:

```
method():: ...
```

```

...
variable: method;
...
variable();
...

```

First of all a method, conveniently named *method*, is declared. A bit further in the code, this method is assigned to a variable. Note that *method* does not get executed, but is simply captured and bound outside of its context. Later on we invoke *variable*. Since `Pic%` is now dynamically scoped, free variables that are being used in *method* (like e.g. the *n* variable was used in the *up* method in our counter example), are looked up in the scope of execution. It is very much possible, however, that some of those free variables don't exist in this scope, which will result in errors.

This problem is solved in a conservative way. When retrieved from an environment, a function is implicitly converted to a closure, which is a pair (`fun`, `env`) such that upon invocation of `fun`, it is applied in an extension of its enclosed environment, by including the environment. Hence, the free variables inside the function can still be retrieved.

Without too much major changes, a simple and expressive prototype-based language has been constructed from Pico.

2.5 Conclusion

In this chapter we've talked about prototype-based languages. We've explained how the prototype-based world is modelled, and discussed the differences between prototype-based languages and class-based languages.

The most important aspect we've seen is that prototype-based languages are more dynamic than their class-based siblings. There is no static class-structure, instead objects are created and cloned at runtime. It is this lack of a static structure that makes it more difficult to reason about this code, which is one of the main problems this thesis attempts to tackle.

We've seen two examples of prototype-based languages, namely `Self`, probably the most evolved prototype-based language to date, and `Pic%`, the language which will be used throughout this dissertation.

Chapter 3

Aspect-Oriented Software Development

Aspect-Oriented Software Development, or AOSD in short, is an area of active research. In this chapter the problems solved by AOSD will be explained. The principle of separation of concerns will be briefly introduced, and it will be shown how crosscutting concerns make it difficult to follow this principle.

Some approaches to deal with the crosscutting problem will be explored, one of which is Aspect-Oriented Programming. In this approach the problem is solved by explicitly stating how a concern cross-cuts a program. We will take a look at some languages used to describe these crosscuts. Later on in this document the use of a logic programming language to do this will be introduced.

At the end of this chapter aspect-oriented programming will be related to prototype-based languages, which were explained in the previous chapter.

3.1 Crosscutting Concerns

When creating a large software system, the development process usually starts by creating a software design. During this design phase the software is decomposed in small, easily comprehensible pieces. In order to have good, maintainable and evolvable software, it is necessary to have a good design. Such good design can be recognized by the degree of modularity in the decomposition.

Modularity means that a certain problem, or concern, in the software, is only dealt with in a limited amount of locations in the code. In object-

oriented languages the lowest level of modularity are classes and objects. A certain problem should thus be dealt with in a single class (or a number of classes linked through inheritance) [18]. To illustrate this an example taken from the AspectJ.org website [3] will be used. This example describes the modularity in the `org.apache.tomcat` package in the implementation of the Apache webserver. In the following images, red lines show the relevant lines of code. As can be seen in figure 3.1, all the code concerning XML parsing in `org.apache.tomcat` fits in one box (each box represents a class). In figure 3.2, the code (handling URL pattern matching) is spread over 2 classes (which are linked through inheritance). These examples illustrate good modularity.

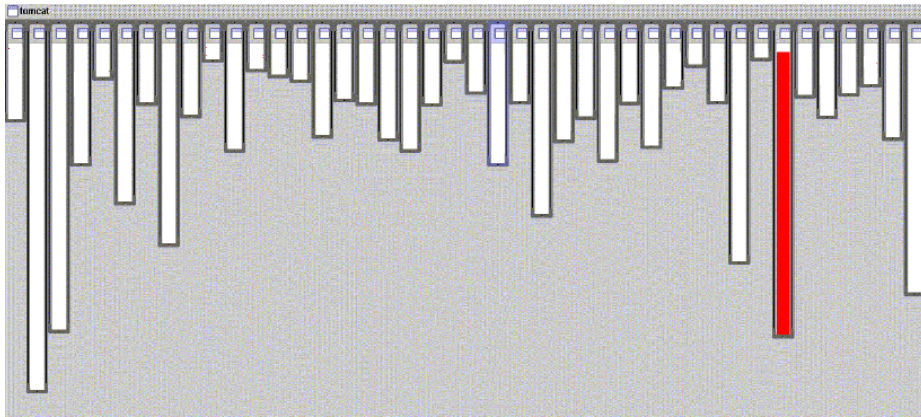


Figure 3.1: XML Parsing in `org.apache.tomcat`

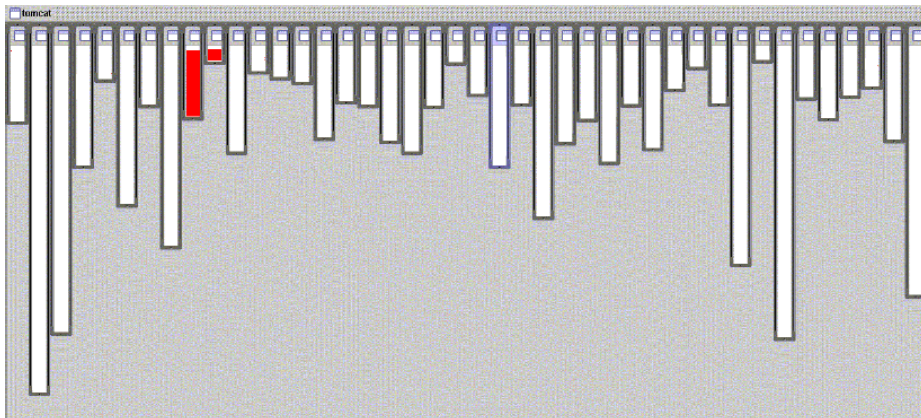


Figure 3.2: URL Pattern Matching in `org.apache.tomcat`

In figure 3.3, however the code that handles logging is shown. The modularity is terrible, it is scattered over almost all classes.

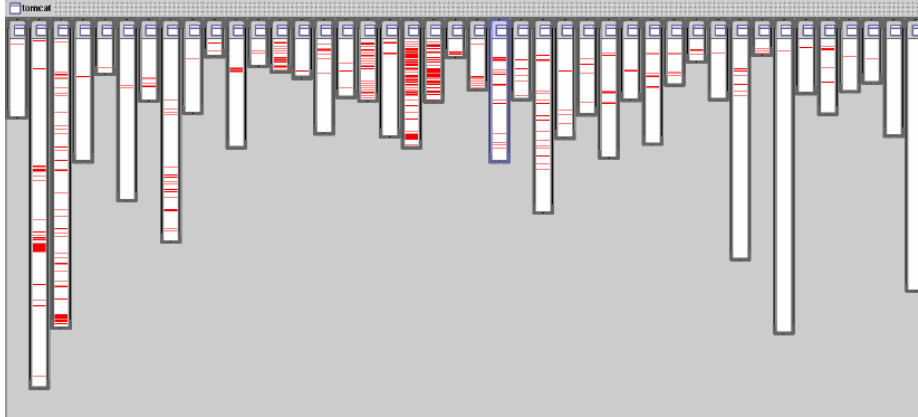


Figure 3.3: Logging in org.apache.tomcat

Given such examples of both good and bad modularity, we discuss how this impacts the maintainability and evolvability of a program. Bad modularity leads to code *scattering* and code *tangling* [18]. Code scattering means that code addressing a certain concern is scattered all over the program (as was shown in figure 3.3). Code tangling on the other hand, means that code at a specific location addresses multiple concerns at once. These two phenomena introduce a number of problems:

- Redundant code: similar code fragments return in many different places (e.g. calls to a logging function that are scattered all over the code)
- Obfuscated code: because one is doing many things at the same time it is difficult to see what the functionality of the code is.
- Unmaintainable code: code becomes hard to change, as changes have to be made consistently throughout a lot of different places.

Good modularity on the other hand has some benefits. It separates concerns, so that the implementation (and adaptation) of a concern can be treated as a relatively separate entity. It also ensures that all the code addressing a given concern appears in a single part of the program.

Now that the importance of good modularity has been established, a way to decompose our software system into a good, modular design should be given. In an object-oriented language, the lowest level of modularity is

achieved via classes and objects. Some concerns however, are hard to model at this degree of modularization: they are spread over multiple classes (e.g. the logging concern in the example above). Such concerns are called *crosscutting concerns* [3]. Though a good design may solve this problem for small software systems, it is said that *in any sufficiently complex system, there will inherently be some crosscutting concerns* [3]. This means that a programmer is forced to use a decomposition that does not separate all concerns, and needs to choose the decomposition that is best for the current problem, although this decomposition may not be suitable for other uses. This is known as the *tyranny of the dominant decomposition* [18]. This implies that it is impossible to find a fully modular design for such a system.

3.2 Solutions to the crosscutting problem

The problem of crosscutting concerns obviously calls for a solution, and in this section some approaches that have been advocated to solve it will be looked at. First we will see how it was dealt with without any changes in the implementation language. Then we will take a look at solutions that extend object-oriented languages with new concepts to handle crosscutting concerns in a modular way.

3.2.1 Inheritance

A first attempt at solving the crosscutting problem is by simply using inheritance. Consider for example a `Point` class [18] with a `move`-method. If this method needs to be logged, a subclass `LoggedPoint` can be created that does the logging first and then does a super call to perform the actual method. This way the logging concern is successfully separated from the moving concern. However, if methods of other classes also need to be logged, subclasses for all these classes need to be created too, and more logging-code needs to be added there, which leads to code duplication. Code duplication is something that should be avoided.

In addition, imagine that except for logging code, also synchronization code has to be added. The same principle can now be used: a subclass of the `LoggedPoint`-class is made containing the synchronization code with a super call. However, if the logging-code is to be disabled, this can be solved in two ways. Either the `LoggedPoint` class is modified so that it just forwards methods without doing the actual logging, which is obviously not a very clean design, or a new subclass of `Point` is made that contains only synchronization code. Though this is still manageable for a single class and two crosscutting concerns, it is difficult to imagine that this would be easy to use in a large software system with hundreds of classes and crosscutting

concerns. Hence subclassing only solves crosscutting concerns in an ad hoc fashion.

3.2.2 Aspects

Since the object-oriented paradigm obviously lacks a way of dealing with these crosscutting concerns in a clean way, it is necessary to add a new concept to OO, a concept that can handle crosscutting concerns in a modular way. This concept is usually called an aspect [27]. Aspects are a new language element, with which developers can encapsulate concerns that cut across classes.

Over the years, different approaches to using and implementing aspects have been proposed, among which *Composition Filters* [5], *Aspect-Oriented Programming* [27] and *HyperJ*[25]. The two first ones will be discussed here, since they are the most interesting for this dissertation.

First this section will be ended with a brief overview of the composition filters approach. Then an entire section will be devoted to aspect-oriented programming, since this is the most common and well-known approach to aspects at this time.

3.2.3 Composition Filters

One approach to aspects is composition filters [5]. In this approach, object-oriented programming is extended with the notion of filters. Composition filters make use of the fact that all behavior in object-oriented programs is implemented by sending messages, arguing that the manipulation of these messages can express a large category of behavior changes. In order to change this behavior, a layer called the interface part is introduced. This is shown in figure 3.4. Input and output filters are the two most important elements in the CF-model. Every single one of these filters manipulates messages in a certain way. Different filter types are available for different kinds of manipulations, for instance altering a message, rejecting a message, throwing an error, buffering a message, etc. Though the predefined filters should suffice for most purposes, it is possible to create and use new filters. Together, these filters compose the object behavior, possibly using other objects. These other objects can be either internal or external. Internal objects reside fully within the composition filter object, external objects on the other hand exist outside of the composition filter object (e.g. globals). The behavior of the object is a composition of the behavior of its internal and external objects. Additionally, it is possible that a part of the behavior is implemented by the 'inner part', which is therefore also dubbed implementation part.

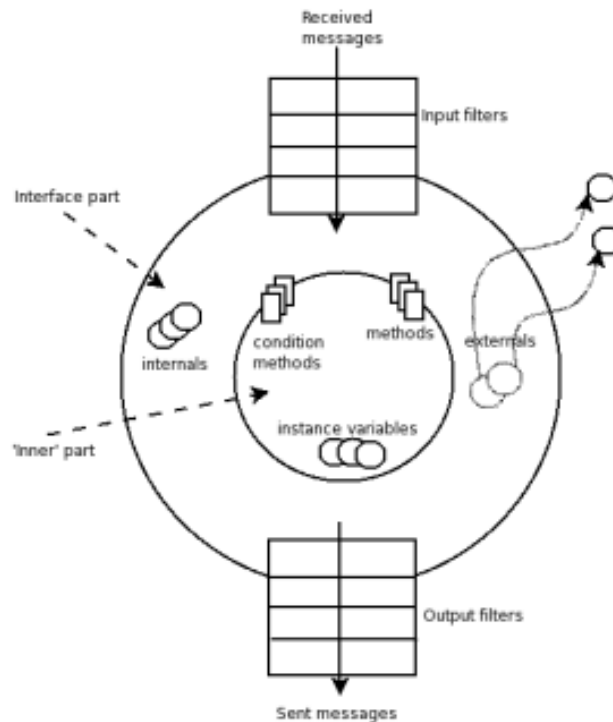


Figure 3.4: Graphical representation of the Composition Filters model [5]

Message filtering

In order to decently explain the message filtering process, we will use the example shown in figure 3.5. This figure visualizes how incoming messages are processed through three filters. Each of these filters can choose to either accept or reject a message. The semantics that are associated with accepting and rejecting messages depend on the type of the filter.

An object can receive many different messages, as shown at the top of the picture using different shapes. Every message the object receives is subject to being changed by all filters. A filter will start by matching messages based on a certain pattern. During this matching process we can simply check for message properties, such as name or number of arguments, but it is also possible to use the current state of the object.

Going back to the figure, we follow message m as it works its way through the filters. The first filter contains no patterns that match with m , so the message is rejected by this filter. In this filter rejection means that the message will just be passed unaltered to the next filter.

In filter 2, there is a pattern that matches with our method, so this filter will accept m . Filter 2 will now manipulate the message. This results in a

new message, m' , which is passed on to the next filter.

In the third and last filter there is a pattern matching m' , so again our message is accepted. This filter dispatches our message to, for instance, a local method of the object. In a set of filters, the last one is generally a dispatch filter, since there is no other way to trigger a method.

Note that this example only handles incoming messages. Outgoing messages are handled in the exact same way, with the outgoing message passing through the filters.

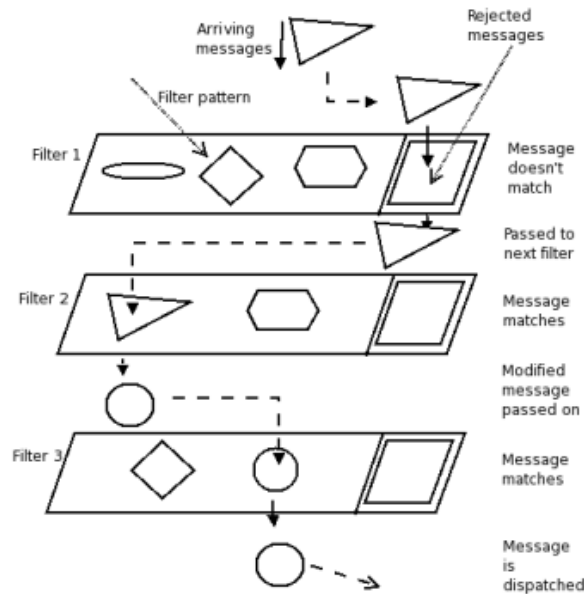


Figure 3.5: Message filtering graphically represented [4]

Superimposition

Though this filter model can be used to implement aspects, it still suffers from some problems. Suppose we have two classes, `Point` and `Line`, each containing a `move`-method, and we want to log all calls to both methods. In the current model as explained so far this would mean we have to create two filters, one for the `Point` class and one for the `Line` class, even though both filters do pretty much the same.

In order to solve this problem installing filters on classes is decoupled from declaring filters. The declarations are done separately, and instances of these declarations are then installed on top of classes. This implies that one filter can be installed on multiple classes.

However, since the declaration is separated a means is needed to install

certain filters on top of certain classes. This is done using *superimposition* [4]. Superimposition specifies which filterinterfaces are installed (or *superimposed*) upon a class.

Summary

Composition filters is one possible model that extends a language in order to support aspects. Composition filters allow expressing crosscutting concerns. In the original model it was only possible to express crosscutting concerns in a single class, but the superimposition construct also allows us to express concerns crosscutting several classes.

Before superimposition was added to composition filters, there was no support for crosscutting concerns. Filters always related to a single class of objects. This approach became much more powerful when the superimposition construct was added, however by adding this construct the biggest distinction between composition filters and aspect-oriented programming has been removed, as will be shown in the next section.

3.3 Aspect-Oriented Programming

The most commonly used approach to aspects is aspect-oriented programming [27], or AOP. AOP was originally designed for class-based languages, and until now it is still almost solely based on them. It is a collection of approaches that are founded on some common ideas and terminology. Later on in this section two such approaches will be discussed, but first the common ideas and terminology are defined.

All AOP approaches are built around the idea of a weaver. It is this weaver that is responsible for combining the functionality of the aspects with that of the base program. This is represented graphically in figure 3.6. In order to allow the weaver to combine an aspect's functionality with the base program, an aspect is required to describe two things: it has to implement the functionality it adds to the base program, and it has to specify the locations in the code where it adds this functionality to. We define the first as the advice of the aspect, and the latter as the hook or joinpoint where the advice should be added.

As said above, the weaver's job is to combine the functionality of aspects with that of the base program. This process is generally referred to as weaving. There are several different approaches to weaving [33]:

Compile-time weaving Compile-time weaving means that aspects are physically woven into the base program at compile-time using bytecode or sourcecode transformations.

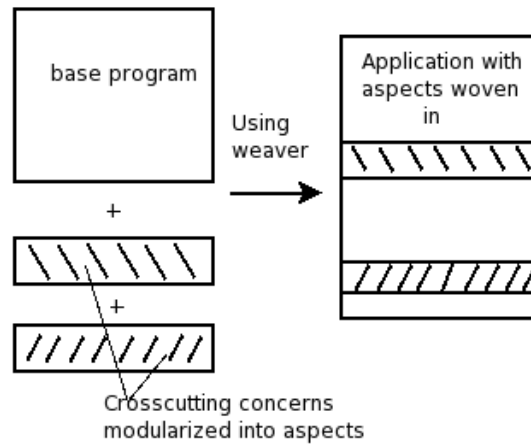


Figure 3.6: Graphical representation of the weaving process [18]

Load-time weaving A load-time weaver weaves the aspects at the moment the application is loaded, for instance by making use of a specialized class-loader that allows inserting aspect-code.

Pseudo run-time weaving Pseudo run-time weaving means that the application itself monitors for aspect execution (using for instance a post-processor), giving us the possibility to attach and detach aspects at run-time, though this comes with a considerable run-time overhead.

Genuine run-time weaving In Genuine run-time weaving, finally, aspects are dynamically woven into the base application at run-time of the application. This happens through a modified, aspect-enabled virtual machine, or through a special runtime weaver that is attached to the virtual machine. This weaving strategy also allows attaching and detaching aspects at run-time, and reduces most of the run-time overhead that came with pseudo run-time weaving.

To properly allow the aspects to specify where they need to be woven in the base program, a so-called *pointcut language* is used to describe joinpoints. A pointcut expression is a description of one or more joinpoints written in the pointcut language. It is the pointcut language that makes AOP powerful. The more expressive and powerful the pointcut language, the easier it becomes to describe joinpoints.

3.3.1 AspectJ

The first example of an aspect-oriented language that will be discussed is AspectJ. AspectJ [2] is a simple and practical extension to Java that adds aspect-oriented programming. This discussion is largely based on Kris Gybels' dissertation [22] and the AspectJ website [3].

AspectJ uses compile-time weaving. Since all the weaving has already been done before the program even starts running, there is almost no runtime overhead, making this the fastest way of weaving. This does however mean that a lot of dynamism is given up. It is for instance not possible to add and/or remove aspects at runtime using this model.

Joinpoint model

AspectJ's joinpoint model is based on two graphs, the execution graph and the class graph. The first one contains nodes for objects receiving and sending messages, and for accessing and updating an object's state. The links between the nodes are based on the order in which the program executes them. The class graph is a representation of all classes in the system, where links represent the inheritance relationship between them. These two graphs are used for respectively dynamic and static crosscutting.

Dynamic joinpoints

To explain dynamic joinpoints an example, taken from [22], will be used. We will start by giving some example code describing a `Point` and a `Line` class, shown in listing 3.1.

Listing 3.1: Example code for Dynamic Joinpoints

```
class Point {
    private int x,y;

    /* ... */

    public void slide(int dx, int dy) {
        setX(getX() + dx);
        setY(getY() + dy);
    }
}

class Line {
    private Point a,b;
```

```

/* ... */

public void slide(int dx, int dy) {
    a.slide(dx, dy);
    b.slide(dx, dy);
}
}

```

The depiction of the execution graph that results from executing `ln1.slide(3,6)` is shown in figure 3.7.

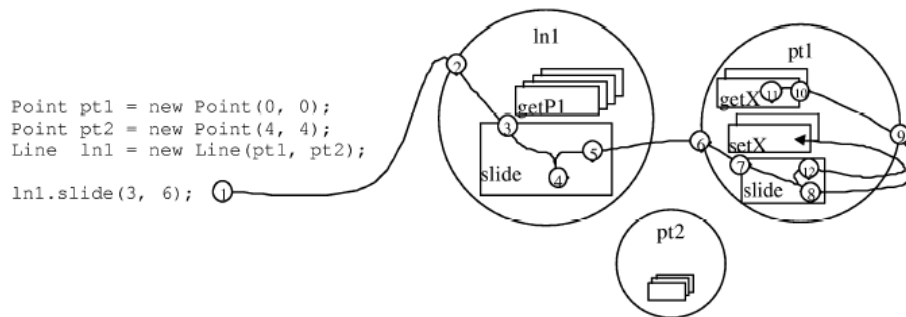


Figure 3.7: Informal depiction of dynamic joinpoints

In the example three objects are being made, two `Points` and a `Line` connecting them. The objects are shown as circles, containing rectangles representing their methods. The execution graph is drawn on top of these objects, so as to illustrate how it crosscuts the objects. The graph is filled with joinpoints, depicted by small circles. All of these joinpoints have been numbered, an explanation for each joinpoint is given below:

1. A message send joinpoint at which the message `slide` is sent to the object `ln1`
2. A message reception joinpoint at which the message `slide` is received by the object `ln1`
3. A method execution joinpoint at which the method matching the `slide` message, with the number and type of arguments, is executed
4. A state access joinpoint where the field `a` of object `ln1` is referenced. The value in the field is the object `pt1`
5. A message send joinpoint at which the message `slide` is sent to the object `pt1`

6. Similar to 2
7. Similar to 3
8. A message send joinpoint where the message `getX` is sent to the object `pt1`
9. Similar to 2
10. Similar to 3
11. A state access joinpoint where the field `x` of object `pt1` is read. After this point, control returns back through joinpoints 11,10 and 9 to 8
12. A message send joinpoint where the message `setX` is sent to object `pt1`
13. and so on until control passes back through joinpoint 1

Pointcuts on dynamic joinpoints

The pointcut language for dynamic joinpoints in AspectJ is used to describe joinpoints by stating the conditions they must meet. To express these conditions AspectJ offers a set of primitive conditions, which can be merged using standard boolean operations. Some examples of these primitives are shown below.

- **call(void Point.slide(int,int))** will match message send joinpoints where the message `slide` is sent to an instance of `Point`, with two integer arguments.
- **execution(void Point.*(*))** will match executions of any method specified by the `Point` class with a single argument.
- **cflow(void Point.slide(int,int))** will match with any joinpoint that occurs after a `slide` message is sent to an object of type `Point`, but before the object is done handling this message.

Note that in the second example wildcards were used. Asterisks match with anything (class names, method names, etc). AspectJ also has a double-dot wildcard (`..`) that can be used for argument lists and will match with any amount of arguments. Should we have used “`Point.*(..)`” in the second example, it would not just match methods with 1 argument, but also methods with multiple arguments (like for instance the `slide` method of the first example). In the last example a different kind of primitive is used. Instead of specifying conditions on the type of the joinpoints, it specifies a condition

on the origin of the joinpoint.

As mentioned above these primitives can be combined using boolean operators. A name can be assigned to new pointcuts through a pointcut declaration:

Listing 3.2: Example of a pointcut

```
pointcut sliding() :
    cflow(void Point.slide(int,int))
    && execution(void Point.set*(int));
```

This example will match with any set-method on a Point-object that is called from within a `slide`-method.

Advice

Advice in AspectJ aspects is a piece of regular Java code woven into a base-level program. These advices can be inserted at different locations in the execution graph, namely before, after or around the joinpoint. Before and after advices are respectively inserted before and after the code of the joinpoint. Around advices are more complicated: a part of the advice is executed, after which control goes to the joinpoint-code. After the joinpoint code finishes the control returns to the after advice, where the remainder of this advice is executed. Around advices can also be used to replace the original code of the joinpoint. A small example of an around-advice is shown below.

Listing 3.3: Example of advice

```
after() : sliding() {
    System.out.println("Executing set after slide");
    proceed();
    System.out.println("Done executing");
}
```

Aspects

In the beginning of this section it was said that an aspect needs to describe two things, namely the code that should be executed, and the place where this code should be executed. Now that we have seen examples of pointcuts and advice, we can combine this and create an aspect with it.

Listing 3.4: Example of an aspect

```

aspect SlideLogger {
    pointcut sliding() :
        cflow(void Point.slide(int , int)) &&
        execution(void Point.set*(int));

    after() : sliding() {
        System.out.println("Executing set after slide");
        proceed();
        System.out.println("Done executing");
    }
}

```

As can be seen in this example, an aspect contains a pointcut description (the upper part of the code sample), and it also specifies advice (the lower part of the code sample). This aspect will log all calls to `set*` when it is executed from within a `Point`'s `slide` method, by printing a string before and after the call.

3.3.2 JAsCo

Another aspect-oriented language, which is also an extension to Java, is JAsCo [34]. JAsCo was designed to deal with some of the restrictions that are imposed on existing approaches:

- Aspect deployment is rather static in current approaches. For instance in AspectJ aspects lose their identities when they are integrated in the base application. This makes it difficult to retract the aspect and replace it by a totally different one, a feature that can be quite necessary in dynamic environments. For instance when employing business rules [21] through aspect-oriented programming it would be rather difficult to recompile an entire application every time a single rule changes.
- In most AOSD approaches aspects are described with a specific context in mind (advice and pointcut need to be specified together), making it impossible to reuse aspects.

JAsCo stays as close as possible to regular Java syntax, but introduces two new concepts, aspect beans and connectors. Aspect beans are regular Java beans that declare one or more related hooks, as a special kind of inner classes. Hooks are entities that can be seen as the combination of AspectJ's advice and pointcut. These aspect beans can be generic, i.e. reusable in different contexts. To bind a hook to a specific context, connectors are used. [32]

Another contribution of JAsCo is the fact that it supports 'hot-pluggable'

aspects. Aspects can be added and removed during runtime, which is very useful in dynamic programs (as opposed to AspectJ, changes to an aspect require recompiling the entire program).

A closer look will be taken at JAsCo's syntax using two examples. First an aspect bean will be shown, after which we will take a look at a connector.

Listing 3.5: JAsCo aspect bean

```

1 package aspects;
2 class SlideLogger {
3
4     hook logger {
5
6         logger(method(.. args),method2(.. args)) {
7             cflow(method) && execution(method2);
8         }
9
10        after() {
11            System.out.println("Executing set after slide");
12            proceed();
13            System.out.println("Done executing");
14        }
15    }
16 }

```

As said above, an aspect bean contains a hook that can be seen as the combination of advice and pointcut from AspectJ. The hook definition is similar to that of inner classes, as can be seen in the example on lines 4 till 13. Inside this hook two methods are defined. The first one (line 6 till 8) is a constructor method, which serves as the pointcut. It describes when the aspect is executed. As can be seen, the `cflow` and `execution` conditions are also available in JAsCo. The second method (line 10 till 14) is the advice part of the hook. As the name of the method implies it is an around-advice. Like in AspectJ, it is also possible to add after and before-advice. This aspect bean behaves in the same way the example shown in section 3.3.1 does, except that there is no context provided for this aspect bean. This is done using connectors, as follows:

Listing 3.6: JAsCo connector

```

1 static connector conn {
2     aspects.SlideLogger.Logger hook0 = new
3     aspects.SlideLogger.Logger(void Point.slide(int,int),
4                               void Point.set*(int));
5 }

```

A connector simply binds aspect beans to a certain context, as can readily be seen in the example. The connector simply instantiates the hook, and adds context to it in the form of two method patterns.

3.4 Aspects and prototype-based languages

Now that some approaches to aspect-oriented software development have been discussed, we will relate this to prototype-based languages, which were described in the previous chapter. This is done because our goal in this dissertation is to add aspect-oriented programming to these prototype-based languages.

The main problem with the approaches described above is that they are based on a class-based object-oriented language, making them inapplicable to prototype-based languages like the ones discussed in chapter 2. This problem is discussed by Cleenewerck, Gybels and Peeters [8], and will be briefly summarized here.

A first approach to implement aspects is to use delegation. The argument for this is that 'putting code' before and after a method is handled readily by delegation. Consider for example a synchronized version of a point object that delegates to an actual point object (as is depicted in figure 3.8). The `move` and `get` methods of the synchronized point perform synchronization, and then delegate to their counterparts in the parent object.

It is however fairly obvious that this approach suffers from roughly the

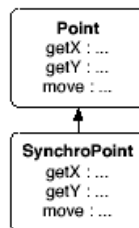


Figure 3.8: Point Synchronization example with delegation [8]

same problems as aspects through inheritance in OO systems (see section 3.2.1). It is for example difficult to reuse the synchronization code for other objects than points. Another problem is that a new method can be added to the point object at runtime. At this point an appropriate method should be added in the synchronized object as well.

However, the real problem is not the advice code that needs to be executed, it is the pointcut description of the location *where* this advice needs to be executed that poses the biggest problem. In class-based languages pointcuts can be described using the class structure, or using method signatures. In prototype-based languages there are no classes however, instead there are objects delegating to parent objects. In addition, most prototype-based languages are dynamically typed, making method signatures rather pointless. It is concluded that due to the dynamic nature of prototype-based lan-

guages, the pointcut languages we have seen so far are not powerful enough to fully support aspects in a prototype-based language. When designing a pointcut language that is powerful enough, it can be beneficial to choose a language that is very close to the implementation language. The higher the degree of interaction between the pointcut language and the implementation language, the easier it becomes to cope with the dynamic nature of the implementation language. When the pointcut language can interact with the base language it can for instance be possible to check delegation links in the base language itself, and use this information through interaction.

3.5 Conclusion

In this chapter a general overview of AOSD was given. The core problem AOSD addresses was briefly introduced, and some approaches to AOSD, including Composition Filters and AOP, were shown. At the end of this chapter AOSD was related to the context of prototype-based languages, where it was concluded that existing pointcut languages aren't powerful enough to fully support aspects in prototype-based languages, and that, in order to make full use of the dynamic nature of these prototype-based languages, it would be interesting to have a high degree of interaction between the pointcut language and the actual implementation language.

Chapter 4

Language Symbiosis

In this chapter a brief look will be taken at language symbiosis. We start by explaining what exactly constitutes language symbiosis, and why it can be interesting for two languages to interoperate symbiotically. The distinction between symbiosis and integration will be shown, after which this chapter will be concluded by taking look at some examples of language symbiosis. In the next chapter we will then elaborate on the language symbionts that will subsequently be used to explore aspect-oriented logic meta programming in.

4.1 Symbiosis

In this section we discuss what constitutes symbiosis, how it relates to multiparadigm integration efforts and conclude with some examples where language symbiosis can be useful.

4.1.1 Definition

In biology, symbiosis is defined as “A close, prolonged association between two or more different organisms of different species that may, but does not necessarily, benefit each member” [17]. Mutualism is then a symbiotic relationship in which each species benefits.

Outside the field of biology, symbiosis is usually restricted to mutualism [35], meaning that symbiosis can be defined as “A relationship of mutual benefit or dependence”. Language symbiosis is then simply a relationship between two programming languages, from which both programming languages can benefit through the ability to share one another’s concepts in a transparent way.

In practice, this means that two languages can use each others functionality,

for instance an object-oriented language executing a query in a declarative language it's symbiotically related to.

4.1.2 Language Symbiosis vs. Integration

Another way to allow concepts from two languages to be used in a single program is the creation of a new multi-paradigm language that combines the strength of two other languages. This approach is called integration. There is a big difference between integration and language symbiosis, as can be seen in figure 4.1. Where language integration merges the features of multiple languages into a single new language, language symbiosis only allows these languages to use each others features. In language symbiosis, no new language is created, and each language holds on to its own identity.

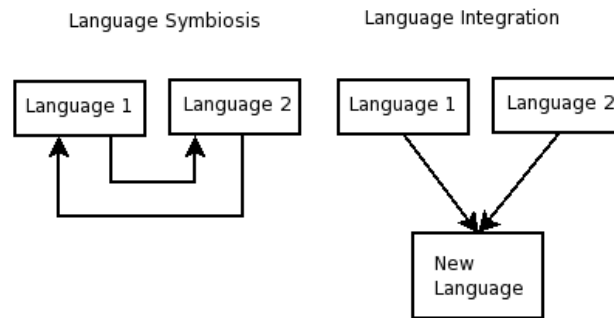


Figure 4.1: Language Symbiosis vs. Language Integration

4.1.3 Use of symbiosis

Some parts of a program may be more easily expressed in another language or paradigm. This is exactly what symbiosis allows. It offers the programmer additional expressiveness, since the latter is able to implement different parts of a program in different languages. This way it is for instance possible to represent rule-based knowledge in a declarative language, while algorithms that use this knowledge can be implemented in an object-oriented language. An implementation of such an example will be given in the next chapter, where it will be used to validate the described symbiosis.

4.2 Linguistic Symbiosis

Linguistic symbiosis is a concept that arose from work on reflectively extensible interpreters [21]. In [26] it refers to the ability of base-level objects

(objects in the interpreter itself) and meta-level objects (objects in the interpreted language) to send each other messages, even though the base-level and meta-level language are not the same. However in the early days of symbiosis, this concept was only explored in cases where both the base-level and the meta language were object-oriented languages. Later on language symbiosis was also explored for different paradigms, as with Smalltalk and SOUL [23].

In this section we will discuss some approaches to linguistic symbiosis. We will start with a classic method where buffers and pipes are used to do communication, after which we will discuss a cleaner way to achieve symbiosis.

4.2.1 Approaches to linguistic symbiosis

In order to achieve linguistic symbiosis, it is by definition necessary to combine programs written in different languages. This combination is achieved by letting one or both languages use functionality of the other. This implies that a form of communication needs to be present between both languages. In classical approaches communication is achieved through buffers or pipes (a very simple example of this is the linux pipe command that uses the output of one program as the input of another one). The programmer is then able to transmit raw data through these buffers or pipes, which can be read by the other program. Usually the data sent to the buffer is preceded by tags, to indicate what the data represents and what the other program should do with it. This typically means that only primitive data values such as strings and numbers can be passed to the other language, implying that each program needs to reconstruct compound structures. The use of tags to indicate what action needs to be taken also implies that the other program needs to interpret this tag and to translate it to something useful for the program, like a procedure or a query. It is obvious that this kind of communication is very primitive, and a much better way to transmit data is necessary.

One way to do this is by creating a symbiosis with the implementation language. Since one of the symbionts is implemented in the other one, it is rather easy to transmit data. An excellent example of how this can be achieved is the symbiosis between Agora and its implementation language [31]. Agora programs and programs in Agora's implementation language can pass objects to each other, and objects in either program can be manipulated by the other through message sends, just like when they would be written in that other language. We will come back to Agora in section 4.3.1, where we will discuss it as an example of symbiotic languages.

Another example of a symbiosis between a programming language and its implementation language is this between Smalltalk and SOUL. Since SOUL is a logic programming language, and Smalltalk an object-oriented one, sym-

biosis obviously becomes more complex. When combining object-oriented languages it is necessary to find a way to pass objects between the two languages, and to translate messages from one language into messages from the other language. In a multiparadigm symbiosis, like the one between Smalltalk and SOUL, it is necessary to map messages and objects to rules and terms. This will be shown in section 4.3.2 where we will go into more detail on the symbiosis between Smalltalk and SOUL.

Another way of allowing two languages to communicate with each other is through a common implementation language. Both approaches are represented graphically in figure 4.2.

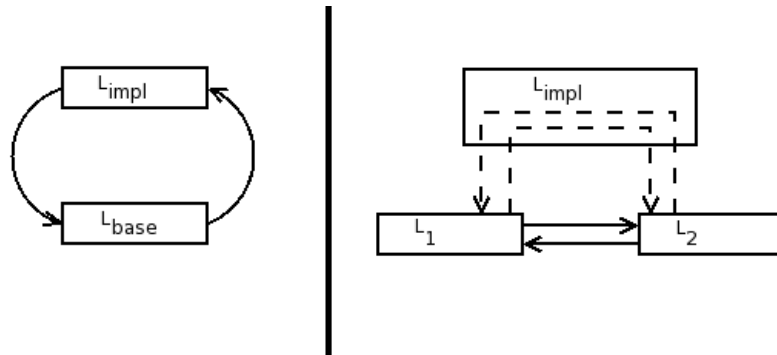


Figure 4.2: Symbiosis between a language and its implementation language (left) versus symbiosis between two languages in a common implementation language (right)

In this figure the full arrows represent the inter-language communication as it would seem to the programmer. The dashed arrows represent the actual communication. In a symbiosis between a language and its implementation language these two kinds of arrows are equal. In the other kind of symbiosis the communication is achieved through the common implementation language.

In the next chapter we will use this second kind of symbiosis for our own implementation. In section 4.3.3 we will see another example of this kind of symbiosis: lillambi.

4.2.2 Summary

In linguistic symbiosis two languages are able to use each others functionality. This is achieved through a form of communication between those languages. In classical approaches this was done by using buffers, allowing only primitive values to be passed. In more advanced approaches it

is also possible to pass objects between languages. Two such approaches were discussed. In the first one a symbiosis is constructed between a language and its implementation language. The second approach handles a symbiosis between two languages that are implemented in a third, common language. More difficulties arise if the programming languages involved adhere to different paradigms, for instance if one language is object-oriented and the other declarative. Multiparadigm symbioses require a mapping between different concepts in the involved languages, like message sends and query execution. This will be clarified in the next section.

4.3 Case studies in Language Symbiosis

To illustrate the concept of language symbiosis, this chapter takes a look at some examples of symbiotic languages. First the symbiosis between Agora and its implementation language will be explained. Since the symbiosis described in the next chapter will be between a declarative and an object-oriented language, the symbiosis between Smalltalk and Soul will also be discussed. This overview of symbioses will be concluded with a brief discussion of lillambi, a multi-lingual programming environment, since its technical setup is similar to the one we will use in the next chapter.

4.3.1 The Agora Framework

Agora [31] is a prototype-based programming language for which a symbiosis with its implementation language is constructed. Both Agora and its implementation language are object-oriented, so they are an excellent example of symbiosis between two programming languages that adhere to the same paradigm. Programs in both languages can pass objects to each other, and objects written in either language can be manipulated from within the other language as if they were written in this language (i.e. in one language messages can be sent to objects written in the other language). This section will show how Agora objects can be used as objects in the implementation language, and vice versa. This is illustrated informally in figure 4.3.

4.3.1.1 Additional terminology

Before we go into further technical detail, some of the original Agora terminology based on the work of Steyaert [31] will be introduced. Since Agora objects and objects on the implementation level are referable from within both Agora and its implementation language, the distinction between them can become rather vague. Therefore some new terminology is introduced,

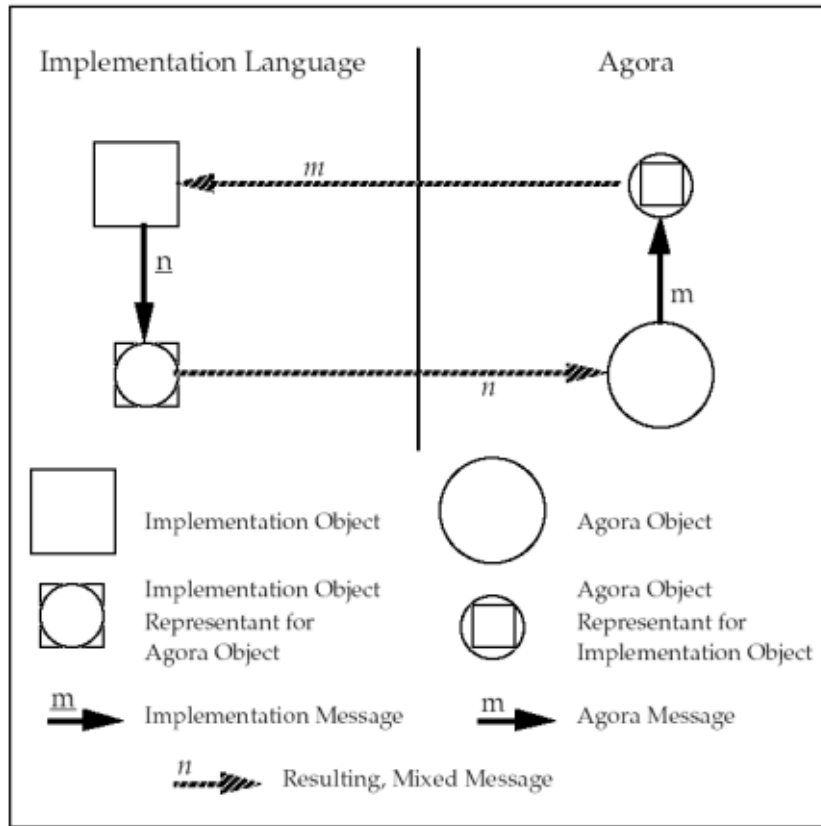


Figure 4.3: Language Symbiosis between Agora and its implementation language [31]

where a distinction is made between the language in which an object is expressed, and the language from which messages can be sent to an object. First of all a distinction is made between *implicit messages*, messages expressed in the implementation language, and *explicit messages* which are expressed in Agora.

An *explicitly encoded* object is an object which is expressed in Agora, while an *implicitly encoded* object is expressed in the implementation language. An object that can be sent implicit messages is an *implicitly referable* object, an object to which explicit messages can be sent is called *explicitly referable*. Further, *implicit* and *explicit* objects are objects which are respectively implicitly and explicitly encoded *and* referable. The new terminology is summarized in table 4.1.

	Implementation Language	Agora
referable	Implicitly Referable Object	Explicitly Referable Object
encoded	Implicitly Encoded Object	Explicitly Encoded Object
referable & encoded	Implicit Object	Explicit Object

Table 4.1: A catalog of objects in Agora's symbiosis model [31]

4.3.1.2 Meta objects

The terminology introduced in the previous section can be interpreted ambiguously for a particular kind of objects, namely meta objects [31]. Meta objects are both implicitly and explicitly referable (with different protocols). These objects represent Agora entities and can as such be sent explicit messages. They can also be sent implicit messages from within the interpreter. This can be illustrated by taking a look at the representation of explicit objects in the implementation language. All explicit objects are represented by an implicitly referable meta object. An explicit message to an explicit object is represented by an implicit message to the implicitly referable representation of that object (though the signature of the message will be different).

4.3.1.3 Constructing the symbiosis

In order to construct the symbiosis between Agora and its implementation language, two conversion methods are introduced [31]. The first one, named `asImplicit`, turns a meta-object into an implicitly referable object. The second one, `asExplicit`, turns any implicitly referable object into a meta-object. In other words, the `asImplicit` message allows Agora-objects to be referred from within its implementation language, vice versa the `asExplicit` message allows an implementation level object to be referred from within Agora. These two methods are implementation level methods, meaning they can only be sent at the implementation level.

How the `asImplicit` and `asExplicit` messages are used is illustrated below.

Consider an implicit object. The `asExplicit` method converts an implicitly encoded object into an explicitly referable object. This resulting object can receive explicit messages. Upon reception of such a message, the converted object will translate this message to a message on the implementation level. It also makes sure that arguments are translated into implicitly referable objects, and that the result is translated back into an explicitly referable object. These translations are necessary because the message was sent from within an Agora program. In the same fashion explicitly encoded objects can be converted to implicitly referable objects that can receive implicit

messages.

This makes it possible to use Agora objects from within Agora’s implementation language, and vice versa, which is exactly why a symbiosis was constructed.

4.3.2 Smalltalk and SOUL

The next example of linguistic symbiosis we will discuss is the symbiosis between the object-oriented programming language Smalltalk and the logic programming language SOUL. SOUL, or Smalltalk Open Unification Language [41], is used to reason about Smalltalk code declaratively. Moreover, SOUL is implemented in Smalltalk, meaning that this is another example of a symbiosis between a language and its implementation language. This symbiosis is hard to achieve because the paradigms these languages adhere to are fundamentally different [7]. Generally, object-oriented programs consist of objects communicating through messages. Most control flow is explicitly programmed. Logic programs consist of rules and facts, and the control flow is implicit (for readers not familiar with the logic programming we refer to section 5.1.1 in the next chapter, where the declarative programming paradigm is explained).

We will now see how Smalltalk programs can be invoked from within SOUL programs, and vice versa.

4.3.2.1 Invoking Smalltalk from within SOUL

In order to invoke Smalltalk programs from within SOUL, two special constructs, *Smalltalk term* and *Smalltalk clause* are introduced. Additionally, Smalltalk objects are treated as constants.

A Smalltalk term contains a Smalltalk expression enclosed in square brackets. Each time this term is unified with another term, the expression is evaluated, after which the resulting Smalltalk object will be used to complete unification. It is also possible to use logic variables in this expression. A Smalltalk clause resembles a Smalltalk term syntactically, but differs in that the Smalltalk expression between square brackets must evaluate to either true or false.

Smalltalk objects can be bound to logic variables by unifying a Smalltalk term with a logic variable. In SOUL objects are treated as constants. This means that they will only get unified with free logic variables and themselves.

An example of how Smalltalk expressions can be used in SOUL is shown below, in listing 4.1.

Listing 4.1: Invoking Smalltalk from within SOUL

```
class(?x) if
  nonvar(?x),
  [ ?x isClass ]
```

This example shows a SOUL rule to check whether a certain variable `?x` is a Smalltalk class. The `nonvar` predicate checks whether there is a value attached to `?x` since otherwise the Smalltalk message `isClass` is sent to a free variable.

4.3.2.2 Invoking SOUL from within Smalltalk

Smalltalk can easily start logic queries in SOUL, since SOUL is implemented in Smalltalk. This can be done by explicitly sending a message to the SOUL evaluator class, which will return an object containing a collection of all possible variable bindings, or an indication that the query has failed. An example of how Smalltalk can execute queries using SOUL is shown below in listing 4.2.

Listing 4.2: Invoking SOUL from within Smalltalk

```
argumentArray := Array with: (Array with: #x with: Class).
evaluator := SOULEvaluator eval: 'if class(?x)'
           withArgs: argumentArray.
```

In this example an array is constructed in which an array with `#x` and `Class` is inserted. Then the query is called by sending an `eval:withArgs:` message to the `SOULEvaluator` class, with the actual query and the argument-array as parameters.

4.3.2.3 Transparency

In the two above sections it was shown that it is possible to use SOUL from within Smalltalk and vice versa. However, it is not possible to do this in a transparent way. In order to make this symbiosis transparent, the syntax of SOUL was changed to resemble that of Smalltalk [23]. In this new syntax predicates look like message sends, as illustrated in listing 4.3.

Listing 4.3: Old versus new SOUL syntax[23]

```
member(?x, <?x | ?rest >).
member(?x, <?y | ?rest >) if
  member(?x, ?rest).

<?x | ?rest > contains: ?x.
<?y | ?rest > contains: ?x if
  ?rest contains: ?x.
```

The first two rules are written in old SOUL syntax. Lists are represented using `<` and `>`. Inside a list the `|` symbol is used to separate a number of elements (before `|`) from the rest of the list (after `|`). Logic variables are prefixed with a question mark.

The first set of rules clearly contrasts with the second, where the `contains:` query looks like it is sent to a list, resembling Smalltalk's syntax. The

second rule for `contains:` can be interpreted as “for all `?x`, the answer to the message `contains: ?x` to objects matching `<?y | ?rest>` is true if the answer of the object `?rest` to `contains: ?x` is true”.

Using this syntax, a Smalltalk program no longer has to send a message to the SOUL evaluator class in order to execute a query. Instead, switching between SOUL and Smalltalk now occurs as a result of method and rule lookup. Smalltalk was changed so that when a message is sent to an object that has no method implemented for it, the message is translated to a query. The opposite is true for SOUL: when a rule is not found for the predicate of a condition, the condition is translated to a message [23]. This implies that interchanging methods and rules becomes much easier and much more transparent.

Transparency is thus achieved by switching between languages when an error occurs. This approach will also be used in our experiment.

4.3.3 Lillambi

The last example of symbiosis we will describe is the lillambi multilingual environment. Lillambi [12] is an example of the second kind of symbiosis shown in section 4.2.1. It is a symbiosis between multiple programming languages that are all connected by the fact that they share a common implementation language. This resembles our own approach, which will be presented in the following chapter.

Lillambi is not restricted to some languages. Currently, lillambi supports four languages: Java, Shell, BeanShell and TuProlog. Apart from Java itself, all these languages are implemented in Java, and they communicate via the Java Virtual Machine. This is represented graphically in figure 4.4

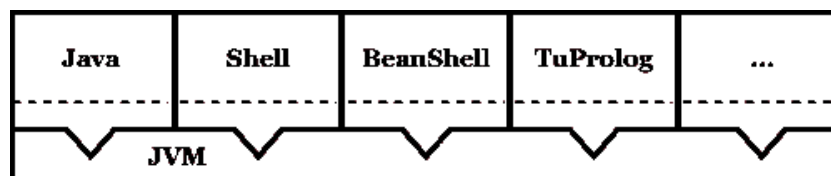


Figure 4.4: Graphical representation of the lillambi setup [12]

In lillambi each piece of code is in its own file, and in a file only one kind of code is possible. How these files need to be loaded is described in some sort of “meta” file. Each of these files maps onto an object, so each object is implemented in a single language. It is the responsibility of the programming language’s interpreter to translate these objects to Java-objects, and vice versa.

4.4 Conclusion

In this chapter language symbiosis was introduced. The chapter started by giving a definition of symbiosis, stating that two languages are symbiotically related if they possess the ability to share one another's concepts in a transparent way. The difference between language symbiosis and language integration was shown. The latter combines multiple languages in a new, multi-paradigm language. The former allows a certain language to use functionality of another language, but both languages hold on to their own identities.

The chapter then went into further detail on how symbiosis between two languages can be achieved, and how the approaches for language symbiosis have evolved over time.

Finally, this chapter described three examples of symbiosis. First, we described Agora, an object-oriented language that can interoperate with its object-oriented implementation language. Since the symbiosis we have implemented ourselves also involves a declarative language, we also highlighted the symbiosis between Smalltalk and SOUL. Because of SOUL's declarative nature this symbiosis was more complex: message sends needed to be mapped on queries, and a new syntax had to be defined for SOUL in order to make the symbiosis transparent.

The final example that was described was lillambi. Lillambi was different from the two examples mentioned above in that it is a symbiosis between multiple languages that interact through a common implementation language. Communication in lillambi is achieved through the Java Virtual Machine.

Chapter 5

Symbioco

In the previous chapter some examples of language symbiosis have been described. In this chapter Symbioco, a symbiosis between Pic% and Loco which was implemented to serve as an experimental platform for this thesis, will be discussed.

Pic% was already discussed in detail in chapter 2, so we will start this chapter with a brief introduction to Loco, a declarative programming language. After this introduction we will discuss Symbioco in full detail, to conclude this chapter with an example in which the symbiotic facilities of Symbioco will be heavily used.

5.1 Loco

As mentioned above, Loco is a declarative logic programming language. It was developed by Prof. Dr. Theo D'Hondt and was originally written in Pico, meant for educational use. Syntactically Loco looks a lot like Pico and Pic%, which will later on prove to be very beneficial for this symbiosis. We will now briefly introduce the declarative programming paradigm, and then show how this applies to Loco.

5.1.1 The declarative programming paradigm

The declarative programming paradigm, also known as logic programming, was developed in the 70's. Instead of regarding a program as a step-by-step description of an algorithm, it is built by a number of logical facts and rules. A procedure call can be seen as a theorem of which the truth must be proved. Unlike traditional, imperative programming languages, where a program itself is a list of steps that need to be taken to solve a problem (a program specifies *how* a problem should be solved), a logic program rather

specifies *what* the problem is. [19]

Theorems (or procedure calls) are proven by a logic programming language using knowledge. To illustrate this, consider proving 4 is greater than 3. This is only possible if we know what 'greater than' implies, so we need knowledge about 'greater than' before we are able to prove this statement. This knowledge should be specified in a rule base which can be queried. Therefore procedure calls are named *queries* in logic programming.

Upon query execution, the system will search its knowledge for all possible information it has about the query. This search will return a number of expressions that can be divided into two different groups, facts and rules.

Fact A fact is an unconditional truth, for example '4 is greater than 3'. Suppose we ask the system whether 4 is greater than 3, it can immediately see that this is indeed so. However, if we ask whether 5 is greater than 3, the system can't prove it. As can be seen this would mean that we need to insert an infinite amount of facts to do even simple number comparison. Therefore we need another kind of knowledge, rules.

Rule A rule is a conditional truth, for example 'a number x is greater than a number y if $x - y$ is positive'. A rule is only true if all of its premises are true (only if $x - y$ is positive, we can conclude that x is greater than y). With this rule in our knowledge we can compare all numbers we wish (provided that we know how to subtract and how to see whether a number is positive).

As can be seen in the previous example, we often use variables in rules. In an imperative language, a variable is a name for a memory location, containing data of certain types. This data may vary over time, but the variable always points to the same location, and is well-defined at every moment. In a declarative programming language however, a variable is a variable in the mathematical sense, a placeholder that can take on any value, but once a value is assigned to it, it is immutable. This implies that if we execute a query with a variable in it, the system will try to find a value for this variable so that it can prove the query. For example if we ask whether x is greater than 3, the system will try to find a value making this query true (e.g. $x=4$). The process of finding values for these variables is called unification.

When the query 'x is greater than 3' is executed, the system will first look up what it knows about 'greater than'. Suppose the system finds the fact '4 is greater than 3' in its knowledge, then it will unify the terms in the query (x and 3) with the terms in the fact (4 and 3). Unifying a value with

another value (3 and 3 in this example), will result in comparing the values. If they are equal, the unification is successful.

Unifying a variable with a value, will first check whether the variable has already a value bound to it. If this is the case, both values will be unified again. If the variable is unbound, the value will be assigned to it. In this example, `x` will get 4 assigned to it. From this it can easily be seen that once a variable gets a value assigned to it, this value will not change anymore, as was said above.

It is also possible that we need to unify two variables. If the first variable is still free, the second variable will be assigned to it. Otherwise the value of the first variable will be unified with the second variable.

5.1.2 The Loco programming language

Now that a brief introduction to declarative programming has been given, we will move on to Loco itself. In this section we will describe Loco's syntax, using an example from [19]. This example is based on a part of the London Underground, shown in figure 5.1.

There is a lot of information, or knowledge, in figure 5.1. There are lines,

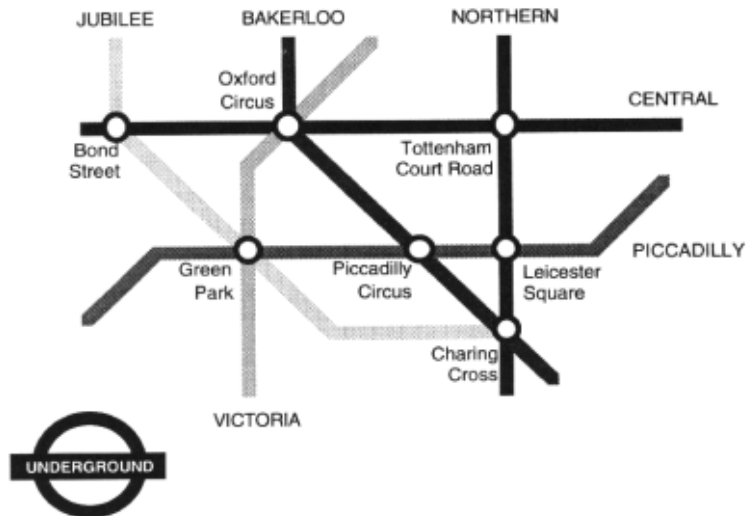


Figure 5.1: The London Underground [19]

stations, and connections between these stations. Using the figure, it's easy to see which stations are directly connected by which lines. This gives us 11 facts, shown in listing 5.1.

Listing 5.1: Connections

```
{ connected("bond_street", "oxford_circus", "central");
```

```

connected(" oxford_circus", "tottenham_court_road", "central");
connected(" bond_street", "green_park", "jubilee");
connected(" green_park", "charing_cross", "jubilee");
connected(" green_park", "piccadilly_circus", "piccadilly");
connected(" piccadilly_circus", "leicester_square", "piccadilly");
connected(" green_park", "oxford_circus", "victoria");
connected(" oxford_circus", "piccadilly_circus", "bakerloo");
connected(" piccadilly_circus", "charing_cross", "bakerloo");
connected(" totenham_court_road", "leicester_square", "northern");
connected(" leicester_square", "charing_cross", "northern") }

```

Suppose 2 stations are said to be nearby if they are on the same line with at most 1 station in between them, more facts can be added, as shown in listing 5.2.

Listing 5.2: Nearby

```

{ nearby(" bond_street", "oxford_circus");
  nearby(" oxford_circus", "tottenham_court_road");
  nearby(" bond_street", "tottenham_court_road");
  etc. }

```

In total 16 `nearby` facts can be derived. However, it is possible to derive these facts from the 11 `connected` facts. If 2 stations (X and Y) are directly connected by a line L , they are nearby. If there is another station Z , which is connected to Y via the same line L , X and Z are also nearby. This can be represented using logic rules, as shown in listing 5.3.

Listing 5.3: Nearby rules

```

{ nearby(X,Y): connected(X,Y,L);
  nearby(X,Z): connected(X,Y,L) & connected(Y,Z,L) }

```

The colon should be read as 'if', '&' becomes 'and'. Other valid symbols are '!' for 'not', and '|' for 'or'.

Now that the system contains some facts and rules, it is possible to query the system. In listing 5.4 some queries and their results are shown.

Listing 5.4: Querying the system

```

> connected(" green_park", "charing_cross", "jubilee");
:ok

> connected(" piccadilly_circus", otherstation , line );
:ok
otherstation = "leicester_square"
line = "piccadilly"

> #
:ok
otherstation = "charing_cross"
line = "bakerloo"

> connected(" green_park", "tottenham_court_road", "unexistingline");
:No match found

```

```
> nearby("oxford_circus", nearbystation);
:ok
nearbystation = "tottenham_court_road"
```

The first query has no free variables. While going through all the knowledge about `connected`, the system comes across the fact `{ connected("green_park", "charing_cross", "jubilee") }`. The terms of the query unify with the terms of this fact, so the query is successful.

The second query contains two free variables. This query can be read as 'Find the stations with which "picadilly_circus" are connected and the lines connecting them'. While going through its knowledge, the system finds a fact that unifies with our query: `otherstation` will be bound to "leicester_square", and `line` will be bound to "piccadilly". This query is also successful. However, there are more solutions for it. Since the number of solutions can be infinite (e.g. a query asking for a number greater than 3), it would be dangerous to try to print all solutions. Therefore the system will only return a single solution. If the user wishes to see more solutions, he can enter # as a query, as is shown in the third example query. This query will continue the search where it stopped last time.

Query 4 is an example of a query without result. The station "green_park" is not connected to "tottenham_court_road" and certainly not by "un-existingline". The system returns a failure.

The last query is much like the second one, but differs in the fact that it will not unify with a fact but with a rule. It unifies with the first `nearby` rule: "oxford_circus" is first bound to `X`, then `nearbystation` is bound to `Y`. This means that the query `connected("oxford_circus", nearbystation, L)` must be successful in order for the query to be successful. The system will now try to find a solution to this new query. It will be unified with `{ connected("oxford_circus", "tottenham_court_road", "central"); }`, which will be successful, meaning that the last query in our code listing also results in a success.

5.1.3 Summary

In this section Loco was briefly described. We started by introducing the declarative programming paradigm, after which we've shown Loco's syntax. Now that Loco has been introduced, we will move on to Symbioco, a symbiosis between Pic% (which was described in chapter 2) and Loco.

5.2 Symbioco

In this section we will introduce Symbioco, a symbiosis between Pic% and Loco. We will start by defining two goals we wish to accomplish in our implementation. Then we will continue by taking a look at the general approach we've chosen to implement Symbioco. Afterwards we'll see how Loco-expressions can be evaluated in Pic% and vice versa, to conclude with a short summary.

5.2.1 Goal

While implementing Symbioco, we had 2 clear goals in mind, which we will describe here.

Transparency Our first goal was that the symbiosis should be as transparent as possible, meaning that there should be no difference between for instance a Pic%-expression that will be evaluated in Pic%, and another Pic%-expression that will be evaluated in Loco. This implies that it should be possible to re-implement a certain Pic% function as a Loco-predicate (and vice versa), without having to change any code used to call this function (or predicate).

No syntactic changes Our second goal is related to the first one. Since we want transparency, we obviously do not want any syntactical changes to the languages involved. This means that for instance no additional syntax is introduced in Pic% to indicate that a certain expression should be evaluated in Loco and vice versa. Since there is no additional syntax, using the symbiosis should become easier, since there is no need for a programmer to understand additional syntax. It should also be possible to use existing code, like libraries, symbiotically.

5.2.2 Approach

In order to accomplish the goals described above we have chosen for an approach that resembles lillambi's approach (as described in section 4.3.3). By exploiting the fact that both the Loco-interpreter and the Pic% interpreter are written in a common language (namely Java), a layer was built that can evaluate both Loco and Pic% expressions, using the respective interpreters. A graphical representation of the Symbioco layer with the two interpreters on top of it is given in figure 5.2. Besides being able to start interpreters, there is a second point of interaction: all errors and output operations that occur in both interpreters are passed to the Symbioco layer. This way the

layer is able to redirect output to its own interface, and more importantly, take action when certain errors occur. It is by acting on these errors that the real symbiosis comes to life: when a certain predicate or function is not found by the current interpreter, symbioco will trap this error and try to evaluate the same expression in the other interpreter. Pic% and Loco are still treated like separate languages, each with its own grammar and internal representation. It is therefore obvious that in order for the other interpreter to understand the expression, some translation is required. The next section will discuss how expressions are translated from Pico to Loco and vice versa.

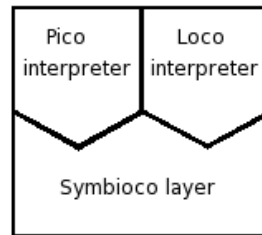


Figure 5.2: Representation of the Symbioco-layer

5.2.3 Translating expressions

When translating Pic% expressions to Loco and vice versa, it is obvious that good use can be made of their syntactic resemblance. In fact, it is this resemblance that enables us to implement the symbiosis without any syntactic changes. Consider for instance the two following expressions, both used to calculate a factorial.

```
fac(4)
fac(4,x)
```

The first expression is a Pic%-expression that calls the function `fac`. The second expression is a Loco-query where `x` will be unified with a certain value (Unifying the last value with the result is an idiom used in logic programming language to model return values). It can very easily be seen that without these descriptions it is impossible to say which expression belongs to which language. Both are valid Pic% and Loco expressions, which can be parsed in both interpreters.

Suppose the expressions in the example above are parsed in their respective interpreters. The first expression will result in an application of the function `fac` to the argument `4`. The second one however will result in a query with `fac` as symbol, and `4` and the logic variable `x` as arguments. As we shall

illustrate, these parse-trees –although composed out of fundamentally different elements– are quite similar. This similarity is exploited to translate Pic% function calls to Loco queries and vice versa.

5.2.3.1 Translating Loco expressions to Pic% expressions

The idea is to convert a Loco query to a Pic% function call. In order to do this we will exploit the similarity of the parse-trees that Loco and Pic% generate. We will illustrate this with the `fac`-example of the previous section. In figure 5.3, it is shown that although the parsetrees of both `fac` expressions, as generated by their respective interpreter, are indeed quite similar. This makes translating a Loco query to a Pic% function call easy. Symbioco will simply execute a function application in Pic%, with the query-symbol as name of the function. However, it can be seen that a problem arises when converting the arguments: the Pic% function `fac` only takes a single argument, but the query in Loco has 2, because it expects the value of `fac(4)` to be unified with `x`. To solve this problem Symbioco will simply “drop” the last argument from a Loco query, treating it as the return value of the function. When Pic% has completed execution, Symbioco will unify Pic%'s return value with the dropped argument. If this unification is successful, the query will succeed, otherwise it will fail.

However, dropping the last argument isn't the best solution if we want to call Pic% functions acting as predicates. Consider the following example.

```
greaterThan(4,3)
```

Suppose `greaterThan` is implemented in Pico, taking 2 arguments and returning either `true` or `false`. It would be rather odd to call this from Loco using `greaterThan(4, 3, x)`. This would mean that `x` gets unified with `true`, and that we should use `greaterThan(4, 3, x) & x` to have the desired result. Therefore, while translating the arguments, Symbioco will first check the number of arguments the Pic% function accepts. If this number equals the number of arguments specified in the query, all arguments will be passed. Otherwise Symbioco will drop the last specified argument as described above.

If no argument was dropped and Pic% has completed execution, Symbioco will make the Loco-query succeed or fail, depending on whether the return value is true or false. On any other value, Symbioco will throw an error.

Note that when Pic% predicates are called from Loco with an additional “return value” argument, this argument will be unified with `true` or `false`. When executing `greaterThan(4,3,x)` in Loco for instance, `x` will be unified with `true`.

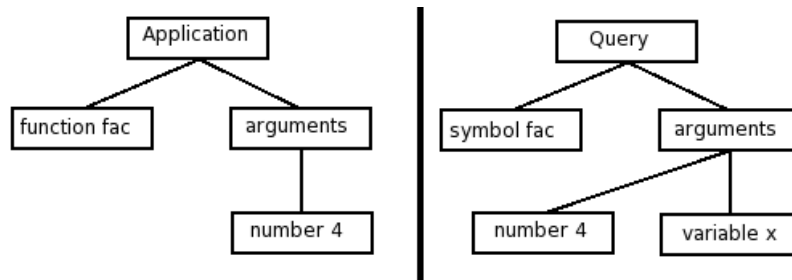


Figure 5.3: Conceptual representation of a Pic% (left) and a Loco (right) parsetree

This system is conform with our first goal, transparency. There is no difference between calling a Pic% function and executing a Loco query. Because of this, it is also very easy to replace a Pic% function with a Loco rule.

5.2.3.2 Translating Pic% expressions to Loco expressions

In the previous section it was shown how Loco expressions are translated to Pic% expressions. In this section the opposite, the translation of Pic% expressions to Loco expressions, will be shown.

When a function call in Pic% causes a lookup error, and control is given to Loco, this Pic% function call needs to be translated to a Loco query. In this translation, it will become harder to hold on to the principle of transparency. This is caused by some of Loco’s logic programming abilities, not present in Pic%. To illustrate this, consider the following query.

```
append(list1, list2, [1, [2]])
```

When such a query needs to be invoked from within Pic%, there are some problems to be dealt with. First of all, Loco can “return” multiple values at once. In the above example both `list1` and `list2` will be unified with a value. When returning to Pic%, both values can be considered return values.

A second problem is that it is impossible to predict on which position the value that needs to be returned to Pico will be: any argument is a potential return value. These two problems make it impossible to do the reverse of the Loco-to-Pic% translation, by simply adding an extra argument when executing a query.

The third problem is related to Loco’s ability to backtrack and generate more results. Even the simple example above can have 3 different results, where `list1` is respectively a list with no elements, a list containing a single

argument, 1 and a list containing both 1 and 2. It can be useful to get all these possible results with a single call in Pic%.

Enforcing transparency would mean throwing away a lot of the power of the declarative language symbiont, which is something we did not want to do. Throwing away Loco's backtracking features would render the entire symbiosis useless, since it is this power we want to be able to use in Pic%. Therefore a Pic% call resulting in a Loco query has to be called as shown in the following example.

```
append(list1:value(), list2:value(), [1, [2]])
or
append(list1[n]:value(), list2[m]:value(), [1, [2]])
```

The first call shows how the problem of multiple return values is addressed. Furthermore this technique introduces the ability to write "logic variables" in Pic% syntax, such that these return values can be in arbitrary positions of the call. In this example, Symbioco will replace the first two arguments by logic variables, and execute a query in Loco with `append` as symbol and the two logic variables combined with `[1, [2]]` as arguments. When Loco has finished, Symbioco will assign the values belonging to the logic variables to `list1` and `list2`. If one or more variables are unbound, Symbioco will throw an error.

The second call shows the solution to the problem of Loco's ability to generate multiple results. When the arguments marked as return values are tables, Symbioco will automatically execute a `findall` query in Loco, instead of just using `append` as symbol. When Loco returns, Symbioco will set `list1` to all possible results for the first argument of the `append` query, `n` to the amount of found results, and do the same for `list2` and `m` for the second argument.

Though this conflicts with our first goal, transparency, it allows us to use the full power of Loco, which we believe to be more important. Note that although the calls in the above example look quite unnatural, they are legitimate Pic% syntax and thus conform with our second goal.

As we have said above, the function call syntax introduces the ability to represent logic variables in Pic%. This syntax may seem odd, but is in fact very natural. To the Pic% programmer logic variables are represented as lazy arguments. In these arguments, the function `value()` can be used to get the value of the variable. The same can be said for the table syntax that is used to execute a `findall` query. In Pic% tables are defined by evaluating their body `size` times. Our syntax can thus be interpreted as executing `value()` until the table is filled up. This means that, on a conceptual level, the symbiosis is still transparent.

5.2.4 Summary

In this section an elaborate description of Symbioco was given. We first defined two goals we wished to achieve in this symbiosis: we did not want any syntactical changes, and we wanted our symbiosis to be fully transparent. The second goal was fully achieved, thanks to the syntactic resemblance between Pic% and Loco. The first goal was only partially achieved, because we did not want to give up the power of Loco in favour of transparency.

5.3 Validation

In this section we will demonstrate how Symbioco can be used in practice. We will do this by means of an example, namely the Rijmenam Example, as described in [14]. We will start by explaining the example, and then discuss the implementation step by step.

5.3.1 The Rijmenam Example

In the Rijmenam example, a shortest path algorithm is applied to geographic data, in order to find the shortest path between two cities. Originally this example was used to demonstrate that logic programming languages are very suitable to describe domain knowledge, but here it will more be used to show how Symbioco can be practically used.

There are five cities in the example (Rijmenam and four neighbouring cities), which are connected through a number of roads, each road with its own length. There are also one-way streets, i.e. roads which are prohibited in a certain direction. A graphical representation of this data is shown in figure 5.4.

We will now implement a symbiotic shortest path algorithm that can be applied to this data. We will start by showing how all the data can be easily represented using Loco, after which we will take a look at the actual implementation of a very basic shortest path algorithm.

5.3.2 Implementation

One may notice that the data represented in figure 5.4 is not very different from the data in figure 5.1 that was shown in section 5.1. We will start in the same fashion as we did then, by listing some facts to capture the knowledge from the figure. These facts are shown in listing 5.5, shown below.

Listing 5.5: Representing knowledge in Loco (1)

```
{ connected(" Bonheiden " , " Keerbergen " , 9);
```

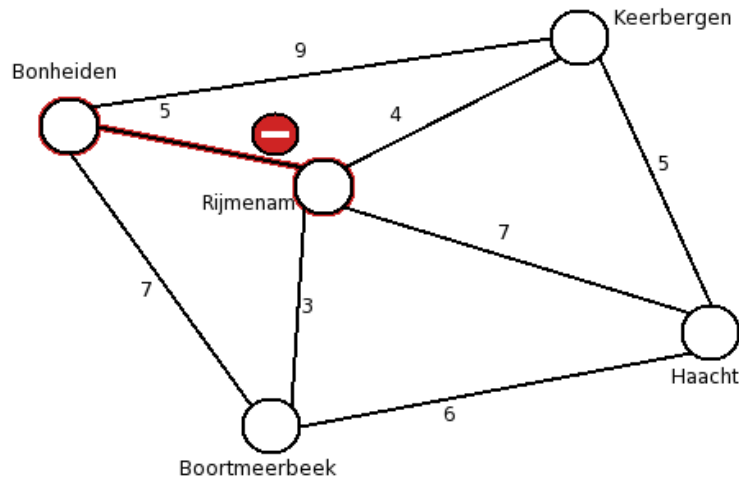


Figure 5.4: Graphical representation of Rijmenam domain knowledge [14]

```

connected(" Bonheiden ", " Rijmenam " , 5);
connected(" Bonheiden ", " Boortmeerbeek " , 7);
connected(" Rijmenam ", " Keerbergen " , 4);
connected(" Rijmenam ", " Haacht " , 7);
connected(" Rijmenam ", " Boortmeerbeek " , 3);
connected(" Keerbergen ", " Haacht " , 5);
connected(" Boortmeerbeek " , " Haacht " , 6);

prohibited(" Bonheiden " , " Rijmenam " ) }

```

The first eight facts represent the roads connecting two cities. The last fact indicates that it is forbidden to use the road from **Bonheiden** to **Rijmenam** in the direction of **Rijmenam**. Note that it is still possible to use this road in the opposite direction.

Most roads in the example can be used bidirectionally. Therefore a **connection** rule is added that allows to use connections in two directions, unless one of the directions is prohibited.

Listing 5.6: Representing knowledge in Loco (2)

```

{ connection(x,y,z): connected(x,y,z) & !prohibited(x,y);
  connection(x,y,z): connected(y,x,z) & !prohibited(x,y) }

```

Now that all connections are covered, we can move on to the algorithm. The implemented algorithm will first take a list of all possible paths. It will then calculate the length of each path separately, and keep track of the shortest path so far. We can subdivide this into three steps. First we need to be able to find paths, implying that a search algorithm will be needed. Since usually these algorithms use backtracking, and backtracking is something a declarative language like Loco offers by default, this algorithm was implemented in Loco, as shown in listing 5.7 below.

Listing 5.7: Finding paths in Loco

```
{ path(from, to, path): pathWithVisited(from, to, [from, to], path);

  pathWithVisited(from, to, visited, [from, to]):
    connection(from, to, z);
  pathWithVisited(from, to, visited, path):
    connection(from, y, z) &
    !member(y, visited) &
    append([y], visited, newvis) &
    pathWithVisited(y, to, newvis, p) &
    append([from], p, path) }
```

The `path` query allows a user to query the system for paths. It simply executes the `pathWithVisited` query, which does the actual searching, and holds on to a list of visited cities in order to prevent loops.

The second step is the ability to calculate a path's length. This is done by looping over all connections in a path and adding their lengths. An imperative language like Pic% is usually better in looping and performing mathematical operations like addition, so we've chosen to implement this part in Pic%.

Listing 5.8: Finding a path's length in Pic%

```
length(path):{
  total: 0;
  i: 1;
  pi: 0;
  j: 2;
  pj: 0;
  s: size(path);
  if(s<2, 0, {
    while(s >= j, {
      pi: path[i];
      pj: path[j];
      connection(pi, pj, l:value());
      total := total + l;
      i := j;
      j := j + 1
    })
  });
  total
}
```

Note that in order to get the length of a certain `connection` this code symbiotically calls Loco, as can be seen by the following line of code:

```
connection(pi, pj, l:value());
```

Finally we need to be able to calculate the length of a list of paths, and hold on to the shortest one. Once again this implies loops and mathematical operations, so this function will also be implemented in Pic%.

Listing 5.9: Finding the shortest path in Pic%

```
shortest(from, to): {
  path(from, to, paths[number]:value());
```

```

ln: -1;
p: -1;
if(number >= 1, {ln := length(paths[1]);
                 p := 1});
i: 2;
while(i <= number, {
  curr: length(paths[i]);
  if(curr < ln, { ln := curr; p := i});
  i := i+1
});
if(p = -1, display("No paths found", eoln),
  display("Shortest path is: ", paths[p],
    " with length ", ln , eoln))
}

```

Once again there is a symbiotic call to Loco, this time to get a list of all possible paths between two cities. This function contains two local variables to keep track of the shortest path: `p` and `ln`. The first one (`p`) will contain the index of the shortest path so far in the list of paths, the second one (`ln`) will contain the actual length of the path. The first path (if there is any) will be set to be the shortest one, after which the function will loop over all other paths, compare them with the first one, and adjust its local variables appropriately if it comes across a shorter path. Finally it will simply display its findings.

To conclude the implementation part of this example, we will show two function calls and their result.

Listing 5.10: Rijmenam examples

```

shortest("Bonheiden", "Haacht")

> Shortest path is: [Bonheiden, Boortmeerbeek, Haacht] with length 13

shortest("Boortmeerbeek", "New York")

> No paths found

```

5.3.3 Summary

In this section we have given a symbiotic implementation of the Rijmenam example. This example consists of applying a shortest path algorithm to a set of geographic data. This data was represented in Loco, and the algorithm was implemented in both Loco and Pic%. Pic% was used wherever loops and arithmetic operations were important, and Loco was used when backtracking was needed. Though the implemented algorithm is rather basic, the power and practical use of Symbioco was clearly shown.

5.4 Technical Aspects of Symbioco

Until now we have only talked about Symbioco on a very conceptual level. In this section we will go into much more detail and take a look at Symbioco's implementation.

Symbioco is implemented in Java, and is based on the implementation of JavaPico, a Pico interpreter written in Java. In this section we will start by briefly discussing the implementation of both the Pic% and Loco interpreter (which are also written in Java), and then elaborate on how these two interpreters are combined. Then we will see when exactly Symbioco will switch between the Pic% and Loco interpreter. We will finish this section with a description of how Pic% and Loco environments are restored.

5.4.1 Combining multiple interpreters

In this section we will take a very brief look at the implementation of the Pic% and Loco interpreters, and their similarities. We will then talk about how Symbioco combines these interpreters.

5.4.1.1 The Pic% and Loco interpreters

Conceptually the Pic% and Loco interpreters are very much alike. This is no coincidence: the Loco interpreter was built with the symbiosis in mind. In order to make the symbiosis as easy as possible, a special effort was made to make sure the Loco implementation resembled the (already existing) Pic% implementation. Because of this resemblance we are able to discuss both interpreters at the same time.

These interpreters are both stack-based. This means that central to the interpreter there is a stack consisting of several stack frames (called continuations). These continuations each specify a small piece of evaluation behaviour. Together they form the continuation stack, which represents the “future” of a program. Continuations are able to push and pop other frames on and off the stack. Evaluating a program can then be done by a simple loop that constantly executes the code of the continuation on top of the stack. Note that this implies that continuations are responsible for removing themselves from the stack. Since all computation is performed on a stack, the future of a running program can be captured and passed on when switching to symbioco.

The evaluation loop is executed by a `Process` object. The interpreters are able to run multiple processes at the same time (this is used to be able to

run multiple programs in different windows at the same time). Stack manipulation (pushing and popping continuations) is always done through this `Process`.

If any user interaction is required (in- and output, reporting errors), the current `Process` will handle this using a callback, usually provided by the graphical user interface.

5.4.1.2 Combining the `Pic%` and `Loco` interpreters

Since most continuations in `Pic%` and `Loco` call `Process` to manipulate the stack (as we've mentioned in the previous section), it would require a lot of adjustments to let the Symbioco-layer control these stacks directly. Instead, Symbioco uses the original `Process` classes from both interpreters, and manipulates the stack through them. In order to manage these `Processes`, a wrapper is built around them. Symbioco uses its own stack to manage these process wrappers. When executing a program, Symbioco will create a new process in one of the interpreters (the user can specify which interpreter Symbioco should start with), pushes it on its process stack, and start this process.

To find out whether a process is currently running, Symbioco also provides a “fake” callback for both interpreters. This means that technically Symbioco can be considered as a user of both interpreters. These callbacks notify Symbioco when execution finishes, or when errors occur during execution. Symbioco will act on both these events. When a process finishes execution cleanly, Symbioco will either give control back to another process or return the return value to the user. When a process stops execution due to an error, Symbioco will either forward this error message to the user, or create and start a new process to achieve symbiosis. We will elaborate on these actions in the upcoming sections.

5.4.2 Using errors as symbiosis joinpoints

As said in the previous section, Symbioco uses errors to switch between interpreters. However, not all errors cause Symbioco to make this switch. In fact, for both interpreters there is only a single error that is interesting to Symbioco, namely lookup errors. When in `Pic%` a function call is executed, the `Pic%` interpreter will first issue a dictionary lookup. If this lookup fails, the interpreter will notify its user that there is an error (through the callback). The same can be said for `Loco`. If a query gets executed, the `Loco` interpreter will do a dictionary lookup to find rules and/or facts to unify the

query with. When neither rules nor facts are found, a lookup error will be thrown in the same way as the Pic% interpreter. Note that this means that if there is any rule or fact with the same name as the query, no error will be thrown. Even if this rule or fact does not result in a solution, Symbioco will not be notified and the query will simply fail.

Using lookup errors to achieve language symbiosis is a common technique. In the symbiosis between Smalltalk and Soul this technique is also being used, as described in section 4.3.2.

However, in Pic% not all lookup errors are useful. Consider for example the following function call:

```
display(unknownVariable)
```

When `display`¹ is looked up in the dictionary, it will be found. Then the arguments will be evaluated: there is only a single argument, the variable `unknownVariable`. Evaluating this variable will also result in a dictionary lookup. This variable is not in the dictionary, so a lookup error will be thrown, that looks exactly the same as the error thrown when a function is not found. This is problematic, since Symbioco should only act on lookup errors caused by undefined functions. This problem is solved by inspecting the stack at the moment a lookup error is thrown. If there is an application evaluation continuation (which is responsible for the function lookup) on top of the stack, Symbioco will start another process. Otherwise Symbioco will just forward the error to the user.

In Loco this problem does not arise. The only lookups that can fail are rule and fact lookups. Variables are unified with values, or they simply remain free variables, so no lookup error is thrown by variable lookup.

When a lookup error (that is not caused by variable lookup in Pic%) is thrown, Symbioco will take the top process from its process stack, and inspect the top of the continuation stack. This top continuation contains either a Loco query or a Pic% function call. This expression will be translated to its equivalent in the other language. Symbioco will then create a new process, push it on its process stack, manipulate the stack of the process so that the translated expression can be evaluated, and start the new process. In the next section we will go into deeper detail on how Symbioco switches between different processes.

5.4.3 Switching between processes

There are two cases in which Symbioco needs to switch between processes. First there is the case described in the previous section. When a lookup error occurs Symbioco will switch to a new process to try to evaluate the

¹display is a native that prints out all its arguments sequentially

expression causing this error in another interpreter. The second case has also been mentioned before, in section 5.4.1.2. When a process finishes execution cleanly, (that is, if evaluation ends without an error being thrown), Symbioco needs to switch to the previous process, or simply return the return value to the user.

When switching between processes, we benefit greatly from the fact that the Pic% and Loco interpreters are stack-based. When an error is thrown, all control goes to Symbioco, but the continuations that still need to be executed are still on the stack. This is useful for two reasons: first of all Symbioco can find all necessary information to start a new process on the top of this stack. Secondly this means that if this process is to be continued, the future computation of the program is still available in the form of its continuation stack. All that needs to be done is some stack manipulation to make sure the continuation that caused the error isn't executed again (since this would obviously lead to infinite loops).

As said above processes can be continued, and new processes can be added. If a process finishes execution, Symbioco will take the final return value from this process, pop the process from the process stack, and insert the return value into the new top process on the stack. There are two exceptions to this. When there are no more processes on the stack, Symbioco will return the final return value to the user. The second exception was discussed in section 5.2.3.1. If a Loco query results in the execution of a Pic% function call, and the number of arguments in the query equals the number of arguments in the Pic% function, the return value of the Pic% function (expected to be either `true` or `false`) will determine whether the Loco query succeeds or fails.

If a process stops execution because of an error, Symbioco will add a new process, as described in the previous section.

When adding and removing processes special care needs to be taken of dictionaries. Suppose a Pic% program defines a function `f`, and then calls a Loco query, which in its turn calls the function `f`. In this example we want that the function call to `f` in the Loco query results in the execution of the `f` function defined in the original Pic% program. In order to accomplish this we will have to make sure the Pic% process in which `f` gets executed (which will be the third process on the stack) has a dictionary containing the function `f` from the first process. In other words: when jumping back and forth between two languages, not only do we have to store the computational context (the stack), we also have to store the environmental context (the dictionary). We will now describe Symbioco's dictionary management.

5.4.3.1 Managing dictionaries

Symbioco's dictionary management is fairly simple. If a new process is added, Symbioco will search its process stack for the last process of this type (so if the newly added process is a Loco process, Symbioco will search for the last added Loco process in the stack). Symbioco will then take the dictionary from this process, and insert it in the newly created process.

Symbioco will also restore dictionaries in the opposite direction. When a certain process has finished execution and will get popped from the process stack, Symbioco will save the dictionary from this process. When a process of the same type will be continued, Symbioco will first change that process' dictionary to the stored dictionary. Using this technique, it is possible to use functions (or rules) that are defined in a process that is created after the current process (and that has already finished execution).

5.5 Conclusion

In this chapter a detailed discussion of Symbioco, a symbiosis between Pic% and Loco, was given. This symbiosis allows for Pic% programs to perform Loco queries, and vice versa. We started with a brief introduction to Loco and to the declarative programming paradigm of which Loco is an example, after which we started our discussion on Symbioco with a conceptual description of how it works.

We then moved on to a validating example, in which we have shown how Symbioco can be used, and where we demonstrated the power of Symbioco. Certain functions could more easily be expressed as Loco queries where some others were easier to implement in Pic%. The symbiosis between Loco and Pic% allows us to implement a certain function (or query) in the language we prefer.

After the validating example we returned to our discussion on Symbioco, but instead of staying on a conceptual level we took a closer look at Symbioco's technical implementation details.

Now that we have developed a working symbiosis, the next step is to add AOP to Pic%, using Loco as a symbiotic pointcut language. This will be elaborately discussed in the next chapter.

Chapter 6

Prototype-Based Aspect-Oriented Logic Meta Programming

Having described a symbiosis between Pic% and Loco, Loco will be employed as a pointcut language to enable aspects in Pic%. Because of the symbiosis, it will be very easy to use certain Pic% functions to extract information from a certain pointcut.

We will start this chapter with a section on aspect-oriented logic meta programming, since we only discussed it very briefly in chapter 3. We will then show how aspects are introduced to Pic%, while at the same time discussing the changes that needed to be made to Loco and Pic% to support aspects, the way in which advice can be weaved, and how we can use Loco as a pointcut language. We will then continue with an example and show the usefulness of the symbiosis between the pointcut language and the base language.

6.1 Aspect-Oriented Logic Meta Programming

In section 3, aspect-oriented programming was elaborately discussed. It was concluded that the power of the pointcut language had a major impact on the ease with which aspects could be described, and that the use of a full-fledged declarative programming language might make describing aspects easier.

This is exactly what AOLMP is: describing aspects using a logic programming language. In this section aspect-oriented logic meta programming is further explained, based on [13]. The section will start by showing how a declarative programming language can be used as a pointcut language, after

which an example of AOLMP will be given. This section will be ended with a summary in which the most important differences between the approach described here and our own approach will be highlighted.

6.1.1 Using a declarative language to describe pointcuts

In [13], the logic programming language TyRuBa is used to describe aspects in Java. To allow TyRuBa to reason about Java-code, this is first represented in TyRuBa using facts. Once TyRuBa can reason about Java-code, it can be used to describe joinpoints in.

This section will start by showing how Java-code is represented in TyRuBa. Then it will be explained how joinpoints are described in TyRuBa.

6.1.1.1 Representing programs as facts

How programs are represented as facts varies, and determines the kind of information that is reified and accessible to meta programs. In TyRuBa, a mapping that represents classes means of facts which state that the class has certain methods, instance variables or constructors is assumed.

An instance variable in a class can be represented by a fact of the form:

```
var(?Class, ?VarType, ?VarName, { ...declaration code ...}).
```

This example also shows how Java code can be specified in TyRuBa: Java code is enclosed in curly braces. This “quoted” Java code will later be used by a code generator to create Java code with advice woven in. Pieces of quoted code may contain references to logic variables, an example of which will be shown later on.

Apart from instance variables, it is also possible to represent methods in TyRuBa. A method declaration is asserted by a fact of the following form:

```
method(?Class, ?ReturnType, ?MethodName, ?ArgTypeList,
      { ...declaration head ...},
      { ...method body ...}).
```

An example of how a Java class declaration is represented as a set of TyRuBa facts is shown in figure 6.1.

6.1.1.2 Describing aspects

In TyRuBa, four logic facts can be used to describe aspects. Using `mutually-Exclusive` it is possible to represent mutual exclusion between two methods.

<pre> class Stack { int pos = 0 ; Stack() { contents = new Object[SIZE];} public Object peek () { return contents[pos]; } public Object pop () { return contents[--pos]; } ... } </pre>	<pre> class(Stack). var(Stack,int,pos,{int pos = 0;}). constructor(Stack,[],{public Stack()}, {contents = new Object[SIZE]; }). method(Stack,Object,peek,[], {public Object peek()},{return...}). method(Stack,Object,pop,[], {public Object pop()},{return...}). ... </pre>
---	--

Figure 6.1: Java code for a Stack versus TyRuBa representation of this Java code [13]

```
mutuallyExclusive(Stack, push, pop)
```

The use of `mutuallyExclusive` facts will trigger the weaver to insert appropriate expressions at the beginning of methods. These facts are very specific, and do not allow the programmer to add any additional code.

The other three facts can be used by the programmer to add code however. First there is the `requires` fact, that makes sure a method `?m` in class `?c` is only executed if a certain `?condition` expression evaluates to `true`.

```

requires(?c, ?m, ?condition).
requires(Stack, push, { !full() }).
requires(Stack, pop, { !empty() }).

```

The two lower facts say that the `push` and `pop` method in the `Stack` class may only be executed if the stack is respectively “not full” and “not empty”. The most versatile facts to describe pointcuts in TyRuBa are `onEntry` and `onExit`, which can be used to specify code that has to be performed upon respectively entry and exit of a method.

```

onEntry(?class, ?method, ?statements).
onExit(?class, ?method, ?statements).

```

It is possible to extend the aspect language described by the four facts mentioned above, by adding logic rules. An example of this is shown below.

```

mutuallyExclusive(?c,?m1,?m2) :-
  mutuallyExclusiveList(?c,?l),
  element(?m1,?l),
  element(?m2,?l),
  NOT(equal(?m1,?m2)).

```

When this rule is added to the system, it becomes possible to specify an entire list of messages that are mutually exclusive, by simply adding a fact like the following:

```
mutuallyExclusiveList(Stack, [push,pop,peek]).
```

When the weaver looks for a `mutuallyExclusive` fact it will come across this rule, and all possible combinations of two non-equal message from the list will be `mutuallyExclusive`. The fact that the programmer can extend the aspect language is the fundamental advantage of using logic facts to declare aspects [13]

In the next section the `onEntry` and `onExit` methods will be used in a small example to describe a more general aspect (i.e. one where the programmer can specify advice code that needs to be executed).

6.1.2 Example

Suppose that all messages to the `Stack` class need to be logged. Before the message is executed "`executing msg`" should be printed, where `msg` is replaced by the actual message called. After the execution has finished, "`done executing msg`" should be printed, again with the actual message called substituted for `msg`.

This will be done using two facts in TyRuBa. First, the `onEntry` fact will be used to print the notification *before* the message is executed.

```
onEntry(Stack, ?method,
  { System.out.println("executing " + ?method); })
```

In order to print the notification *after* the message is executed, we do something similar with the `onExit` fact.

```
onExit(Stack, ?method,
  { System.out.println("done executing " + ?method); })
```

6.1.3 Summary

In this section an approach to aspect-oriented logic meta programming was shown. TyRuBa was used to describe aspects in a Java program. As we will see in the rest of this chapter there are some fundamental differences with our approach however.

First of all TyRuBa reasons about a number of facts representing Java code. In our approach we will not translate Pic% code to Loco code, instead Loco

will reason about the running `Pic%` program. This immediately shows a second difference. In TyRuBa advice code is added before the Java program starts running (i.e. TyRuBa employs source-code weaving), where our approach will add advice code in a running program (i.e. runtime weaving). Another difference is that the approach presented in this section is very specific. TyRuBa aspects in Java were only meant to solve the synchronization crosscutting concern (hence the `mutuallyExclusive` fact), and even the other three facts were originally meant for adding synchronization related actions. In our approach only a single, more general, fact will be used to describe aspects, as will be shown in the next section.

6.2 Aspects in Symbioco

This section describes how aspect-oriented logic meta programming is added in Symbioco. We will limit ourselves to a conceptual discussion here, and go into more technical detail later on in this chapter (see section 6.6). As was shown in the previous section, aspects in AOLMP are described using facts and rules. An example of how an aspect in Loco should look is shown below.

```
around(joinpoint, $ display("executing advice") $)
```

This is a simple aspect that displays "executing advice" for every joinpoint. We will come back to this example later on in this section, when we show how Loco is used as a pointcut language. First however, we will describe the changes made to `Pic%` and Loco in order to allow for aspects, after which the aspect weaver will be briefly discussed.

6.2.1 Introducing aspects in `Pic%` and Loco

In order to support aspects in Symbioco, some changes were made to the implementation of `Pic%` and Loco. The most invasive change is the addition of *quoted `Pic%` code* in Loco, used to specify advice code. Apart from that the evaluation engine of `Pic%` is slightly modified in order to call the aspect weaver when necessary. However, when no aspects are specified this change is unobservable by the programmer.

6.2.1.1 Quoted `Pic%` Code in Loco

The first change we will discuss is that of quoted `Pic%` code in Loco. In order to specify advice in Loco, we need to be able to write `Pic%` code that

can be parsed by the Loco parser. This wasn't a problem in the Symbioco implementation, since all Pic% code that was expected in Loco consisted of function calls, and these function calls are syntactically equivalent to Loco queries. In advice code, however, it should be possible to go beyond this very basic syntax and use all provisions that Pic% has to offer, such as operators and assignments/definitions. Using this Pic% syntax will cause problems when parsing Loco-code, since the Loco parser does not understand it.

This problem is solved by adding a new syntactic element in Loco, namely quoted Pic% code blocks. These blocks are syntactically represented as a Pic% expression enclosed in dollar-signs (e.g. `$ display("executing advice") $`). This is comparable to Smalltalk terms in SOUL (see section 4.3.2.1), except for the fact that the code between dollar signs does not get evaluated. The Loco parser will take the string between two dollar signs, and start the Pic% parser to parse this string. The resulting parsetree will then be treated as a boxed Pic% value in Loco, which will unify with free logic variables (but not with anything else). Using quoted code it becomes possible to use more advanced Pic% syntax like operators. Since symbiotic calls to Loco queries are also valid Pic% syntax, nothing stops the programmer from using them in quoted code blocks (and thus later on in advice code). Note that using the dollar sign syntax implies that the dollar sign should no longer be used as operator in Pic% in quoted code, in order to avoid parse errors in Loco.

6.2.1.2 Making the Pic% Runtime joinpoint-aware

Apart from allowing the programmer to specify advice code, it is also necessary to trap joinpoints in Pic%, in order to inform the weaver that it is possible that advice code needs to be woven in. Trapping joinpoints in Pic% is done by inspecting of the runtime stack, and thus via continuationframes. In this dissertation only continuationframes representing messages sends are trapped.

6.2.2 The Symbioco Aspect Weaver

The weaver used to weave advice-code into an actual Pic% program is fairly simplistic. Every time a message send is trapped, the weaver is called. The weaver will then search for advice that can be woven in by executing a query in Loco. If such an advice is found, the weaver will perform the actual weaving. When the weaver is done, Pic% will continue execution, possibly with a modified interpreter state that denotes the woven advice.

This approach of weaving, where the aspect weaver is called during execution of the base program and only then starts its search for advice, is called runtime weaving (see section 3.3).

6.2.3 Loco as a pointcut language

This section illustrates how an advice can be specified in Loco. In this dissertation only around advices are considered, since it is easy to simulate before and after advice using around advice. As said before, this advice is searched by the weaver by executing a query in Loco. This means that advice should be added in the form of facts or rules. When the weaver needs to find advice code for a certain joinpoint `aJoinPoint`, it executes the query shown below.

```
around(aJoinPoint, advice)
```

In this query, `aJoinPoint` is the actual joinpoint for which the weaver is searching advice code, and `advice` is a free logic variable that will unify with a quoted Pic% code block that needs to be woven in as advice code (if the query returns no result the weaver will simply return control to Pic% again without weaving anything in).

In order to add advice code one simply needs to add a fact as the one that was shown in the beginning of this section:

```
around(joinpoint, $ display("executing advice") $)
```

This example can be interpreted as 'around any joinpoint¹, display the string `executing advice`. It is possible to extend this example using a rule, as shown below.

```
around(joinpoint, $ display("executing advice") $):
  isMessageSend(joinpoint)
```

In this example the string `executing advice` will only be printed if `joinpoint` is a message send joinpoint. The query `isMessageSend` is actually a function from a Pic% library that is used symbiotically, since examining joinpoints is easier in Pic% than in Loco (as will be shown later on in this chapter). This immediately shows a big advantage of the symbiosis between the pointcut language and the base language: it is very easy to retrieve information from the latter.

6.3 Validation

In this section a validating example will demonstrate how logging code can be added to an object using aspects. First a Pic% object is defined, as shown below.

¹In this fact `joinpoint` is a free variable that matches with everything.

Listing 6.1: A simple Pic% login object

```

{
loginobj(): {login(user):: "logged in"; capture()};
l: loginobj()
}

```

This object implements a single method, `login`, which takes a username as argument and simply returns the string "logged in". Suppose we want to log when certain users log in. It is obvious that we can simply redefine the `login` method to make it check whether the specified user is one for which we want to enable logging, and to execute a logging call (in this example logging will simply be done using a call to `display`). However, this can more easily be solved by an aspect.

Consider the following aspect.

Listing 6.2: Example of a logging aspect

```

{
around(jp, ${ display(user, " logged in", eoln);
  proceed() }$):
  isMessageSend(jp) &
  message(jp, msg) &
  equals(msg, "login") &
  args(jp, arg) &
  elementAt(arg, 1, user) &
  loggedUser(user)
}

```

The advice code of this aspect is pretty straightforward: first the call to `display` is made to do the logging, after which `proceed` is called which will do the actual login (Note that this is actually a before-advice simulated using an around-advice). The pointcut description is more interesting, however. This aspect will only be triggered if the joinpoint (`jp` in the code) is the result of a message send, of which the message `msg` equals "login". The first specified argument also needs to be a `loggedUser`. In order to trigger this aspect when a certain user logs in, all that needs to be done is creating a fact in Loco like the one shown below.

```
{ loggedUser("Jake") }
```

By adding this fact, all logins of the user with username "Jake" are logged. However, in approximately the same way we can log the logins of an entire group of people. This is illustrated by the following example.

```
loggedUser(user):
  getUserInfo(user, info) &
  isFemaleUser(info)

```

Assuming that the `getUserInfo` and `isFemaleUser` rules are implemented, this would immediately enable logging for all female users. This once again illustrates the power of using a logic programming language as pointcut language.

6.4 Symbiotic Pointcut Predicates

In this section the use of the symbiosis between the pointcut language and the base language will be shown. First joinpoint representation in Pic% will be explained. Then we will take a look at how some pointcut predicates are implemented in Pic%. Finally, the use of the symbiosis between Pic% and Loco will be discussed.

6.4.1 Joinpoint representation in Pic%

This section gives a brief overview of how joinpoints are represented in Pic%. Recall from chapter 2 where we elaborately discussed Pic%, that all internal Pic% grammar was accessible from within Pic% programs through tables. A dictionary was for instance represented as a table of size three, containing a key, a value and a pointer to the next dictionary.

Joinpoints are represented in Pic% in roughly the same way. A joinpoint can be indexed as a table too, where the first index contains a tag indicating the type of the joinpoint. A message application joinpoint for instance, is represented as a table of size four containing its tag, the object to which the message was sent, the message that was sent and a table containing the arguments for the sent message.

6.4.2 The Pic% joinpoint predicate library

Using the joinpoint representation described in the previous section, it is fairly easy to construct a small Pic% library in which joinpoint predicates are implemented. Note that it is only possible to implement these joinpoint predicates in Pic%, since there joinpoints are represented as tables. Thanks to the symbiosis between Loco and Pic%, Loco can use these predicates symbiotically.

A small subset of the Pic% joinpoint library is shown below, together with its implementation.

Listing 6.3: Subset of the Pic% joinpoint library

```
{
isMessageSend(joinpoint): joinpoint[1] = "mAP";
message(joinpoint): joinpoint[3];
```

```
args(joinpoint): joinpoint[4]
}
```

As can be seen it is very easy to implement these functions in Pic%. Implementing such a library would certainly be more tedious in Loco.

6.4.3 The use of language symbiosis

The examples in the previous section make it clear that the symbiosis between Pic% and Loco is extremely useful. Pic% is able to inspect its own state using its reflection mechanism, and can thus easily implement pointcut predicates that depend on the dynamic state of a Pic% program. Loco can call these predicates through the symbiosis, giving us a powerful mechanism to describe pointcuts based on dynamic properties of the running program. In addition, the fact that a logic programming language is used as pointcut language allows for recursively defined pointcuts. This will be illustrated with an example.

Suppose that in a certain system all statechanging methods need to be logged. Instead of listing all statechanging methods (as would be necessary in a more conventional pointcut language), this can be solved by implementing a recursive rule `stateChanging` as follows.

Listing 6.4: `stateChanging` rule

```
{
  stateChanging(method):
    containsAssignment(method);

  stateChanging(method):
    allCalls(method, listOfCalls) &
    member(call, listOfCalls) &
    stateChanging(call).
}
```

These rules state that a method is statechanging if it contains an assignment (for the sake of clarity local variables are ignored), or if it calls another statechanging method. The second rule recursively calls itself, thereby showing the real power of a logic programming language.

Note that this example also needs the symbiosis between Loco and Pic%. Since there is no representation of Pic% methods in Loco (as for instance in TyRuBa, as shown in section 6.1), Loco can not decide whether a certain method contains an assignment. It is also impossible for Loco to list all the calls made by a certain method. In Pic% on the other hand this is possible. Methods are first-class in Pic% (as shown in section 2.4.2.3), so it is possible to dissect them from within Pic%.

6.5 AOLMP with Prototypes

As shown in the previous chapter, our pointcut language allows the programmer to describe pointcuts based on dynamic properties of the running program. This is exactly what we needed to allow for aspects in prototype-based languages. Recall from chapters 2 and 3 that prototype-based languages were much more dynamic than their class-based counterparts, and that their lack of static structure was the main cause why they weren't suitable for aspect-oriented programming. This program is tackled by using Loco as pointcut language, since it can use Pic% symbiotically to inspect the dynamic properties of the running program.

This will be illustrated with some examples.

- In a prototype-based language traits [38] or shared parents can be used to represent class-like structures. Using our approach it is possible to describe pointcuts based on these structures, i.e. it is possible to weave code around the methods of all objects delegating to a specific trait of parent.
- Another example is that advice can be woven around the methods of specific objects only, i.e. singleton objects, as opposed to class-based approaches where advice is specified at the class level and as a consequence affects all instances of the class. Error-handling and tracing are examples of aspects which may benefit from the prototype approach, as it allows the instrumentation of only the objects under inspection.

6.6 Technical aspects of Aspects in Symbioco

As in the previous chapter we have only limited ourselves to a conceptual description up to this point. This section will go into much further detail about how aspects are added to Symbioco. In this section the implementation of the weaver will be discussed, and an elaborate description of how around-advice is weaved will be given.

6.6.1 The Symbioco Aspect Weaver Revisited

As mentioned in section 6.2.1, we have chosen to implement a very simplistic weaver. When a continuationframe is trapped as a joinpoint. it sends an `examineJoinPoint` message to the `Weaver` object, passing a `JoinPoint` instance that represents the current point of execution.

When the `Weaver` object receives the `examineJoinPoint` message, it will execute the query `around(aJoinPoint, advice)` to find advice in a new Loco process. This Loco process is created similar to how Symbioco normally creates Loco and Pic% processes. The only difference with the system explained in section 5.4.3 is that the weaver uses another `CallBack` class so that errors and finished execution are reported to the `Weaver` class instead of to Symbioco.

When the Loco process is finished, the weaver will check the result of its query. If there is no result, it will return control to the process that originally called the weaver, and execution will just continue. If a result was found, the weaver will weave the advice code into the execution stack of this process. An elaborate description of how this weaving is done will be given in the next section.

6.6.2 Weaving around advice

As mentioned before only around advice is handled by the weaver. However, since before and after-advice can easily be simulated using around-advice this is not considered a fundamental limitation. The weaving process will be discussed in two steps. First the stack manipulations made by the weaver when a joinpoint is triggered will be discussed. Then we will show how the stack is manipulated if a call to `proceed` occurs. Finally we will illustrate how the stack manipulation works using a small example.

6.6.2.1 Stack manipulations by weaver

As mentioned before, when a Pic% continuationframe notifies the weaver, the latter will start looking for around advice to be woven into the Pic% program. The state of Pic%'s continuation stack at the moment the weaver starts looking for such advice is graphically represented in figure 6.2.

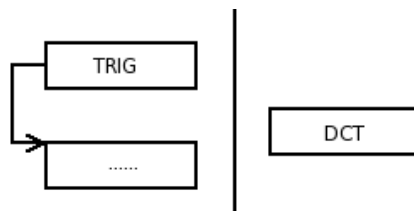


Figure 6.2: Weaving Pic% advice

On top of the stack is a certain continuationframe that triggered the weaver. This continuationframe will from now on be referred to as `TRIG`. Pic% also

holds a dictionary for lookup-purposes. This dictionary is shown on the right side of the figure. The `Pic%` dictionary at the time a joinpoint is triggered will be referred to as the original dictionary.

When the weaver finds advice, the actual weaving can begin. First the weaver will remove the `TRIG` frame from the top of the stack, and replace it by a `CFReturn` frame. This frame holds a reference to a certain dictionary, and when it gets executed it will set `Pic%`'s dictionary to the dictionary it holds, i.e. it will restore `Pic%`'s environment). This instance of `CFReturn` holds the original dictionary. This is shown in figure 6.3.

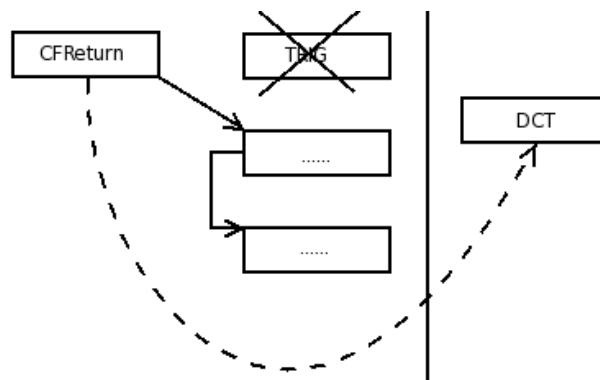


Figure 6.3: Preparing install of extended dictionary

The last step for the weaver is to make sure that the advice-code gets evaluated. In order to do this a `CFEval` frame is pushed on top of the stack. This frame will hold the parsetree of the advice code, and when it gets executed it will evaluate this parsetree. However, it is likely that there is no `proceed` function defined in the original `Pic%` dictionary, and it is also likely that the advice code will at a certain point call `proceed`. Therefore a new dictionary, which extends the original dictionary, is created. We call this dictionary the aspect-dictionary. In this aspect-dictionary a `proceed` native is added. The exact workings of this native will be described in the next section.

In order to allow the programmer to use variables from the advice query inside advice-code, all logic variables in the query are added to the aspect-dictionary too. The new state of the `Pic%` continuation stack is shown in figure 6.4.

When the advice for the joinpoint is set up, the weaving process is finished. The weaver will now restart the `Pic%` process. The advice code will be executed until `proceed` is called. At that time, the `proceed` native will manipulate the stack again in order to execute the `TRIG` frame, representing the original code. This is explained in detail in the next section. If `proceed`

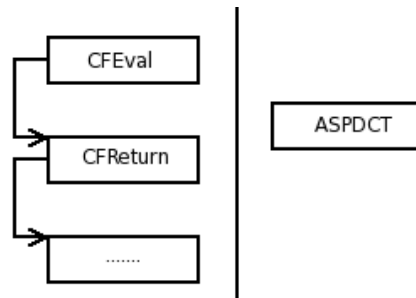


Figure 6.4: Evaluate advice code

does not get called, the advice code finishes normally using the `CFReturn` frame which restores the original dictionary. This means that the original code is *replaced* by the advice code.

6.6.2.2 Stack manipulations by `proceed`

The `proceed` native is created by the weaver and added to the aspect-dictionary when a joinpoint is triggered. The `proceed` native holds two local variables: the `TRIG` frame (that needs to be executed if `proceed` is called), and the original `Pic%` dictionary. The original dictionary will be used as the evaluation dictionary when the `TRIG` frame is executed, as this frame should not use the `proceed` native or any of the logic variables that were added in the aspect-dictionary (which may shadow variables in the original dictionary).

The state of the `Pic%` continuation stack at the time `proceed` is called is shown in figure 6.5.

On top of the stack, there is a `NATEval` frame, which is used to evaluate a native (in this case the `proceed` native). The next frame is an `ACTION` frame. This could be a definition, assignment, or any other action that has lead to a call to `proceed`. Underneath this frame the `CFReturn` and `CFEval` frames pushed by the weaver are still present. All frames between the `ACTION` and `CFEval` frame represent the “continuation” of the advice code, i.e. the actions that still need to be taken when `proceed` has finished.

The `proceed` native will first replace the `NATEval` frame by another `CFReturn` frame. As mentioned before, this frame will restore the `Pic%` dictionary to the dictionary it holds. This instance of the `CFReturn` frame holds the aspect-dictionary (that at this time may have been extended by the advice code already). The aspect-dictionary has to be used as the evaluation dictionary when `Pic%` returns from the `proceed` call so that after advice gets

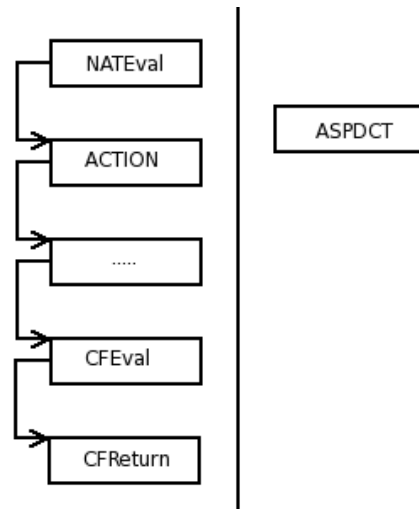


Figure 6.5: Calling proceed

executed using this dictionary. The state of the stack after these changes is shown in figure 6.6.

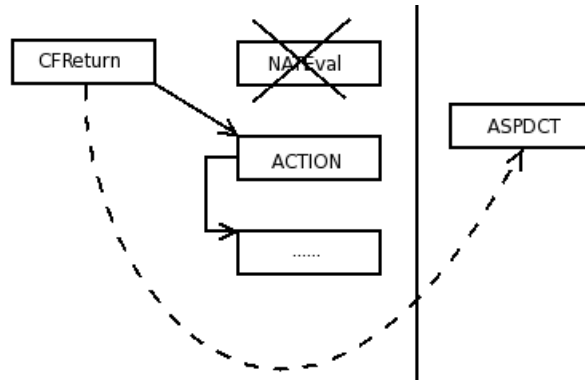


Figure 6.6: Restoring original dictionary

Then the `proceed` native will make sure the TRIG frame gets executed by pushing this on top of the stack. It will also set `Pic%`'s dictionary to the original dictionary, for the reason we mentioned before. The stack after this final step is represented in figure 6.7.

We have now given a rather detailed description of the weaving process. How this works in practice will be shown in the next section with a small

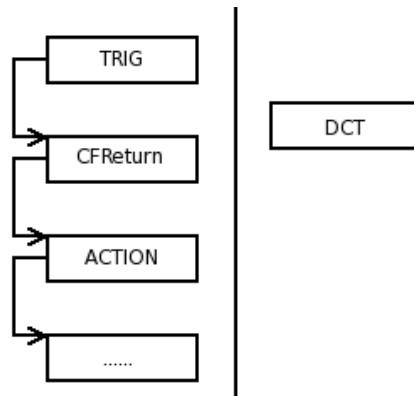


Figure 6.7: Proceeding with original call

example.

6.6.2.3 Example

In this section an example will be given of how the stack-manipulations work in practice. This example implements around advice that adds logging to message sends. For this example we will use the following pointcut and advice.

```
around(jp,
  ${display("executing ", msg);
  return: proceed();
  display("done executing ", msg);
  return }$):
isMessageSend(jp) &
message(jp, msg)
```

This will simply display "executing msg" and "done executing msg", (where msg is the message that was sent, this illustrates that logic variables in the pointcut can be used in the advice) before and after any message send, and return the result of this message send. The `message` query is in fact a `Pic%` function that returns the message of a certain joinpoint if this joinpoint is a message send joinpoint.

Now suppose the following code is executed in `Pic%`.

```
aVar: anObject.aMessage()
```

In this code `aMessage` is sent to `anObject`, and the result is put in the variable `aVar`. During the execution `Pic%` will, at a certain point, execute

the actual message send (using a message application (MAPL) frame). This will trigger the weaver, and the advice code shown above will be woven in. When Pic% restarts, there will be a CFEval frame on top of the stack, and the Pic% dictionary will contain the `proceed` native, and the `msg` and `jp` variables.

The advice code will now be executed, and "executing aMessage" will be displayed. Then the `proceed` native will be called as a result of the definition of `return`. The MAPL frame that originally triggered the weaver will now be put on top of the stack again, followed by a CFReturn frame to restore the aspectdictionary.

When the `proceed` native has finished executing, Pic% will start with the execution of the MAPL frame. When this execution has finished the CFReturn frame will restore the aspectdictionary, and the result of the MAPL frame will be inserted in the ACTION frame, in this case a definition.

This brings us back into the advice code, where `return` is bound to the return value of `anObject.aMessage()`, "done executing aMessage" is displayed, and `return` is returned. At this point we have come to the CFReturn frame pushed by the weaver, that will restore the original dictionary, and pass the value of `return` to the next frame (which is the definition of `aVar`). Note that if `proceed` has finished execution, the aspectdictionary is restored, meaning that it is possible to call `proceed` again. In fact, it is possible to call `proceed` numerous times.

6.7 Conclusion

This chapter dealt with the addition of aspect-oriented logic meta programming to the prototype-based language Pic%. It started with explaining what constitutes AOLMP, and some examples of AOLMP were given. Then the discussion on our approach started. A conceptual description of how aspects were introduced in Pic% and Loco and how the Symbioco aspect weaver works was given, followed by a validating example.

We then explained why the symbiosis between the pointcut language Loco and the base language Pic% is useful. Using this symbiosis Loco is able to call Pic% functions that inspect the state of the running Pic% program using reflection. Since there is no representation of the Pic% program in Loco this is the only way to inspect Pic% programs.

After this explanation we showed how our approach can be used to describe aspects in a prototype-based language. Since the symbiosis allows our pointcut language to inspect the dynamic properties of the running program, it is possible to describe pointcuts based on these dynamic properties, which is what we wished to achieve.

This chapter was concluded with a technically detailed description of the

aspect weaver implementation, and an overview of how exactly aspects are weaved in.

Chapter 7

Conclusions

In this dissertation we set out to discuss the benefits of having a symbiosis between the pointcut language and the base language in prototype-based aspect-oriented logic meta programming. This chapter presents our conclusions on this matter.

The first three chapters were used to give an overview of the domains of prototype-based programming languages, aspect-oriented programming and language symbiosis. The last two chapters discussed the proof-of-concept implementation supporting the thesis. In chapter two prototype-based languages were discussed. It was shown that prototype-based languages are much more dynamic than their class-based counterparts. Instead of a static class-structure, prototype-based languages only feature objects that are created and cloned at runtime. This lack of static structure makes reasoning about prototype-based software more difficult.

In the third chapter aspect-oriented software development was described. The problem of crosscutting concerns was explained, and some approaches to deal with this problem were shown. It was explained how this problem was solved by aspect-oriented programming, where advice code is woven in at certain joinpoints. The joinpoints where advice code needs to be woven in are described using a pointcut language. It was concluded that this pointcut language is the most important feature in aspect-oriented programming, and that a high degree of interaction between the pointcut language and the base language would be beneficial to the description of joinpoints in prototype-based languages.

The fourth chapter introduced language symbiosis. Two languages are symbiotically related if they have the ability to share one another's concepts (for instance an object-oriented language that is able to execute a query in a declarative language it's symbiotically related to), in a transparent way. Language symbiosis allows two languages to interoperate, and offers a high degree of interaction between these languages.

Chapter five and six gave an elaborate description of our own work. In chapter five Symbioco was discussed. Symbioco is a language symbiosis between the prototype-based language Pic% and the declarative language Loco. The chapter started with a brief introduction to Loco, after which the symbiosis itself was elaborately discussed. A validating example was given in the form of a shortest path algorithm that was implemented partially in Pic% and partially in Loco, and used symbiotic calls to find a solution.

In chapter six this aspect-oriented logic meta programming was added to the prototype-based language Pic%, using Loco as a symbiotic pointcut language. This chapter started by explaining aspect-oriented logic meta programming, stating that it is a special kind of aspect-oriented programming where the pointcut language is a full-fledged declarative programming language. The approach to the addition of AOLMP was thoroughly discussed, after which it was shown that this approach was suitable to describe pointcuts in a prototype-based language.

In the rest of this chapter our approach will be evaluated, by giving an overview of its contributions and limitations. This dissertation will then be concluded by proposing some possibilities for future work and research.

7.1 Evaluation

In this section an overview will be given of the contributions and limitations of our implementation of a symbiosis between the pointcut language and the base language in prototype-based aspect-oriented logic meta programming. First the contributions will be elaborated on, then the limitations will be discussed.

7.1.1 Contributions

In this section, the contributions of our approach are listed and described.

7.1.1.1 Runtime weaving

When deciding what kind of weaver to use for our approach, there weren't a lot of options. The entire point of our pointcut language is to capture dynamic, runtime behaviour of the base program. In addition, language symbiosis only works at runtime. Since our pointcut language depends on a symbiosis with the base language, our aspect weaver employs runtime weaving.

Runtime weaving provided us with a number of advantages. As mentioned in section 3.3 this kind of weaving is very dynamic. It is possible to add

and remove aspects at runtime, and to enable or disable the weaver completely. Additionally, runtime weaving allows access to runtime values (such as properties and return values of message sends) to decide whether advice should be executed [22].

7.1.1.2 Recursive pointcut descriptions

As shown in section 6.1, using a full-fledged logic programming language as pointcut language makes it easier to describe pointcuts. Since a Turing-complete language is used as pointcut-language, it is possible to express pointcuts recursively. This was shown using an example in section 6.4, where a recursive pointcut description was used to find joinpoints triggered by state-changing methods.

7.1.1.3 Pic% reflection through symbiosis

Pic% offers a powerful reflection mechanism using tables to represent first-class objects (as described in section 2.4.1.4). This makes it possible to reason about the computational and environmental state of a running Pic% program. Because of the symbiosis between the pointcut language Loco and the base language Pic%, a programmer can describe certain conditions for a pointcut in Pic% using reflection, and use these conditions symbiotically. An example of this was given in section 6.4, where a library was described that contained joinpoint predicates and functions which were implemented in Pic% using a joinpoints' table representation. Thanks to the symbiosis these predicates and functions can easily be called from within Loco.

7.1.1.4 Prototype-based approach

The most important contribution of our approach is that it allows for aspects in a prototype-based environment. Prototype-based languages are more dynamic than their class-based siblings, and this makes it very hard to describe pointcuts in them [8]. In our approach we are able to use Pic% reflection and recursive pointcut descriptions to get a hold on the structure of a prototype-based language, as will be illustrated with some examples.

- In a prototype-based language traits [38] or shared parents can be used to represent class-like structures. Using our approach it is possible to describe pointcuts based on these structures, i.e. it is possible to weave code around the methods of all objects delegating to a specific trait of parent.

- Another advantage is that advice can be woven around the methods of specific objects only, i.e. singleton objects, as opposed to class-based approaches where advice is specified at the class level and as a consequence affects all instances of the class. Error-handling and tracing are examples of aspects which may benefit from the prototype approach, as it allows the instrumentation of only the objects under inspection.

7.1.2 Limitations

Besides the advantages described in the previous section, there are also some limitations to our approach. In this section we will describe these limitations of our implementation.

7.1.2.1 Execution Speed

One of the main limitations of our approach is the fact that it's very slow. There are multiple reasons for this.

Runtime weaving In our approach we have chosen to implement a runtime weaver. Runtime weavers are more dynamic than e.g. source code weavers, but they are also slow. At runtime, *all* events that may lead to advice code being woven in will call the weaver. In our system this means that every message send will lead to a call to the weaver. On top of that, every possible joinpoint will give rise to a dynamic if-test: the weaver always has to launch a Loco query to check whether the joinpoint satisfies a certain advice to be woven in.

Weaver implementation The second reason why this approach is so slow is the weaver itself. It was never our goal to implement an optimized weaver, instead the implementation should be regarded as a proof of concept.

7.1.2.2 Multiple advices

A second limitation of our system is that it does not support multiple advices. When multiple advices are applicable for a certain joinpoint, only the first one will be executed. Although multiple advices are quite useful (consider for example a stack in which we want to weave a synchronization aspect to limit concurrent access to the stack, and a logging aspect to log pops and pushes), this was considered to be outside the scope of this dissertation.

7.2 Future research

In this section some possibilities for future work and future research will be proposed.

7.2.1 Weaver improvements

As said before, the weaver used in our approach is only very basic. The first proposal for future work consists of improving the weaver implementation. Optimizing the weaver would make our approach faster and therefore more useful. This could for instance be done by using partial evaluation, where a part of the weaver's work is done at compile time so that it has less work at runtime. This way it can be possible to decide that for some messages sends in Pic% no pointcut query has to be executed. Another way to optimise the weaver would be by caching results of Loco pointcut queries.

Another possible improvement for the weaver consists of adding support for weaving in multiple advices.

7.2.2 Prototype-based Idioms

An interesting area for future research is the study of how prototype-based programming idioms can be used to describe pointcuts, i.e. how certain idioms offer a pointcut language the necessary structure to capture cross-cutting concerns.

An example of such an idiom that was already discussed in this dissertation are Self's traits [38], but maybe (and hopefully) more idioms can be found.

7.2.3 Representing Pic% programs as Loco facts

In section 6.1 it was shown how Java programs can be represented as logic facts, and how these facts can be used to reason about the program and to describe pointcuts. In section 6.4 we used Pic%'s reflection mechanism symbiotically for this purpose. A possibility for future research is to also translate Pic% code to Loco facts (by for instance converting Pic% parsetrees to Loco functors), so that it becomes much more convenient to reason about Pic% source code in Loco.

Bibliography

- [1] H. ABELSON, G. J. SUSSMAN, AND J. SUSSMAN, *Structure and Interpretation of Computer Programs*, MIT Press, 1993.
- [2] *The aspectj programming guide*
(<http://www.eclipse.org/aspectj/doc/released/progguide/index.html>).
- [3] *Aspectj.org website*. <http://www.eclipse.org/aspectj/docs.php>.
- [4] L. BERGMANS AND M. AKSIT, *Composing multiple concerns using composition filters*.
- [5] L. BERGMANS, M. AKSIT, AND B. TEKINERDOGAN, *Aspect composition using composition filters*.
- [6] G. BLASHEK, *Object-oriented programming with prototypes*, 1994.
- [7] J. BRICHAU, K. GYBELS, AND R. WUYTS, *Towards linguistic symbiosis of an object-oriented and a logic programming language*, 2002.
- [8] T. CLEENEWERCK, K. GYBELS, AND A. PEETERS, *Aspects in a prototype-based environment*, 2004.
- [9] W. DE MEUTER, J. DEDECKER, AND T. D'HONDT, *Intersecting classes and prototypes*, in Proceedings of PSI-Conference, Novosibirsk, Russia, Springer-Verlag, 2003.
- [10] W. DE MEUTER, T. D'HONDT, AND J. DEDECKER, *Pico: Scheme for mere mortals*, 2004.
- [11] W. DE MEUTER, S. GONZALEZ, AND T. D'HONDT, *The design and rationale behind pico*, 1999.
- [12] K. DE SCHUTTER, *Lillambi website*.
<http://allserv.ugent.be/~kdschutt/lillambi/>.
- [13] K. DE VOLDER AND T. D'HONDT, *Aspect-orientated logic meta programming*, in Reflection '99: Proceedings of the Second International Conference on Meta-Level Architectures and Reflection, London, UK, 1999, Springer-Verlag, pp. 250–272.

- [14] M. D'HONDT, W. DE MEUTER, AND R. WUYTS, *Using reflective logic programming to describe domain knowledge as an aspect*, 1999.
- [15] T. D'HONDT AND W. DE MEUTER, <http://pico.vub.ac.be>.
- [16] T. D'HONDT AND W. DE MEUTER, *Of first-class methods and dynamic scope*, 2003.
- [17] *Dictionary*. <http://www.dictionary.com>.
- [18] V. J. ET AL., *Introduction to aosd (course slides techniques of software architecture)*, 2005.
- [19] P. FLACH, *Simply Logical: Intelligent Reasoning By Example*, Wiley, 1994.
- [20] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [21] K. GYBELS, http://prog.vub.ac.be/~kgybels/research/linguistic_symbiosis.html.
- [22] K. GYBELS, *Aspect-oriented programming using a logic meta programming language to express cross-cutting through a dynamic joinpoint structure*, 2001.
- [23] K. GYBELS, *Soul and smalltalk - just married: Evolution of the interaction between a logic and an object-oriented language towards symbiosis*, in Proceedings of the Workshop on Declarative Programming in the Context of Object-Oriented Languages, 2003.
- [24] K. GYBELS, R. WUYTS, S. DUCASSE, AND M. D'HONDT, *Inter-language reflection*, 2005.
- [25] <http://www.alpha-works.ibm.com/tech/hyperj>.
- [26] Y. ICHISUGI, S. MATSUOKA, AND A. YONEZAWA, *Rbcl: A reflective object-oriented concurrent language without a run-time kernel*, in IMSA '92 International Workshop on Reflection and Meta-Level Architectures, 1992.
- [27] G. KICZALES, J. LAMPING, A. MENDHEKAR, C. MAEDA, C. V. LOPES, J.-M. LOINGTIER, AND J. IRWIN, *Aspect-oriented programming*, in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, 1997, Springer-Verlag.

- [28] H. LIEBERMAN, *Using prototypical objects to implement shared behavior in object-oriented systems*, in OOPSLA '86: Conference proceedings on Object-oriented programming systems, languages and applications, New York, NY, USA, 1986, ACM Press, pp. 214–223.
- [29] H. LIEBERMAN, L. STEIN, AND D. UNGAR, *Treaty of orlando*, in OOPSLA '87: Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum), New York, NY, USA, 1987, ACM Press, pp. 43–44.
- [30] J. NOBLE, A. TAIVALSAARI, AND I. MOORE, *Prototype-Based Programming: Concepts, Languages and Applications*, Springer-Verlag, 1999.
- [31] P. STEYAERT, *Open design of object-oriented languages, a foundation for specialisable reflective language frameworks*, 1994.
- [32] D. SUVEE, W. VANDERPERREN, AND V. JONCKERS, *Jasco: an aspect-oriented approach tailored for component based software development*, in AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development, New York, NY, USA, 2003, ACM Press, pp. 21–29.
- [33] D. SUVEE, *Aspect weaving (course slides techniques of software architecture)*, 2005.
- [34] D. SUVÉE, W. VANDERPERREN, AND V. JONCKERS, *Jasco: an aspect-oriented approach tailored for component based software development*, 2003.
- [35] <http://users.rcn.com/jkimball.ma.ultranet/biologypages/s/symbiosis.html>.
- [36] A. TAIVALSAARI, *Classes vs. prototypes - some philosophical and historical observations.*, 1996.
- [37] R. TOLKSDORF AND K. KNUBBEN, *dself - a distributed self*, 2001.
- [38] D. UNGAR, C. CHAMBERS, B.-W. CHANG, AND U. HOLZLE, *Organizing programs without classes*, Lisp and Symbolic Computation, 4 (1991), pp. 0–.
- [39] D. UNGAR AND R. B. SMITH, *Self: The power of simplicity*, in OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications, New York, NY, USA, 1987, ACM Press, pp. 227–242.
- [40] T. VAN CUTSEM AND S. MOSTINCKX, *A prototype-based approach to distributed applications*, 2004.

- [41] R. WUYTS, *A Logic Meta Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*, PhD thesis, 2001.