Macchine Virtuali per Sistemi Orientati agli Aspetti

Tesi di Laurea Specialistica in Ingegneria Informatica

Giordano Battilana

Relatore: Prof. Sergio Congiu

Università di Padova

Padova, 2010

Virtual Machines for Aspect-Oriented Systems

Master Thesis

Giordano Battilana
s060675

IMM

Technical University of Denmark

Kongens Lyngby 2008

Preface

This report constitutes my Master of Science Thesis, written in the period between February 1st, 2008 and September 1st, 2008, in the Language-Based Technology group of the Computer Science and Engineering section at the Informatics and Mathematical Modeling department of the Technical University of Denmark. My supervisor has been Associate Professor Christian W. Probst.

I would like to thank the people that made it possible for me to face this challenge.
I sincerely thank my supervisor Christian W. Probst for being always available whenever I needed guidance, for finding the time to answer my questions and for supporting me in the making of my thesis.
Special thanks to my friend and fellow student Risto Gligorov for the frequent exchanges of opinions about our projects and for the infinite number of fruitful discussions that we had during these months.
Thanks to all the members of the Language-Based Technology group for the Wednesday meetings and for providing an always warm and motivating environment.
Thanks to Emilie, Rajesh, Michal, Birgir, Lukasz and Marco for the coffees, the meals and your valuable company.

I cannot thank enough my family, my father Gianfranco, my mother Paola, Marco and Matteo, for their continuous support and affection. *Thank you.*

Copenhagen, September 1st, 2008

Giordano Battilana

Abstract

Aspect Oriented Programming is a programming paradigm that allows separating frequently used functionalities (concerns) from the application logic, de facto enhancing the modularization of the code. Aspects are generally woven into the code at compilation time and thereafter left untouched. If an aspect is modified, a re-compilation is required in order to propagate the change into the code. In a scenario where aspects are used to dynamically change the behavior of an application according to environmental conditions, this is a relevant limitation.
In recent years, virtual-machines-based solutions for the dynamic weaving of aspects have started to gain popularity [1, 2, 3].

This thesis presents the design and implementation of the AspectK virtual machine, a virtual machine that supports the dynamic weaving and unweaving of aspects while a program is in execution. AspectK [4] is a coordination language that natively supports the Aspect-Oriented Programming paradigm.

The dynamic weaving of aspects in the code is performed by a special component of the virtual machine, the *weaver*. The weaver supports three different strategies for weaving the aspects, each of them targeted to a different type of workload. The thesis presents the details of such strategies and the implications that their implementation have on the overall design of the virtual machine.

Contents

Chapter 1

Introduction

Software systems occupy an important role in our everyday life.

Since their first steps in our society some decades ago, software systems have evolved a lot, moving from being tools exclusive to the academia to efficient assistants now familiar to the masses.
The extraordinary diffusion of electronic devices such as mobile phones and personal computers has certainly improved the quality of our lives but on the other hand it has brought under the spotlight a new major concern, namely security.

Everyday, computers copy, move and transform an incredibly large amount of confidential data, exposing it to the risk of malicious or even accidental access from unauthorized parties. Enforcing an adequate level of protection against such threats is one of the primary concerns for software engineers and system administrators.
Unfortunately, designing a secure system is a true challenge and over the years companies and other organizations devoted significant amounts of capital and men-power to improve the level of security of their software products. Furthermore, the level of complexity of software systems is growing at a fast pace and often the technologies used to enforce security seem to be scarcely effective and sometimes even inadequate.

An explanation to this problem could be that some of the technologies that are still used for designing and developing software were created at a time when Internet and software security were at the embryonal stages and few people could possibly foresee their future worldwide adoption. In the last two decades a lot of research has been conduct to try to improve this situation and new solutions have quickly gained popularity.

The employment of *virtualization* in the field of security is one first example. Virtual machines have been around since the 60s, when the very first steps into this field were moved. It is only recently, however, with the release of the Java virtual machine [5], that virtualization has known large diffusion as a technology used to enforce security.
Originally, virtual machines were developed with the intention to overcome the limitations in the design of the hardware and permit the simultaneous execution of multiple instances of an operating system at the same time.
Popek and Goldberg [6] define the *virtual machine* as an *efficient, isolated duplicate of the real machine.* Hence, a virtual machine can be turned into a *sandbox* where any type of untrusted code can be safely executed: if any disruptive/malicious behavior of the program arises, its effects are confined within the virtual environment.
Secondly, a virtual machine can be used to closely follow the execution of the *guest* code confined in the virtual environment, hence it can work as a *reference monitor* [7] that can check every instruction against a security policy.

Besides virtualization, research in software engineering has recently [8] produced another interesting technology, namely the *Aspect Oriented Programming* paradigm (also known as AOP).

Aspect Oriented Programming, similarly to virtualization, did originally not target security, but tried to solve a very common programming issue.
The Object-Oriented programming paradigm enables the software engineer to design software in a more rational and effective way than what the procedural programming paradigm allows to. Unfortunately there are some programming problems (concerns) for which even Object-Oriented programming seems to be inadequate. The common property that is at the basis of these problems is that they *cross-cut* the system architecture and it is not sufficiently easy or even possible to separate these aspects of the system into confined modules.
The consequence of this fact is that the code belonging to these concerns is *scattered* through the entire system, affecting significantly its maintainability.
Examples of these cross-cutting concerns are debugging or logging.
The goal of Aspect Oriented Programming is to provide a programming technique that allows the separation of these cross-cutting concerns into confined modules called *aspects*.

The employment of AOP in the field of security is obvious when one realizes that the code that enforces security is usually scattered through the entire system and it fits perfectly the definition of *cross-cutting concern*. AOP can be used to separate the security concern from the rest of the system, making it possible to treat it as a confined module. There are several examples ([4] and [9] are two of them) on how researchers are trying to exploit AOP in this sense.

A close look to virtualization and AOP highlights the possibility of combining them to join their strengths.
The point of contact is represented by the following consideration. AOP allows to separate the cross-cutting concerns of the system into separate modules, the aspects. Usually, the aspects are then *merged* (woven) into the rest of the system at compile time and thereafter left untouched. From this point of view, we can state that aspects are *statically* merged into the code, since they are never changed after the compilation. The problem with this approach emerges when security itself is considered as an aspect. Security policies are generally expected to change over time, and if we are willing to enforce it by mean of aspects we must re-compile the program each time the security policy changes. Clearly, for large applications, this could be a non-viable solution and other alternatives should be considered. The idea is to somehow *weave* the new aspects within the application without requiring it's complete recompilation.
Virtual machines enter the scene at this point. Because of their nature, virtual machines can closely follow the execution of a program, thus they are the perfect tool for "manipulating" the program itself or change some parts of its execution at run time possibly requiring the re-compilation of a minor portion of the code.

1.1 Thesis Objectives

This thesis presents the design and the implementation of the AspectK virtual machine, a virtual machine capable of supporting the weaving of aspects at run time. The language supported by the virtual machine is AspectK [4], a coordination language developed at the Informatics and Mathematical Modeling department of the Technical University of Denmark. AspectK is the language of choice for at least two reasons.
AspectK has native support for Aspect-Oriented Programming, hence the effort could be focused on creating the virtual machine rather than on extending some existing language or even designing a new one.
Secondly, AspectK is simple enough to make this project feasible in the given amount of time.

The work carried out consists of the following parts:

- Design and implementation of the virtual machine.
- Design and implementation of a compiler that compiles an AspectK net into the virtual machine's bytecode.
- Design and implementation of the apparatus (the weaver) that enables the dynamic weaving of aspects.
- Analysis of the properties of AspectK and subsequent implementation of three different techniques for the dynamic weaving of code.

1.2 Thesis Outline

The thesis consists of seven chapters and the introduction is the first of them.

In Chapter 2 we present the AspectK language and how it provides support for Aspect-Oriented Programming.
Chapter 3 provides some basic background on virtual machines. The concepts presented here are useful for understanding the architecture of the AspectK virtual machine.
Chapter 4 outlines a prototype of the virtual machine, the *AspectK Interpreter* that was developed before the real virtual machine. This small sub-project was very useful for gaining some knowledge on what the major challenges to take into account in the design of the virtual machine were. Moreover it gave me the chance to start brainstorming some of the infrastructure that would have eventually make it into the real virtual machine.
Chapter 5 analyzes all the components that form the virtual machine. All the phases of the development are outlined here.
In Chapter 6 we analyze what could be the major challenges to take into consideration if the virtual machine had to be extended to support more process calculi besides AspectK.
Finally, Chapter 7 draws some conclusive comments on the outcome of this project and presents some possible ideas for further improving and extending the work presented in this thesis.

Appendix A contains an example of the code generated in the compilation of an AspectK net.
Appendix B gives an overview on the software included with this thesis, namely the AspectK compiler *aspectkc*, the AspectK GUI *aspectKgui* and the AspectK Interpreter.
Appendix C contains the full specification of the bytecode instructions.

Chapter 2

The AspectK Language

This chapter presents *AspectK* [4], a coordination language that supports Aspect-Oriented Programming.
AspectK is based on a fragment of the KLAIM [10, 11, 12] language.

Section 2.1 presents the KLAIM kernel language which is at the core of AspectK, whereas Section 2.2 outlines the Aspect-Oriented part of AspectK. Finally, Section 2.3 provides an overview on how the specification of this language could be modified to support different approaches for dynamically weaving code at run time.

2.1 KLAIM

The syntax of the fragment of KLAIM incorporated in AspectK is shown in Figure 2.1.

A net N is either a parallel composition of located processes or located tuples. The components of tuples can be location constants only. Nets must be *closed*, meaning that all the variables must be in scope of a defining occurrence.

$$
\begin{array}{llll}
N \in \mathsf{Net} & N & ::= & N_1 || N_2 \mid l :: P \mid l :: \langle \overrightarrow{l} \rangle \\
P \in \mathsf{Proc} & P & ::= & P_1 | P_2 \mid \sum_i a_i.P_i \mid *P \\
a \in \mathsf{Act} & a & ::= & \mathbf{out}(\overrightarrow{\ell})@\ell \mid \mathbf{in}(\overrightarrow{\ell})@\ell \mid \mathbf{read}(\overrightarrow{\ell})@\ell \\
\ell, \ell^{\lambda} \in \mathsf{Loc} & \ell & ::= & u \mid l \qquad\qquad \ell^{\lambda} ::= \ell \mid !u
\end{array}
$$

Figure 2.1: KLAIM Nets and Processes syntax

A process P is either a parallel composition of processes, a guarded sum of action-prefixed processes or a replicating process (prefixed by *). The guarded sum $\sum_i a_i.P_i$ is written 0 if the index set is empty.

A tuple can be output (*out* action), input (*in* action) or read (*read* action) from a location. If a tuple is read, it is not removed from the location.

The parameters can be location constants l, defining occurrences of location variables $!u$ and applied occurrences of a location variable u.
The scope of a defining occurrence is the entire process to the right of the occurrence.

An example of Net is provided in Figure 2.2. This example describes a restaurant environment.

The location *IngrDB* stores a record for each of the available groceries, whereas the *RecipeDB* location stores a record for each recipe that the cooks are able to prepare.
The two waiter processes *WaiterAlfredo* and *WaitressLaura* are in charge of gathering the orders from the customers and outputting them at the *Board* location. Specifically, WaiterAlfredo is in charge of the pasta-based dishes, whereas WaiterLaura is in charge of handling the pizza orders and the desserts orders. Both the waiters also deliver the meal to the customers when it is ready, which is achieved by outputting the meal at the *Customer* location.
The process *CookLuigi* inputs the order from the board, reads the corresponding recipe from the database and collects the required ingredients. The selected ingredients are removed from the database once CookLuigi has used them. Eventually, when the meal is ready, it is output at the location whose name is the "category" of the dish and the waiter in charge of that category can deliver the dish to the hungry customer.
The *GroceryProvider* process makes sure that the restaurant is never out of ingredients.

```
   IngrDB :: <Spaghetti>
|| IngrDB :: <Tomato>
|| IngrDB :: <Eggs>
|| IngrDB :: <PecorinoCheese>
|| IngrDB :: <ParmigianoCheese>
|| IngrDB :: <Bacon>

|| RecipeDB :: <Pasta,Pesto,Spaghetti,Basil,Oil,ParmigianoCheese>
|| RecipeDB :: <Pasta,Carbonara,Spaghetti,Eggs,PecorinoCheese,Bacon>
|| RecipeDB :: <Pizza,Margherita,Pasta,Tomato,Mozzarella,Basil>
|| RecipeDB :: <Cake,Tiramisu,Biscuits,Eggs,Sugar,Coffee>

|| CookLuigi :: *in(!cathegory,!name)@Board.
                 read(cathegory,name,!ingr1,!ingr2,!ingr3,!ingr4)@RecipeDB.
                 in(ingr1)@IngrDB.
                 in(ingr2)@IngrDB.
                 in(ingr3)@IngrDB.
                 in(ingr4)@IngrDB.
                 out(name)@cathegory.0

|| WaiterAlfredo :: *(     out(Pasta,Pesto)@Board.0
                         + out(Pasta,Carbonara)@Board.0
                       | in(!meal)@Pasta.out(meal)@Customer.0 )

|| WaitressLaura :: *(   out(Cake,Tiramisu)@Board.
                         in(!meal)@Cake.out(meal)@Customer.0
                       + out(Pizza,Margherita)@Board.
                         in(!meal)@Pizza.out(meal)@Customer.0)

|| GroceryProvider :: *(   out(ParmigianoCheese)@IngrDB.0
                          | out(PecorinoCheese)@IngrDB.0
                          | out(Coffee)@IngrDB.0
                          | out(Mozzarella)@IngrDB.0 | out(Pasta)@IngrDB.0
                          | out(Spaghetti)@IngrDB.0 | out(Tomato)@IngrDB.0
                          | out(Basil)@IngrDB.0 | out(Eggs)@IngrDB.0
                          | out(Bacon)@IngrDB.0 | out(Oil)@IngrDB.0
                          | out(Biscuits)@IngrDB.0 | out(Sugar)@IngrDB.0 )
```

Figure 2.2: An example of Net

Well-formedness of Locations and Actions. To express the well-formedness of locations we need to introduce two functions:

- *bv*, for calculating the bound variables of the various types of location that may occur in actions, i.e. $bv(l, u, !v) = v$;
- *fv*, for calculating the free variables of the various types of location that may occur in actions, i.e. $fv(l, u, !v) = u$.

An input action is well formed if its sequence $\overrightarrow{\ell^{\lambda}} = \ell_1, \cdots, \ell_k$ of locations is well formed. A sequence of locations is well formed if the following two conditions are fulfilled.

$$\forall i, j \in \{1, \cdots, k\} : i \neq j \Rightarrow bv(\ell_i^{\lambda}) \cap bv(\ell_j^{\lambda}) = \emptyset$$
$$bv(\overrightarrow{\ell^{\lambda}}) \cap fv(\overrightarrow{\ell^{\lambda}}) = \emptyset$$

These conditions pose the following constraints:

- no multiple defining occurrences of the same variable in an action;
- in a single action, bound variables and free variables cannot share any name.

2.1.1 Semantics

Figure 2.3 outlines the rules of structural congruence of nets, whereas Figure 2.4 illustrates the one-step reduction relation on nets.

$$l :: P_1 \mid P_2 \;\equiv\; l :: P_1 \parallel l :: P_2 \qquad l :: *P \;\equiv\; l :: P \mid *P$$

$$\frac{N_1 \equiv N_2}{N \parallel N_1 \;\equiv\; N \parallel N_2}$$

Figure 2.3: KLAIM structural congruence

Figure 2.4 introduces the function *match*, which is shown in Figure 2.5.

$$l_s :: \mathbf{out}(\overrightarrow{l})@l_0.P + \cdots \rightarrow l_s :: P \;||\; l_0 :: \langle \overrightarrow{l} \rangle$$

$$l_s :: \mathbf{in}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots \;||\; l_0 :: \langle \overrightarrow{l} \rangle \rightarrow l_s :: P\theta \qquad \text{if } match\left(\overrightarrow{\ell^\lambda}; \overrightarrow{l}\right) = \theta$$

$$l_s :: \mathbf{read}(\overrightarrow{\ell^\lambda})@l_0.P + \cdots \;||\; l_0 :: \langle \overrightarrow{l} \rangle \rightarrow l_s :: P\theta \;||\; l_0 :: \langle \overrightarrow{l} \rangle \quad \text{if } match\left(\overrightarrow{\ell^\lambda}; \overrightarrow{l}\right) = \theta$$

$$\frac{N_1 \rightarrow N_1'}{N_1 \;||\; N_2 \;\rightarrow\; N_1' \;||\; N_2} \qquad \frac{N \equiv N' \quad N' \rightarrow N'' \quad N'' \equiv N'''}{N \rightarrow N'''}$$

Figure 2.4: KLAIM reaction semantics (on closed nets)

$$match(\langle\rangle, \langle\rangle) = id$$

$$match(\langle \ell_1'^{\lambda}, \cdots, \ell_k'^{\lambda} \rangle; \langle l_1, \cdots, l_k \rangle) = \text{let } \theta = \text{case } \ell_1'^{\lambda} \text{ of}$$
$$l_1' : \quad \mathbf{if}\ l_1' = l_1\ \mathbf{then}\ id\ \mathbf{else}\ fail$$
$$!u : \quad [l_1/u]$$
$$\text{in } \theta \;\circ\; match(\langle \ell_2'^{\lambda}, \cdots, \ell_k'^{\lambda} \rangle; \langle l_2, \cdots, l_k \rangle)$$

Figure 2.5: KLAIM pattern matching of templates against tuples

2.2 AspectK

This section presents the syntax and the semantics of AspectK, a language that extends the portion of KLAIM presented in the previous section to incorporate Aspect-Oriented Programming features.
But before going into the internals of the language we present the core principles of the Aspect-Oriented Programming paradigm. These concepts are necessary to understand what are the possible ways to support AspectK and the dynamic weaving in the virtual machine.

2.2.1 Aspect-Oriented Programming

Methodologies such as Object-Oriented Programming proved to be satisfactory in capturing the essence of *core* concerns of software systems but they were not that effective in treating *cross-cutting* concerns. Cross-cutting concerns are aspects of a program which affect other concerns.
Aspect-Oriented Programming [8] (also AOP) was designed with the goal of providing means for separating the cross-cutting concerns from the core concerns and confining them into stand-alone modules.

AOP is a rather new technology, as a matter of fact the term itself was coined by Gregor Kiczales in 1996. He and Cristina Lopes, of the Palo Alto Research Center (subsidiary of Xerox Corp.), were among the early contributors of AOP [13].

Gregor Kiczales led the team at Xerox that created AspectJ, one of the first implementations of AOP. Since then, a lot of work on this new technology was carried out by universities all around the world and this resulted into the proliferation of AOP-dialects for a large variety of languages, including the well-known C/C++, Python, C# and Perl.

A close look on the AOP methodology reveals the presence of three major phases [13]:

Aspectual Decomposition. In this phase the design of the system is analyzed and two categories of concerns are identified:

1. *Core* concerns, namely the components that define the core business logic of a software system.
2. *Cross-cutting* concerns, namely aspects of the system that affect many other components. Traditional examples are logging, security and debugging.

Concern Implementation. The two categories of concerns previously identified are implemented with the appropriate methodologies:

- the *core* concerns using a traditional methodology such as Object-Oriented Programming;
- the *cross-cutting* concerns using Aspect-Oriented Programming.

Aspectual Re-composition. This process, also known as *weaving* or *integrating* consists in specifying *re-composition* rules by creating *modularization units* also called *aspects*. Each aspect is characterized by the following structures:

- The *pointcut*, namely a construct that selects identifiable points in the execution of a program, called *joinpoints*.
- The *Advice*, which defines the code that must be executed at the *joinpoints* selected by the *pointcuts*.

During the re-composition phase, the aspects are analyzed and whenever a pointcut selects a joinpoint in the code, the code specified in the advice is *injected*.

2.2.1.1 Implementation

The basic AOP language implementation is composed by two logical steps [13]:

1. The individual concerns (core and cross-cutting) are combined using the weaving rules.
2. The resulting information is converted into executable code.

The process that combines the individual concerns according to the weaving rules is called *weaving* and the processor that performs this job is called *weaver*. Figure 2.6 illustrates how the traditional process of compilation (Figure 2.6a) is changed when the AOP weaver is incorporated in the compiler (Figure 2.6b).

The method presented in Figure 2.6b describes the *compile-time* weaving and it suggests how, in the basic implementation of AOP, the weaving of aspects is a one-time process which modifies in a definitive way the program. We will explore, instead, dynamic weaving techniques, whose aims are:

- enable the weaving of code in the program *after* the compilation.
- enable the unweaving of previously woven code.

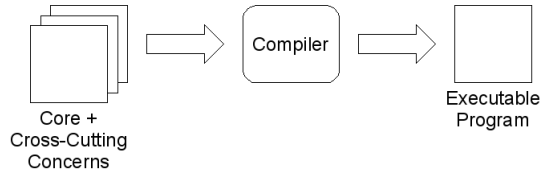
2.2.1.2 Some Terminology

This section summarizes some terminology specific to the Aspect-Oriented Programming paradigm.

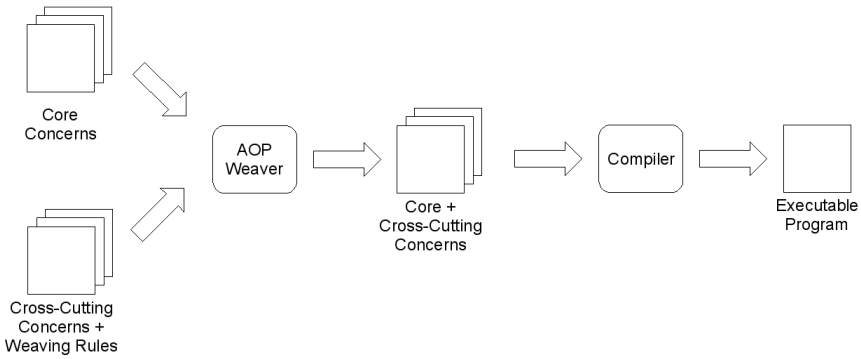
Compile-time Weaving. The operation of weaving performed on the program at compile-time.

Dynamic Weaving. The operation of weaving performed on the program at run-time.

Just-In-Time Weaving. The operation of weaving performed at run-time on the portion of code that is about to be executed.



(a) Traditional compilation



(b) Compilation involving the AOP weaver

Figure 2.6: Comparison of the traditional compilation and the compilation involving the AOP weaver

Join Point. The *join point* is an identifiable point in the execution of a program.

Pointcut or Cut. The *pointcut* is a program construct that selects join points an collect context at those points.

Advice. The *advice* is the code to be executed at a join point that has been selected by a pointcut.

Aspect. The *aspect* is the central unit of AOP. Pointcuts and advices are combined in an aspect.

2.2.2 AspectK Syntax

The syntax of AspectK, which is shown in Figure 2.7, extends the syntax of KLAIM (Figure 2.1).

$S \in \mathsf{System}$	S	::=	$\mathbf{let}\ \overrightarrow{asp}\ \mathbf{in}\ N$
$asp \in \mathsf{Asp}$	asp	::=	$A[cut] \stackrel{\triangle}{=} body$
$body \in \mathsf{Advice}$	$body$	::=	$\mathbf{case}(cond)\ sbody;\ body \mid sbody$
	$sbody$	::=	$as\ \mathbf{break} \mid as\ \mathbf{proceed}\ as$
$as \in \mathsf{Act}^*$	as	::=	$a.as \mid \epsilon$
$cond \in \mathsf{BExp}$	$cond$	::=	$\mathbf{test}(\overrightarrow{\ell^\lambda})@\ell \mid \ell_1 = \ell_2 \mid cond_1 \wedge cond_2 \mid \neg\ cond$
$cut \in \mathsf{Cut}$	cut	::=	$\ell :: a$
$\ell^\lambda \in \mathsf{Loc}$	ℓ^λ	::=	$\ell \mid !u \mid ?u$

Figure 2.7: AspectK Syntax

AspectK introduces some new constructs, outlined below.

Aspects. AspectK introduces the notion of *System*, which is a *Net* prefixed by a sequence of *Aspect* declarations.

An aspect declaration takes the form of $A[cut] \stackrel{\triangle}{=} body$ where:
- A is the name of the aspect;
- cut is the action to be trapped by A (the pointcut);
- $body$ specifies the advice that handles the trapped action.

Cut. A *cut* is formed by a *cut action* located by a location ℓ.

The definition of ℓ^λ is extended to incorporate the new location expression $?u$, which traps $!u$ and l occurring in actions. The definition of the *check* function (Figure 2.12) clarifies the semantics of $?u$. If *?u* or *!u* is used in a cut pattern, u should only occur in *after* (or post) actions. u is not allowed in neither before actions or conditionals.

Well-formedness of Cuts. In addition to the conditions of well-formedness of KLAIM a new condition is added to express the well-formedness of cuts.
Given the function *cl(cut)* that generates the list of entities involved in a cut, for example:

$$cl(l_s :: \mathbf{in}(!x, y, ?z)@l_0) = \langle l_s, x, y, z, l_0 \rangle$$

the well-formedness condition requires that the variables returned by *cl(cut)* are pairwise distinct.

Advice. In the definition of an *advice* the following rules hold:

- The keyword **break** indicates that the original action is suppressed and the process is prevented from further execution.
- The keyword **proceed** allows the original action to be executed. In case of multiple aspects trapping the action, **break** takes precedence over **proceed**.
- In case of multiple aspects trapping an action, all the before actions (or pre-actions) are executed; then, if no **break** applied, the original action is executed and finally the after actions (or post-actions), in reverse order of declaration.

Condition. A condition *cond* is equivalent to a standard boolean expression. The primitive $test(\overrightarrow{\ell^{\lambda}})@\ell$ succeeds if there is a tuple at location ℓ that matches $\overrightarrow{\ell^{\lambda}}$. *?u* is not allowed within *test* conditions.

AspectK Example

An example of the usage of the syntax of AspectK is outlined in Figure 2.8, which reports four aspects defined for the net of Figure 2.2.

The first aspect limits the activity of the GroceryProvider process: the aspect prevents the delivery of ingredients that are already available at the IngrDB location. The second aspect is activated when Pasta is not served at the restaurant, it stops the WaiterAlfredo process whenever the action *out(Pasta,x)@Board* is going to be fired. The third aspect is used to log the type of dishes prepared by process CookAlfredo. This is achieved by inserting a post-action that outputs at a special location *Notes* the category and the name of the dish.
Finally, the fourth aspect enhances the quality of service at the restaurant. Whenever one of the waiters is about to serve the meal to the customer (*out(meal)@Customer*), the aspect inserts a pre-action that takes care of placing a fork and a knife at the table of the customer (*out(Fork,Knife)@Customer*) and two post-actions that remove the dirty fork and knife and the dirty dish from the Customer location and output them at the *WashingMachine* location.

```
let
   A[$GroceryProvider :: out(x)@$IngrDB ] ^=
       case( test(x)@$IngrDB )
           break;
           proceed,

   A[$WaiterAlfredo :: out($Pasta,x)@$Board ] ^=
       break,

   A[$CookLuigi :: in(!cat,!name)@$Board ] ^=
       proceed out(cat,name)@Notes,

   A[waiter :: out(meal)@$Customer] ^=
       out(Fork,Knife)@Customer
       proceed
       in(!dirtyFork,!dirtyKnife)@Customer.in(!dirtyDishes)@Customer.
           out(dirtyFork,dirtyKnife,dirtyDishes)@WashingMachine
in

[...]
```

Figure 2.8: Specification of aspects for the Net of Figure 2.2

2.2.3 AspectK Semantics

Figure 2.9 illustrates the one-step reduction rules that define the semantics of AspectK. Rules from figures 2.4 and 2.3 should be also taken into account. The rules for the actions come in pair.
The first rule defines the semantics for the action when no advice is allowed to interrupt it (syntactically, this is defined as *underlining*).
The second rule makes use of the function Φ defined in Figure 2.10.
The result of $\Phi_f(\Gamma_A; \ell :: a)$ is a sequence of actions that trap $\ell :: a$. Γ_A is the global set of aspects.
f can be either **break** or **proceed**. The former case arises if at least one "**break**" advice applies, otherwise f will be **proceed**. If f is **proceed**, the action $\underline{a}$ is eventually emitted, otherwise it is replaced by **stop**

Figure 2.11 shows the auxiliary function *trap* used by Φ, whilst Figure 2.12 presents the function *check*, which is used by *trap* to determine whether a cut matches an action.
If so, *check* produces a list of substitutions for the variables occurring in the cut.

Figure 2.13 outlines the $k_f^{\Gamma_A, \ell::a}$ function, which processes the advice associated

$$\frac{N_1 \to N_1' \ \text{(where globally } \Gamma_{\mathrm{A}} = \overrightarrow{asp})}{\textbf{let } \overrightarrow{asp} \textbf{ in } N \ \to \ \textbf{let } \overrightarrow{asp} \textbf{ in } N'}$$

$$l_s :: \underline{\textbf{stop}}.P + \cdots \to l_s :: 0$$
$$l_s :: \underline{\textbf{out}}(\overrightarrow{l})@l_0.P + \cdots \to l_s :: P \mid\mid l_0 :: \langle \overrightarrow{l} \rangle$$
$$l_s :: \underline{\textbf{in}}(\overrightarrow{\ell^{\lambda}})@l_0.P + \cdots \mid\mid l_0 :: \langle \overrightarrow{l} \rangle \to l_s :: P\theta \qquad \text{if } match\left(\overrightarrow{\ell^{\lambda}}; \overrightarrow{l}\right) = \theta$$
$$l_s :: \underline{\textbf{read}}(\overrightarrow{\ell^{\lambda}})@l_0.P + \cdots \mid\mid l_0 :: \langle \overrightarrow{l} \rangle \to l_s :: P\theta \mid\mid l_0 :: \langle \overrightarrow{l} \rangle \quad \text{if } match\left(\overrightarrow{\ell^{\lambda}}; \overrightarrow{l}\right) = \theta$$

$$\frac{l_s :: \Phi_{proceed}(\Gamma_A; l_s :: \textbf{out}(\overrightarrow{l})@l_0).P \to N}{l_s :: \textbf{out}(\overrightarrow{l})@l_0.P + \cdots \to N}$$
$$\frac{l_s :: \Phi_{proceed}(\Gamma_A; l_s :: \textbf{in}(\overrightarrow{\ell^{\lambda}})@l_0).P \mid\mid N' \to N}{l_s :: \textbf{out}(\overrightarrow{\ell^{\lambda}})@l_0.P + \cdots \mid\mid N' \to N}$$
$$\frac{l_s :: \Phi_{proceed}(\Gamma_A; l_s :: \textbf{read}(\overrightarrow{\ell^{\lambda}})@l_0).P \mid\mid N' \to N}{l_s :: \textbf{read}(\overrightarrow{\ell^{\lambda}})@l_0.P + \cdots \mid\mid N' \to N}$$

Figure 2.9: Reaction semantics (on closed nets)

to a matching cut. $k_f^{\Gamma_A, \ell::a}$ makes use of the auxiliary function B which is reported in Figure 2.14.

2.3 Dynamic Weaving in AspectK

The semantics of AspectK defines a *just-in-time* weaving strategy.

The *trap* (Figure 2.11) operation is attempted at run-time, whenever an action is

$$\begin{aligned}
\Phi_f(A[cut] \triangleq body; \Gamma_A; \ell :: a) \quad &= \quad \text{case } trap(cut, \ell :: a) \text{ of fail} : \ \Phi_f(\Gamma_A; \ell :: a) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \theta : \ k_f^{\Gamma_A, \ell::a}(body\ \theta) \\
\Phi_f(\varepsilon; \ell :: a) \quad &= \quad \text{case } f \text{ of } \textbf{proceed} : \underline{a} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{break} : \underline{stop}
\end{aligned}$$

Figure 2.10: Trapping Aspects: Step 1

$$
\begin{array}{l}
trap(cut, \ell :: a) = \text{case } (cut, \ell :: a) \text{ of} \\
\qquad (\ell_s :: \mathbf{out}(\overrightarrow{\ell})@\ell_0, l_s :: \mathbf{out}(\overrightarrow{l})@l_0) \quad : check(\langle \ell_s, \overrightarrow{\ell}, \ell_0\rangle, \langle l_s, \overrightarrow{l}, l_0\rangle) \\
\qquad (\ell_s :: \mathbf{in}(\overrightarrow{\ell^\lambda})@\ell_0, l_s :: \mathbf{in}(\overrightarrow{\ell'^\lambda})@l_0) \quad : check(\langle \ell_s, \overrightarrow{\ell^\lambda}, \ell_0\rangle, \langle l_s, \overrightarrow{\ell'^\lambda}, l_0\rangle) \\
\qquad (\ell_s :: \mathbf{read}(\overrightarrow{\ell^\lambda})@\ell_0, l_s :: \mathbf{read}(\overrightarrow{\ell'^\lambda})@l_0) : check(\langle \ell_s, \overrightarrow{\ell^\lambda}, \ell_0\rangle, \langle l_s, \overrightarrow{\ell'^\lambda}, l_0\rangle) \\
\qquad \text{otherwise fail}
\end{array}
$$

Figure 2.11: Trapping Aspects: Step 2

$$
\begin{array}{l}
check(\langle\rangle, \langle\rangle) = id \\
check(\langle \ell_1^\lambda, \ell_2^\lambda, \cdots, \ell_k^\lambda\rangle, \langle \ell_1'^\lambda, \cdots, \ell_k'^\lambda\rangle) = \text{let } \theta = \text{case } (\ell_1^\lambda, \ell_1'^\lambda) \text{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad (!u, !u') : [u'/u] \\
\qquad\qquad\qquad\qquad\qquad\qquad (?u, !u') : [u'/u] \\
\qquad\qquad\qquad\qquad\qquad\qquad (?u, l') : [l'/u] \\
\qquad\qquad\qquad\qquad\qquad\qquad (u, l') : [l'/u] \\
\qquad\qquad\qquad\qquad\qquad\qquad (l, l') : \mathbf{if}\ l = l'\ \mathbf{then}\ id\ \mathbf{else}\ fail \\
\qquad\qquad\qquad\qquad\qquad\quad \text{otherwise fail} \\
\qquad\qquad\qquad\qquad\quad \text{in } \theta \circ check(\langle \ell_2^\lambda, \cdots, \ell_k^\lambda\rangle, \langle \ell_2'^\lambda, \cdots, \ell_k'^\lambda\rangle)
\end{array}
$$

Figure 2.12: Trapping Aspects: Step 3

encountered in the execution path. *Trap* uses the *check* (Figure 2.12) function, which matches the template defined in the *cut* against the trapped action.

In the implementation of the interpreter first and the virtual machine then, I decided to explore the possibility of adapting the AspectK semantics to support a *dynamic* weaving approach.
Both the dynamic weaving and the JIT weaving are performed at run time, but the advantage of using dynamic weaving is that the injection of code is performed *once* (for the successfully trapped actions) and it is not needed to

$$
\begin{array}{l}
k_f^{\Gamma_A, \ell::a}(\mathbf{case}\ cond\ sbody; body) = \text{case } B(cond) \text{ of } \mathbf{tt} : k_f^{\Gamma_A, \ell::a}(sbody) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{ff} : k_f^{\Gamma_A, \ell::a}(body) \\
k_f^{\Gamma_A, \ell::a}(sbody) = \text{case } sbody \text{ of } as_1\ \mathbf{proceed}\ as_2 : as_1.\Phi_f(\Gamma_A; \ell :: a).as_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad as\ \mathbf{break} : as.\Phi_{break}(\Gamma_A; \ell :: a)
\end{array}
$$

Figure 2.13: Trapping Aspects: Step 4

$$
\begin{array}{lcl}
B(test(\overrightarrow{\ell^{\lambda}})@l) & = & \begin{cases} \textbf{tt} \text{ if there exists a tuple } \overrightarrow{l} \text{ at location l} \\ \quad \text{such that } match(\overrightarrow{\ell^{\lambda}}, \overrightarrow{l}) \neq fail \\ \textbf{ff} \text{ otherwise} \end{cases} \\
B(l_1 = l_2) & = & \begin{cases} \textbf{tt} \text{ if } l_1 = l_2 \\ \quad \text{such that } match(\overrightarrow{\ell^{\lambda}}, \overrightarrow{l}) \neq fail \\ \textbf{ff} \text{ otherwise} \end{cases} \\
B(cond_1 \wedge cond_2) & = & \begin{cases} \textbf{tt} \text{ if } B(cond_1) = \textbf{tt} \text{ and } B(cond_2) = \textbf{tt} \\ \textbf{ff} \text{ if } B(cond_1) = \textbf{ff} \text{ or } B(cond_2) = \textbf{ff} \end{cases} \\
B(\neg cond) & = & \begin{cases} \textbf{tt} \text{ if } B(cond) = \textbf{ff} \\ \textbf{ff} \text{ if } B(cond) = \textbf{tt} \end{cases}
\end{array}
$$

Figure 2.14: Trapping Aspects: Step 5

attempt it anymore, at least as long as the set of aspects Γ_A does not change.

The dynamic weaving can be successfully performed whenever the operation of trapping does not require knowledge of run time context: if we observe the AspectK semantics there are two places were run time information is needed while performing a trap:

1. within the *check* function. When a constant location of the cut template is matched against a *variable* of the trapped action it is necessary to retrieve the value of such variable to perform the comparison of the two values.
2. within the B function, whenever a *test* or an *equality (=)* condition is evaluated. In general, according to the semantics of AspectK, conditions must be evaluated when the trapped action is about to be executed and cannot be evaluated at a different time.

In order to support dynamic weaving we have to deal with both the two cases just described. The semantics of AspectK specifies that the trapping is to be performed at run time when the action is about to be fired. This implies that the *check* function is specified so that it matches only against either constants l or variable definitions $!u$. The match against variable occurrences, which arises if the trapping is attempted at dynamic time, is not considered.

Figure 2.15 illustrates a prototype of the *check* function extended to support dynamic weaving. The modifications are highlighted in bold.
In the two cases $(?u, u')$ and (u, u') the trapped location is substituted to the template location.

$$
\begin{aligned}
&check(\langle\rangle, \langle\rangle) = id \\
&check(\langle \ell_1^{\lambda}, \ell_2^{\lambda}, \cdots, \ell_k^{\lambda}\rangle, \langle \ell_1'^{\lambda}, \cdots, \ell_k'^{\lambda}\rangle) = \text{let } \theta = \text{case } (\ell_1^{\lambda}, \ell_1'^{\lambda}) \text{ of} \\
&\qquad\qquad (!u, !u') : [u'/u] \\
&\qquad\qquad (?u, !u') : [u'/u] \\
&\qquad\qquad (?\mathbf{u}, \mathbf{u'}) : [\mathbf{u'}/\mathbf{u}] \\
&\qquad\qquad (?u, l') : [l'/u] \\
&\qquad\qquad (\mathbf{u}, \mathbf{u'}) : [\mathbf{u'}/\mathbf{u}] \\
&\qquad\qquad (u, l') : [l'/u] \\
&\qquad\qquad (\mathbf{l}, \mathbf{u'}) : \textbf{exception} \\
&\qquad\qquad (l, l') : \textbf{if } l = l' \textbf{ then } id \textbf{ else } fail \\
&\qquad\qquad \text{otherwise fail} \\
&\qquad \text{in } \theta \circ check(\langle \ell_2^{\lambda}, \cdots, \ell_k^{\lambda}\rangle, \langle \ell_2'^{\lambda}, \cdots, \ell_k'^{\lambda}\rangle)
\end{aligned}
$$

Figure 2.15: The *check* function modified to support dynamic weaving

The case (l, u') is probably the most interesting one. The value of u' cannot be fetched and compared to l because the value of u' is unpredictable and can be retrieved only when the action is being fired. Its value could be one of the following ones:

- The value could be *undefined*, because the variable u was not defined yet (one should remember that dynamic weaving can happen at *any* time);
- The value could be defined but *different* from the one that would be available if the check was executed when the trapped action is being fired.

The solution is to return an exception that signals the necessity of trapping the action following the original AspectK semantics, namely with a JIT weaving strategy.

The B function cannot be modified anyhow. In fact, conditional advices must be evaluated only when the action is about to be executed. This means that dynamic weaving cannot be performed on actions trapped by aspects containing conditional advices.

In conclusion, supporting dynamic weaving in AspectK requires the following changes to the semantics:

- The *check* function is changed to support the substitutions $(?u, u')$ and (u, u') and to handle the (l, u') case.

$$k_f^{\Gamma_A,\ell::a}(\textbf{case } cond\ sbody; body) = \text{exception}$$
$$k_f^{\Gamma_A,\ell::a}(sbody) = \text{case } sbody \text{ of } as_1\ \textbf{proceed}\ as_2 : as_1.\Phi_f(\Gamma_A; \ell :: a).as_2$$
$$as\ \textbf{break} : as.\Phi_{break}(\Gamma_A; \ell :: a)$$

Figure 2.16: Prototype of function $k_f^{\Gamma_A,\ell::a}$ modified for dynamic weaving

- The $k_f^{\Gamma_A,\ell::a}$ function (Figure 2.13) is changed as shown in Figure 2.16.
- The function Φ_f is modified as shown in Figure 2.17.

Some remarks:

- dynamic weaving, in general, *cannot* fully substitute the JIT weaving. It may be considered as an optimization of the JIT weaving, since in those cases where no run-time information is required, the injection of actions is performed only once for the trapped action.
- If the *exception* is raised by any aspect involved in the weaving, the process must be aborted. Continuing the weaving by simply skipping the aspect that threw the exception would be wrong, because that same aspect might successfully trap the action if the trap is executed when the action is being fired.
- The B function is never used in the dynamic weaving, since if a conditional-advice is encountered an exception is returned.

$$\Phi_f(A[cut] \stackrel{\triangle}{=} body; \Gamma_A; \ell :: a) = \text{case } trap(cut, \ell :: a) \text{ of fail} : \Phi_f(\Gamma_A; \ell :: a)$$
$$\theta : k_f^{\Gamma_A,\ell::a}(body\ \theta)$$
$$\text{exception} : \text{abort}$$

$$\Phi_f(\varepsilon; \ell :: a) = \text{case } f \text{ of } \textbf{proceed} : \underline{a}$$
$$\textbf{break} : \underline{stop}$$

Figure 2.17: Prototype of function Φ_f modified for dynamic weaving

Chapter 3

Virtual Machines

> *A virtual machine is taken to be an efficient, isolated duplicate of the real machine*

This citation comes from the work of Popek and Goldberg, "*Formal requirements for virtualizable third generation architectures*", which dates back to 1974 [6]. Virtual machines have evolved a lot since that time but these words still capture the nature of virtual machines.

Formally, the process of virtualization is the construction of an isomorphism [6] that maps a virtual *guest* system to a real *host* (Figure 3.1).

The isomorphism maps the guest state to the host state (function V). Moreover, for a sequence of operations e that modifies the state in the guest (the function e changes the state S_i into the state S_j) there is a corresponding sequence of operations e' that performs an equivalent modification to the host's state (the function e' changes the state S'_i into the state S'_j).

Popek and Goldberg have defined a list of characteristics that any virtual machine should possess. The software layer providing virtualization is referred to as the "Virtual Machine Monitor" (also VMM). The VMM fundamental features are outlined in the sequel:

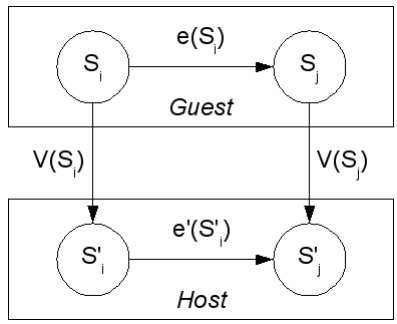


Figure 3.1: Virtualization is the construction of an isomorphism between a guest system and a host application $e' \circ V(S_i) = V \circ e(S_i)$

1. the VMM should provide an *essentially identical* environment to the original machine; this means that an application executed on the virtual machine should exhibit the same behavior as if it was run on the real machine.
2. The VMM should be *efficient*, thus a statistically dominant subset of the virtual processor's intructions should be executed directly by the real processor, with no software intervention by the VMM.
3. The VMM should posses a *complete control over the resources*, namely:
   - a program running in the virtual environment cannot access any resource not explicitly allocated to it.
   - under certain condition the VMM can regain control of resources already allocated.

Given the requirements, we shall now see what the process of virtualization consists of. Smith and Nair [14] describe it as a process characterized by two properties:

1. It maps the virtual resources like registers, memory or files to real resources in the underlying machine.
2. It uses real machine instructions and/or system calls to carry out the actions specified by the virtual machine instructions and/or system calls.

We can identify two classes of virtual machines:

- *System* virtual machines.

- *Process* virtual machines.

Both of these classes satisfy the properties exposed by Popek and Goldberg and implement the process of virtualization described by Smith and Nair. Nevertheless, as we shall see in the following subsections, these two classes of virtual machines are significantly different from several points of view.

3.1 System Virtual Machines

The *System* virtual machines embodies the more "traditional" concept of virtual machines, namely a software that virtualizes a certain set of hardware.

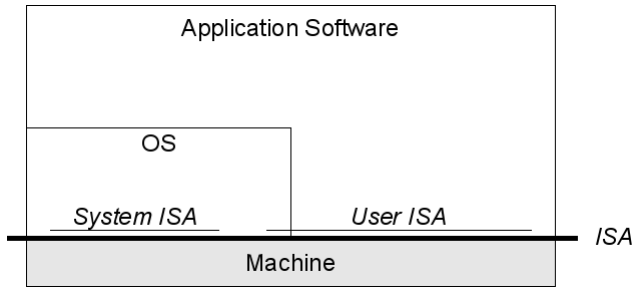


Figure 3.2: Instruction Set Architecture (ISA) interface.

In order to understand what a system virtual machine must virtualize, we should consider the system from the perspective of an operating system. In this case, the "machine" is represented by the the whole hardware platform and the Instruction Set Architecture (ISA) provides the interface between the system and the machine. These concepts are depicted in Figure 3.2.
A system virtual machine can translate the ISA used by one hardware platform to another (Figure 3.3).

There are several types of *system* virtual machines:

Classic virtual machine. The most obvious application of system virtual machines is to virtualize the hardware and enable the simultaneous execution of several guest operating systems. Both the host and the guest ISA are the same.

Hosted virtual machine. This application is equivalent to the previous one, with the difference that the virtual machine runs on top of a host operating system and not on bare hardware.

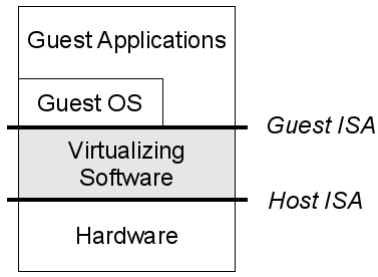


Figure 3.3: The System virtual machine.

Whole-System virtual machine. These virtual machines differ from the classic ones in the fact that the guest system is compiled for a ISA which differs from the host system ISA. Hence, the Whole-System virtual machine *emulates* the guest ISA.

Codesigned virtual machine. This type of virtual machine is usually adopted in conjunction with hardware that supports some kind of innovative ISA and/or hardware implementations for improved efficiency or performance or both. The virtual machine is part of the hardware implementation and is used to enable the extended features that it features.

3.2 Process Virtual Machines

Process virtual machines support the execution of a single process.

In order to understand what the process virtual machine must virtualize, we should consider the system from the perspective of a process. In this case, the "machine" consists of the following components:

- A logical memory address space that was assigned to the process.
- A set of user-registers and user-instructions that permit the execution of code belonging to the process.
- A set of system calls that the process uses to access the I/O system.

In conclusion the "machine" consists of a combination of operating system calls and underlying user-level hardware. The interface that enables the process to interact with the "machine" that we have just defined is called Application

Binary Interface (ABI).
These concepts are depicted in Figure 3.4.

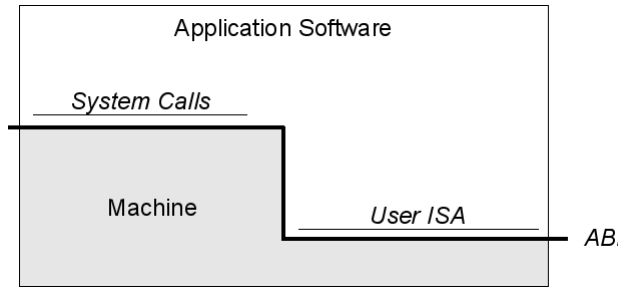


Figure 3.4: Application Binary Interface.

A process virtual machine implements the guest process ABI, consequently it translates a set of OS and user-level instructions composing one platform to another. Figure 3.5 illustrates these concepts.

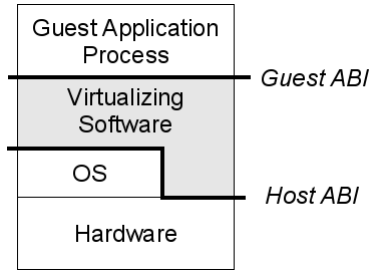


Figure 3.5: The Process virtual machine.

Process virtual machines have multiple applications:

Multiprogramming. This kind of process virtual machine is present in every operating system supporting the execution of multiple user processes. Each process is provided with a process virtual machine and its task it to give to the process the "illusion" of having exclusive access to the hardware.

Emulators and Dynamic Binary Translators. Process virtual machines can be used to execute programs compiled for an instruction set different than the ones executed by the host hardware. This can be achieved through *emulation*, also called *interpretation*, which fetches and decodes each instruction and then emulates its semantics using host instructions. This can be a slow process, requiring tens of the host instructions.

Dynamic binary translation can be used to improve this process. In this case "blocks" of guest instructions are translated into host instructions that perform equivalent functions.

Same-ISA Binary Optimizers. As then name suggests, these process virtual machines simply work as run-time optimizers of the binary code of an application.

Platform Independence. Process virtual machines can be used to create platform-independent software. In this case the process virtual machine works as a translator from a *virtual ISA*, namely an ISA which is paired with the virtual machine and is not necessarily implemented in any real hardware, to the host ISA.
The idea is to compile the guest process into the *virtual ISA* and distribute it for execution on different platforms. Then, for each platform, a virtual machine capable of executing the virtual ISA is implemented.
Sun Microsystems JAVA VM and Microsoft CLI are widely known examples of this type of Process Virtual Machine.

3.3 Virtual Machines and AOP

In recent years, several solutions have been based on virtual machines to implement the dynamic weaving of aspects on code.

The JRockit [1], AspectWerkz [3], SteamLoom [2] virtual machines are three examples of virtual machines that natively support the weaving of aspects. They are all based on the Java virtual machine and this is not mere coincidence: the Java specification [5] enforces a strict and well defined definition of the *class* binary files that contains the compiled bytecode. This means that the format of the file is constrained and this simplifies a lot the process of identification of joinpoints and weaving of code at run time.

These virtual machines use different techniques for identifying the joinpoints within the bytecode. The JRockit virtual machine identifies the joinpoints as events within the virtual machine, such as method calls. The SteamLoom virtual machine, instead, defines the concept of *joinpoint shadows*, which are sets of one or more bytecode instructions that the virtual machine can identify within the compiled code of a method. AspectWerkz instead recognizes joinpoint markers that were inserted in the code at compilation time.

At run time, when the joinpoint in the code are encountered during the exe-

cution, all these virtual machines substantially substitute or modify and then re-compile the original code with the code defined in the aspect. This is a rather efficient solution because the weaving is triggered by the virtual machine itself at the right time and only if needed.
This strategy for code manipulation at run time provides several benefits [1] if compared with approaches that involve code instrumentation (approach adopted by Javassist [15]) which consists in parsing and manipulating the code after compilation, possibly while the program is executing.

AspectK Virtual Machine. The dynamic weaving in the AspectK virtual machine adopts methodologies which are equivalent to those of the virtual machines just presented.
In the AspectK virtual machine the joinpoints are identified by mean of marks (the *aspect tags*) that are inserted in the bytecode at compilation time, similarly to the approach adopted by the AspectWerkz virtual machine.

Moreover, whenever a cut matches a joinpoint at run time, the virtual machine invokes the *weaver*, which attempt the trap on the action (the only joinpoint in AspectK) and if this operation succeeds the weaver compiles the code contained in the advice using the *just in time* compiler and then weaves it into the original code.

Chapter 4

The AspectK Interpreter

This chapter describes the architecture of the AspectK interpreter, which is a software that was designed and implemented with the objective of exploring possible solutions for supporting the dynamic weaving of aspects within a virtual machine.
The work on the interpreter was useful to determine how to challenge the development of the virtual machine and to track down the pivotal issues of the dynamic weaving in AspectK at an early stage of my work.
The interpreter is implemented in Java.

4.1 Design

The design of the interpreter is rather simple. Figure 4.1 illustrates the major components at play.

The front-end component is represented by the *parser*. The parser is in charge of analyzing the AspectK source code and transforming the data therein contained into a suitable form[1].

[1] More information on the parser will be provided in Section 5.6.1.

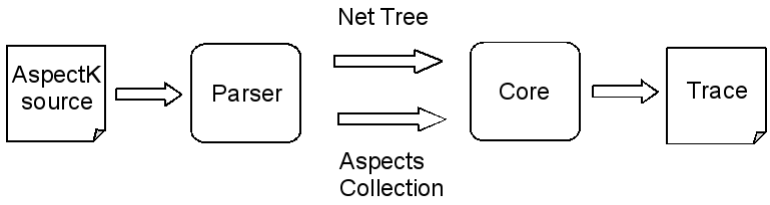


Figure 4.1: Overview of the interpreter

The parser divides the information into two parts, namely:

- the *net* specification, a collection of process definitions and tuples;
- a (optional) collection of *aspects* associated to that net.

After the parser comes the *core* component, which is the central part of the interpreter. Its task is to setup the environment defined by the net and then execute the processes, action after action, until no process is able to evolve any more. Furthermore, the core can inject actions whenever one or more of the aspects *trap* the actions in the net.
Figure 4.1 illustrates additional element, the *trace* component, which is used by the interpreter to record and save the evolution of the net.

The software architecture of the interpreter is shown in Figure 4.2.

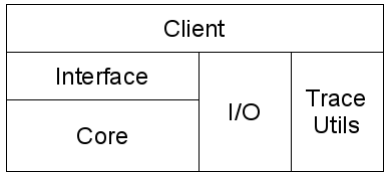


Figure 4.2: Software Components

4.1.1 Interpreter Features and Limitations

The important features implemented in the interpreter are:

- recording of the execution;

- possibility to choose which process to execute, or otherwise let the system choose;
- possibility to enable/disable aspects at runtime;
- possibility to add and remove aspects at runtime;
- support of the actions $eval(P)@\ell$ and $newloc(\overrightarrow{\ell})$.

On the other side there are some significant limitations, due to the fact that this is a prototype:

- the trapping of actions is *not* attempted on the injected actions, thus the interpreter behaves differently from the AspectK specification presented in the "*Advice for Coordination*" [4] paper;
- the condition evaluator has a rather incomplete implementation. *test* conditions can bind variables and the truth value of the condition may depend on the value of such variables. If multiple bindings exist, the condition evaluator chooses one and commits to it: if the condition evaluates to false for such choice, no backtracking is performed and no other binding is attempted.
- some of the features supported by the core are not made available to the user through the user interface. For example it is not possible to enable or disable the aspects.

4.1.2 *Net* Internal Representation

In order to understand the working of the interpreter it is necessary to spend some words on the data structure that is used to model the net. The example of Figure 4.3 will be used to facilitate the explanation and the comprehension of this matter.

```
loc1 :: <t1, t2>
||
loc2 :: read(!x, !y)@loc1.0 + read(!z)@loc3.out(z)@loc1.0
||
loc3 :: *out(a)@loc3.0
```

Figure 4.3: Code example

A net definition is encoded into a tree data structure. Each internal node of the tree describes an operator, which can be one of those reported in the following list:

- `||`, parallel composition of nets;
- `::`, location of a process or a tuple;
- `|`, parallel composition of processes;
- `+`, guarded sum of processes;
- `*`, replication of processes;
- `.`, sequential composition of actions.

Each leaf node, instead, describes one of the following entities:

- a location;
- a tuple;
- an action;
- a null process (`0`).

The net in the example is transformed into the tree depicted in Figure 4.4.

4.2 Interpretation

The interpreter features a multithreaded architecture, each process in the net is emulated by a different Java thread. In the previous section we described how tuples and processes are encoded into sub-trees of the `||` nodes.

The interpreter identifies these sub-trees and then it initializes the environment as follows:

1. it scans the net tree and creates a set of *tuple* sub-trees and a set of *process* sub-trees;

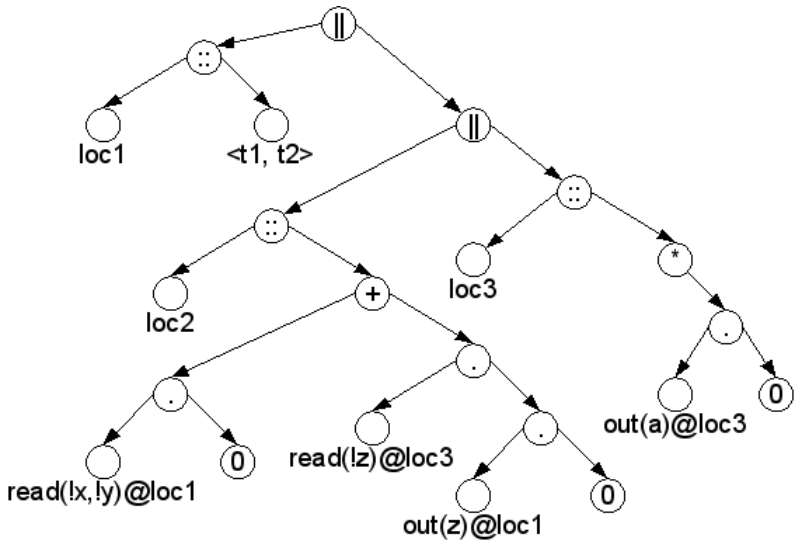


Figure 4.4: Net tree

2. for each tuple, the location the tuple is located at is registered among the "known" locations. Moreover, the tuple is added to the set associated to this location;

3. for each process, the location the process is located at is registered among the "known" locations. Moreover, a new thread is allocated for such process, it is bound to the sub-tree representing the process and it is added to the set of threads associated to this location. Each thread has a variables table associated to it, where it records what variables are available at a certain point of the execution and what their values are.

Each thread walks the branch of the net tree it is bound to and it executes two kind of tasks:

1. *actions*, encoded into the action nodes, which may affect the global environment by removing/creating tuples and binding new variables in the process' variable table;

2. *operators*, encoded into the operator nodes, which affect the process' behavior.

Two or more threads are bound to a same sub-tree *only* in case of replication.

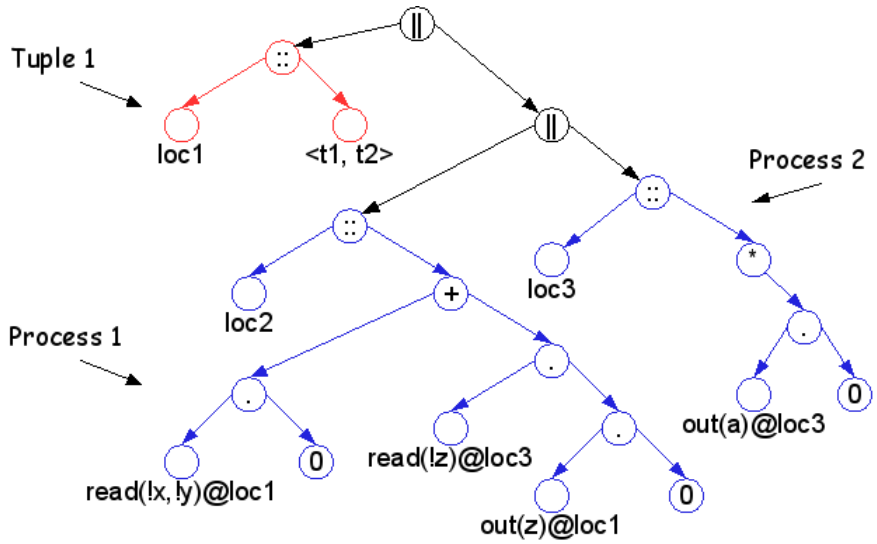


Figure 4.5: Processes and tuples are evidenced on the example net tree

Figure 4.5 illustrates the two sets of tuples and processes defined by the interpreter in the example introduced in Section 4.1.2. The set of tuples initially contains only one tuple and the set of processes contains two processes. *Process 2* is a process that replicates thus, each time it executes, it spawns a new thread bound to the sub-tree which is the children of the operator node *.

Processes that replicate *never* terminate. This means that in presence of a process that replicates the evolution of the net never reaches a termination point.

4.3 Scheduling Of Processes

At any point during its execution, a process can be in one of the following states:

1. *active* execution;
2. *termination*, namely the process is the null process 0.
3. *block*, namely the process hasn't terminated it's execution but it cannot proceed further. This situation occurs because of the absence of a tuple at a certain location that prevents a *read* or *in* action from being fired.

A net can evolve only if it contains some active processes. If no active process is present, the net has reached a final state from which it cannot evolve.

This idea of different states for a process can be used to categorize them and determine the global state of execution of the net:

Green Set: a process is *green* if it is guaranteed that it can successfully perform the next task.

Yellow Set: a process is *yellow* if it is unknown whether it can perform the next task.

Red Set: a process is *red* if it is guaranteed, given the current state of the net, that it will fail to perform the next task.

Grey Set: a process is *grey* if it is the null process.

Replication Set: a process is in the *replication* set if it replicates. These processes could be placed in the green set, however having them in a separate set adds additional information on the nature of the processes present in the net.

Using these categories it is possible to define several scheduling strategies.

4.3.1 Interactive Execution

This execution mode allows the user to manually select which process to execute.

The algorithm works as follows:

1. At the beginning the threads are created. Each of them is tested (not executed) and:
   - if it can successfully perform the next task, given the current state of the net, it is inserted in the green set;
   - if it cannot because the next task is an *in* or a *read* action and the matching tuple for the action is missing, it is inserted in the *red* set;
   - if the process replicates, it is inserted in the *replication* set.
2. The processes belonging to the *green* and *replication* set are exposed for selection and the user chooses one of them. If, instead, these sets are empty, the execution terminates;

3. The process executes the task;

4. One of the following choices is performed:

   - if the process that was executed was a process that could replicate, jump to step 2;
   - if an action was performed and such an action was an *out* action, all the processes in the *red* set are tested again and placed either in the *green* or back in the *red* set. In fact, the *out* action might have generated a tuple that is required by one or more of the *red* processes. Jump to step 2;
   - if an action was performed and such an action was an *in* action, all the processes in the *green* set are tested again and placed either back in the *green* set or in the *red* set. In fact, the *in* action might have removed a tuple that is required by one or more of the *green* processes. Jump to step 2;

4.3.2 Non-Interactive Parallel Execution

This execution mode does not allow the user to select which process to execute. The algorithm works as follows:

1. at the beginning the threads are created. They are all inserted in the *yellow* set. The other sets are all empty;

2. all the threads in the *yellow* and *replication* set are executed. If such sets are both empty, terminate.

3. For each thread:

   - if the thread successfully performed its task, it remains in the *yellow* set;
   - if the thread failed (missing tuple) it is moved to the *red* set, unless one of the threads in the *yellow* set performed an *out* action. In fact, all the threads are executed in parallel and the threads in the *red* set might have failed *before* that the tuple was created;
   - if the thread replicates, it is moved to the *replication* set;
   - if the thread has become the null process, it is moved to the *grey* set.

4. Go back to step 2.

4.3.3 Non-Interactive Sequential Execution

This execution mode is equivalent to the one of the previous section, except for the fact that only one thread from the (*yellow* ∪ *replication*) set is executed instead of launching all of them in parallel.

1. at the beginning the threads are created. They are all inserted in the *yellow* set. The other sets are all empty;
2. One thread in the (*yellow* ∪ *replication* set) set is executed. If, instead, those two sets are both empty, execution terminates.
3. One of the following:
   - if the thread successfully performed its task, it remains in the *yellow* set;
   - if the thread failed (missing tuple) it is moved to the *red* set;
   - if the thread replicates, it is moved to the *replication* set;
   - if the thread has become the null process, it is moved to the *grey* set.
4. Go back to step 2.

4.4 Dynamic Weaving of Aspects

The interpreter features two strategies for the dynamic weaving of aspects. They are presented in this section.
It is assumed that the aspects are available in the environment within a set (the *aspect pool*) and that they can be added/removed or enabled/disabled at any time during the execution, except at the very moment of the weaving.

4.4.1 Lazy Weaving

This first technique implements the Just-In-Time Weaving.
The name *lazy* suggests the fact that the weaving of actions is performed at the very last moment, when the trapped action is about to be executed.

The algorithm is given below:

1. lock the aspect pool. No aspects can be add/removed or enabled/disabled while the aspect pool is locked.
2. iterate through the aspects, following their order of declaration (see Section 2.2.3). For each enabled aspect attempt the trap and:
   - If the trap succeeds, extract from the advice the actions to inject.
   - If the trap does not succeed move to the next aspect.
3. create the list of pre-actions and post-actions to inject, by assembling all the pre-actions and post-actions extracted by the aspects.
4. create a sub-tree using the set of injected actions and link it to the original net tree.
5. move the execution pointer to the first pre-action.
6. unlock the aspect pool.

For example, we may consider the following AspectK net:

```
let
    A[$User :: out(x)@y] ^= read(x)@y proceed out(y)@x
in

    loc :: out(sweet)@home.0
```

The net tree is shown in Figure 4.6.

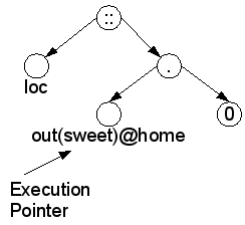


Figure 4.6: `loc :: out(sweet)@home.0`

Step 4 of the algorithm is depicted in Figure 4.7. The sub-tree contains the pre-action, the post-action and the original action. The original action must be included in order to avoid any modification of the links in the original net tree.

The sub tree has an "open" link, which is attached to the original net tree, as shown in Figure 4.8. The execution pointer is moved too.

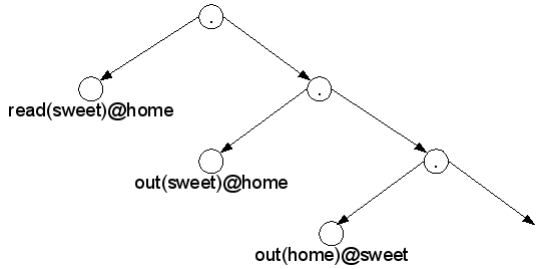


Figure 4.7: Sub-tree

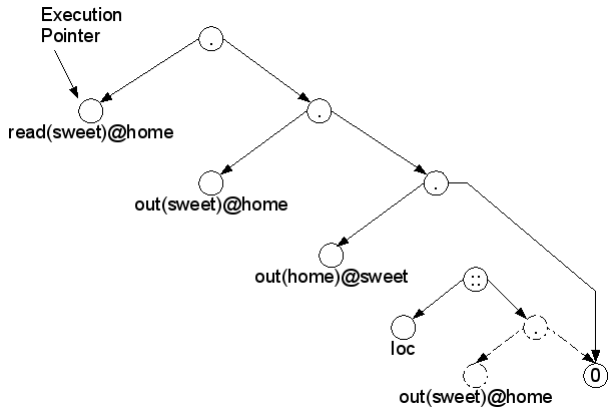


Figure 4.8: The net tree after the weaving

4.4.2 Balanced Weaving

The *Balanced* weaving approach implements the dynamic weaving described in Section 2.3. This strategy tries to reduce the amount of computational power required by the weaving at run-time by moving part of the work at the time when an aspect is either added or enabled in the system.
It is possible to categorize the aspects into two different sets: the ones that can be used in the dynamic weaving and the ones that can only be used for the JIT weaving (see Section 2.3 for more information about this issue).

Whenever an aspect is added in the system (or it is re-enabled if it was disabled), the net tree is traversed and trap is attempted on each action. If dynamic weaving is not feasible, the weaving process is aborted and the action skipped, otherwise the list of substitutions returned by the *check* function is cached for being used when the JIT weaving is re-attempted when the action is executed. The *balanced* weaving cannot substitute the JIT weaving but (in

certain conditions) it can optimize it.

4.5 Lessons Learned and Considerations

The work on the interpreter underlined some interesting aspects to be taken care of in the design of the virtual machine.

First of all Aspectk is a coordination language that describes systems of processes running in parallel. Hence, a first requirement for the virtual machine is to support the execution of multiple threads executing in parallel.

Aspects can be modified or disabled/enabled at any time, and this implies that the injected may need to be modified several times during the execution. This fact underlined the importance of keeping the injected code separated from the net to facilitate its removal and re-injection

Using categories for classifying the state of the processes seemed to be a useful technique that could improve the activity of the scheduler and that could facilitate the determination of the state of the net. In Chapter 5 we see that the most suitable scheduling strategy for the virtual machine is the one that enforces the "non-interactive sequential" execution.

The experience with the interpreter was beneficial in understanding what the possible strategies for weaving could be, besides of course the one defined in AspectK itself, identified by the *lazy* weaving. In the interpreter, the *balanced* weaving strategy can be considered as a mere optimization however, as we will see in the next chapter, in the virtual machine it is extended to become a standalone weaving technique.

Chapter 5

The AspectK Virtual Machine

This chapter discusses the design and the implementation of the AspectK virtual machine.
Section 5.1 provides an overview on the architecture of the virtual machine.
Sections 5.2 and 5.3 describe the memory layout of the virtual machine.
Section 5.4 presents the scheduler.
Section 5.5 provides an overview on the format of the bytecode and on the instruction set of the virtual machine.
Section 5.6 illustrates the compilation scheme and the implementation of the AspectK compiler.
Section 5.7 analyzes the design of the weaver.
Section 5.8 describes the weaving strategies implemented in the virtual machine.
Section 5.9 provides a brief description of the garbage collector.
Section 5.10 illustrates the optimizations implemented in the virtual machine.

A note on the terminology: unless explicitely stated, the locations that compose a tuple will be referred to as *terms* and the term *location* will be reserved to define the "location" at which a process or a tuple is located.
Moreover, the expression *pre-actions* is used to define those actions that, whenever a trap operation succeeds, are injected *before* the trapped action. *Post-action*, instead, defines those actions injected *after* the trapped action.

5.1 Overview

The AspectK virtual machine is a process virtual machine designed to support the AspectK coordination language.
The design of the AspectK virtual machine is inspired by the one of two other virtual machines, namely the Java [5] and the TyCO [16] virtual machines.
The Java virtual machine was taken as a model for the initial definition of the instruction set and for the design of the bytecode layout. The TyCO virtual machine is an example of virtual machine for process calculi and it provided some overall insights on how a process calculus can be compiled for the execution on a virtual machine.

Popek and Goldberg defined *efficiency* as one of the fundamental characteristics that every virtual machine should possess. The AspectK virtual machine was designed and developed seeking a compromise between efficiency and simplicity of design and implementation. Given the time constraints and the absolute lack of experience in the development of virtual machines Java was the language of choice for the development. In general, I tried to focus the effort on the completion of the virtual machine rather than its optimization. However, some small optimizations are present and they are discussed in Section 5.10.

The basic characteristics of the virtual machine are:

- Big endian format
- 32-bit long words
- $2^{31} - 1$ bytes addressable[1].
- The virtual machine is implemented as an *emulator* (see Section 3.2). Each bytecode instruction is fetched, decoded and emulated using Java code. Using the terminology of chapter 3, the host ISA is the Java bytecode, whereas the guest ISA is the AspectK bytecode.

The architecture of the virtual machine is quite simple. Two major components can be identified, namely the virtual machine *core*[2] and the *weaver*[3]. The core part implements the emulator and the weaver is in charge of weaving the code specified by the aspects that are stored in the aspect pool[4].
Two other major components in the core are

[1] This limitation is due to Java, since the native types are signed and the most significant bit is used for the sign.
[2] File `vm.VirtualMachine.java`
[3] Package `vm.aop.weaver.*`
[4] File `vm.aop.AspectPool.java`

- The *scheduler*, in charge of the scheduling of the threads in the *Thread Queue* (see Section 5.2.3). The scheduler is discussed in Section 5.4.

- The *garbage collector*, which takes care of freeing the memory not being accessed anymore (see Section 5.9).

The other components of major interest used by the weaver are:

- The *trapper* (see Section 5.7) which is the component providing the methods for trapping actions.

- The *just-in-time compiler* (see Section 5.7.3), which compiles on-the-fly the actions to be injected into bytecode.

- The *condition evaluator* (see Section 5.7.4), which can evaluate the truth value of the conditions embedded in the advices (with backtracking).

A final module of interest is represented by the library package[5] that contains code which is shared by many classes. An important component of the library is the *bytecode reader*[6], an utility that can be used to read and extract information from the bytecode. This utility is used by the weaver and by the *disassembler* (see Section 5.6).

5.1.1 Net Representation and Execution

This section presents an overview on the scheme used to compile a net into bytecode and it explains how it is executed. The objective of this section is to provide an high level understanding on how the virtual machine operates, whereas details about the single components of the virtual machine will be then provided in the next sections.

A net is formed by two classes of components:

- the processes;

- the tuples;

[5] Package `lib.*`
[6] File `lib.memory.BytecodeReader.java`

Processes

The *action* is the atomic component of a process. Process operators can be used to create new processes by modifying the behavior of a process or by composing multiple processes together. Processes describe a concurrent system, they coexist and execute in parallel at the same time.

The design of virtual machine has to support the concurrent execution of multiple processes: no process should be allowed to keep the processor busy for the entire duration of its execution.

The fact that a process is composed by atomic parts, the actions, can be used to split the execution of a process in multiple chunks. The idea is to allow a process to control the processor for the amount of time necessary to execute the action and then possibly move the control to another process. For the moment we can simplify the discussion by ignoring the presence of operators such as $*$, $|$ or $+$ and consider a process as a simple sequence of actions.

In the AspectK virtual machine the state of each process is traced using a special frame, the *thread frame*. The thread frames are placed in a special region in the memory, the *thread queue* and the *scheduler*, takes care of determining in which order the processes are executed.

The actions are compiled into a chunk of code whose structure is shown in Figure 5.1. A process is allowed to execute one *action chunk* before passing the control of the execution back to the scheduler.

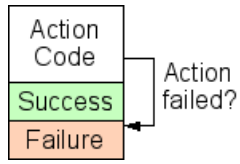


Figure 5.1: A chunk of code.

Three parts are identifiable in a chunk:

1. the first part, containing the bytecode that executes the action;
2. the second part, colored in green, is executed *only* if the action was successfully performed.
3. the third and final part contains the code executed *only* if the action couldn't be performed.

Tuples

The tuple is an array of terms and each term is a *string*. The string is encoded in the virtual machine using a special frame, the *byte array*. the sequence of byte arrays that model a tuple are stored in an additional frame, the *reference array*. The reference array stores in each cell the reference of the byte array representing the corresponding term.

The frames encoding the tuples, along with all the other frames with the exception of the thread frames, are placed in a special area of the memory, the *heap*.

Other Components

There are other two important structures used by the virtual machine to track the execution of the net. They are both allocated on the heap:

- The *global constants table*. AspectK does not support input coming from the user, thus the number of constant locations present in the net is invariant[7]. It seems reasonable then to pre-allocate all these constants and place them in a global table, thus avoiding the need to allocate them again whenever they are needed within a process or a tuple.

- The *channel list*. Locations are the point of synchronization for the processes. They are *asynchronous channels* since a process can post a message in a location without waiting for another process to receive it. Each time a new location is created (because a tuple or a process is located there) a new *channel frame* is allocated in the channel list, which is implemented using the frame *reference list*.

Execution

We have briefly seen in the previous paragraphs how tuples and process are treated internally by the virtual machine. In order to understand how a net is executed we need first to present how it is compiled into code. The compiled program representing a net is made of two parts:

[7]The weaving of actions may introduce new constants. In Section 5.7.2 we will see how to face this challenge.

1. the first part is the *initialization* area. It is the equivalent to the *main* method of languages like Java. It performs the following tasks, in order:
   (a) It allocates all the global constants in the global constants table.
   (b) For each tuple, it allocates in memory its memory representation, it creates, if it does not exists yet, the channel in the channel list corresponding to the location; finally it binds the tuple's memory representation to the channel frame.
   (c) For each process, it allocates in the thread queue a frame which represents it. Each frame points to the address of its first instruction in the code.
2. the second part contains the code of the processes. Each process is basically a sequence of chunks of code similar to the one shown in Figure 5.1, plus some other special chunks representing some of the operators.

Initially the virtual machine starts the execution from the beginning of the initialization part of the program. At its end the control is passed to the scheduler, which schedules the first thread frame.

The thread frame contains some information about the process, among which the value of its *program counter* and the reference to the *variable table* of the process, which is a *reference array* containing the values of the variables appearing in the process. The scheduler changes the value of the program counter to the one stored in the thread frame and the virtual machine starts the execution of the instructions of the chunk pointed by the program counter. At the end of the chunk the control is passed back to the scheduler, which schedules the next thread frame, and so on.
When no more thread frames are available, the virtual machine terminates the execution.

Whenever an input action is performed the variable table of the corresponding thread frame is updated.

If an *in* action is performed, the selected tuple/reference array is removed from the list of tuples (implemented using reference list frames) pointed by the location/channel frame.

Similarly, whenever an *out* action is performed a tuple is added to the list pointed by the target channel.

The virtual machine supports an additional kind of frame, the *code frame*, which is used to encapsulate the code woven by the weaver.

5.1.2 Remarks

There are some remarks to take into account, regarding the design of the virtual machine.

- Some of the logic of the execution and part of the information on how to manipulate the program data are embedded in the bytecode instructions (see for example the `MATCH` instruction that implements the logic of the AspectK *match* function). This approach simplifies the design and make it easier to support other functions or other data structures. In fact, it is enough to implement additional bytecode instructions that can manipulate them.
- The *string* is the only type used in AspectK. The virtual machine features a limited set of instructions and the only instruction that manipulates strings is `NEWSTRING` that permits to create a string.
- The virtual machine, internally, handles several types of data, namely the *reference* (4 bytes), the *byte* (1 bytes), the unsigned *integer* (4 bytes), the signed short integer (2 bytes) and the unsigned short integer (2 bytes) types. None of these types is exposed through the virtual ISA (Instruction Set Architecture).
- The virtual machine defines the `null` reference as a reference of value `0x00000000`.

5.2 Runtime Data Areas

This section presents the memory architecture of the virtual machine, which is illustrated by Figure 5.2.

The Heap and the Thread Queue grow towards each other. When the two areas collide an exception signalling memory exhaustion is thrown.
There are four major memory areas that are used for the execution.

5.2.1 The Program Area

The *program area* is the place where the program bytecode is stored. The first 4 bytes of the program are used to store the size of the global constant table,

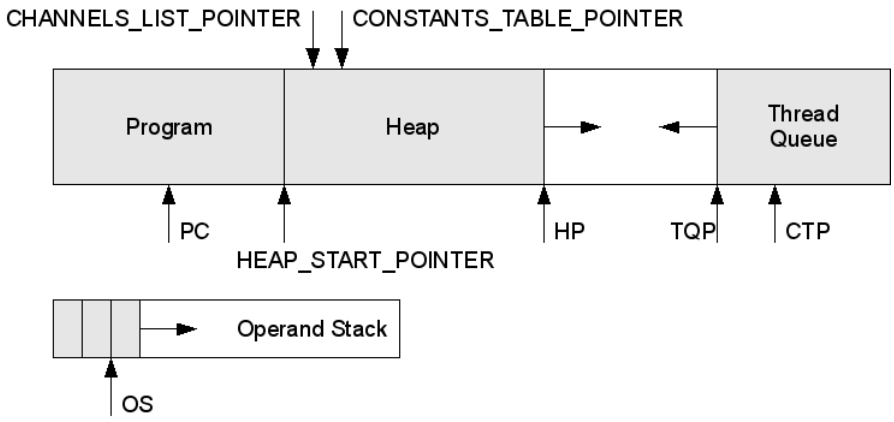


Figure 5.2: Memory layout of the virtual machine.

which is initialized by the virtual machine. After the initialization of the global constant table the program counter is moved to the address 4.
The machine register `PC` stores the value of the program counter.

5.2.2 Heap

The *heap* is the area in the memory where the dynamic data structures are stored. The basic building block of the heap is the byte and the basic allocation unit is the frame.
The heap is created at the virtual machine start-up and its start address is recorded in the `HEAP_START_POINTER` register.
Heap storage for frames is reclaimed by the garbage collector. No frame is ever explicitly deallocated.
The `HP` register stores the first free byte of the heap.

5.2.2.1 Global Constant Table

The *global constant table* is a reference array which stores all the constants defined within the net. This structure is initialized at the virtual machine start-up and its size is stored in the first 4 bytes of the program code.
The address of the global constant table is available at the machine register `CONSTANTS_TABLE_POINTER`.

5.2.2.2 Channel List

The *channel list* is a list (implemented using the reference list frame) of channel frames. Each channel frame encodes one location at which one or more tuples and/or processes can be located.
This structure is initialized at the virtual machine start-up.
The address of the channel list is stored at the machine register `CHANNELS_LIST_POINTER`.

5.2.3 Thread Queue

The *thread queue* stores the thread frames representing the processes in execution.
The register `CTP` stores the address of the thread frame currently in execution.
The register `TQP` stores the address of the last thread added in the queue.
No thread frame is ever explicitly deallocated, instead it is the garbage collector that is in charge of removing the frames of threads not in execution anymore.

5.2.4 Operand Stack

The operand stack is used by the instructions to read some of their arguments or store the result of their computation.
For example, the `MATCH` instruction, which implements the *match* function of Figure 2.5 (page 9), reads the reference of the tuple to match from the operand stack and outputs the result of the matching, which could be either the `null` value or the reference of the returned frame, again on the operand stack.
A cell of the operand stack is 32 bit of size.
The index of the top value in the stack is stored in the `OS` machine register.

5.3 Frames

This section presents the frames, which are the basic unit of allocation of memory of the virtual machine. The next sections present each frame. In the figures, large cells indicate that the field has a size of 32 bits, tiny cells indicate a size of 8 bits. Italic font indicates that the field is a reference.
Every frame has a first field named *Descriptor*. This special field is used by the garbage collector to determine the type of the frame and to mark the frames when they are visited during the collection process.

5.3.1 Byte Array Frame

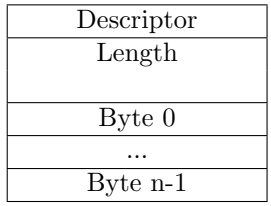


Figure 5.3: The byte array frame.

The byte array frame, presented in Figure 5.3 holds a sequence of bytes.
The 32-bit-long field *size* allows the instantiation a byte array of at most $2^{31} - 1$ cells.
When allocated, all the cells of the byte array are initialized to `null`.

5.3.2 Reference Array Frame

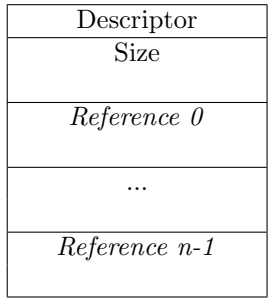


Figure 5.4: The reference array frame.

The reference array frame, presented in Figure 5.4 holds a sequence of references.
The 32-bit-long field *size* allows the instantiation of reference arrays of at most $2^{31} - 1$ cells.
When allocated, all the cells of the reference array are initialized to the `null` value.

5.3.3 Reference List Frame

The reference list frame (Figure 5.5) is a frame that can be used to build lists of frames. It has three fields:

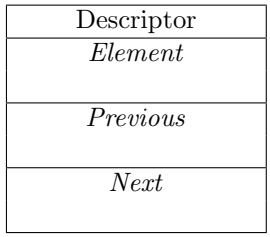


Figure 5.5: The reference list frame.

- *Element.* Stores the reference to the frame that constitutes the element at this position of the list.
- *Previous.* Stores the reference to the previous reference list frame.
- *Next.* Stores the reference to the next reference list frame.

Reference lists may be not allocated using any instruction of the ISA. They are handled internally by some instructions of the ISA.

5.3.4 Channel Frame

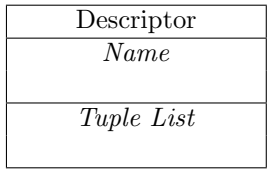


Figure 5.6: The channel frame.

The channel frame (Figure 5.6) encodes the location in AspectK. The channel has two fields:

- *Name.* Stores the reference to a byte array defining the string name of the current channel.
- *Tuple List.* Stores the reference to the first reference list frame of the list of tuples that are located at this location. If there is no tuple, the value of this field is `null`.

Channels are created using the `NEWCHAN` instruction, which reads the reference of the byte array containing the name of the channel from the operand stack.

A tuple can be appended to the channel using the `APPEND` instruction, which reads the reference of the reference array representing the tuple from the operand stack and appends it to the list pointed by *Tuple List*.

5.3.5 Thread Frame

Descriptor
Status
Channel
Variable Table
Program Counter
Aspect Code Reference
Aspect Variable Table Table
Aspect Constants Table Table

Figure 5.7: The thread frame.

The thread frame (Figure 5.7) stores the execution state of a thread.
It contains the following fields:

- *Status.* This field is a flag that signals whether the thread is active or *gray* (see Section 5.4).
- *Channel.* This field stores the reference to the channel/location at which the thread/process is located.
- *Variable Table.* This field stores the reference to the variable table, which contains all the variables defined in the process.
- *Program Counter.* Stores the reference of the chunk of code that the thread executes next.
- *Aspect Code Reference.* Stores the reference of the code frame of which the thread is currently executing a chunk.
- *Aspect Variable Table table.* Stores the reference to the table (reference array) that contains the references to the variable tables associated to the current aspect code frame and its parents.

- *Aspect Constant Table Table.* Stores the reference to the table (reference array) that contains the references to the constant tables associated to the current aspect code frame and its parents.

The last three fields are specific to the weaving of code at run time. Their function is clarified in Section 5.7.
Thread frames are allocated using the instruction `NEWTHREAD`. The program counter is updated using the scheduling instructions (Section 5.5.2).

5.3.6 Code Frame

Descriptor
Freshness
Static Flag
Size
Parent
Number of Variables (16 bits)
Number of Constants (16 bits)
Code ... Code ...

Figure 5.8: The code frame.

This frame (Figure 5.8) is used to store the code woven by the weaver.
The code frame contains the following fields:

- *Freshness.* This field is used to determine whether the code stored in this frame reflects the current state of the aspect pool.
- *Static Flag.* This field is used by the *balanced* weaver (Section 5.8.3) only.
- *Size of Code.* This field stores the size in bytes of the code stored in this frame.
- *Number of Variables.* Stores the number of variables that are defined within the woven code. This field is used in the *balanced* weaving approach (Section 5.8.3).

- *Number of Constants.* Stores the number of constants that are defined within the woven code. This field is used in the *balanced* weaving approach (Section 5.8.3).

- *Code.* This is a variable-length field that stores the woven code.

5.4 The Scheduler

The virtual machine determines the order of execution of the threads inserted in the *Thread Queue* by using a special component, namely the *scheduler*. The scheduler is implemented using some of the ideas presented in the section about the interpreter (4.3).
In fact, the scheduler classifies the threads in four sets:

YELLOW set. In this set fall those threads that have just been created (using the instruction `TH_NEW`) and those threads that have successfully executed a chunk of code and are ready to execute the next.

RED set. Contains the threads that failed to execute their chunk of code.

GREY set. Those threads that terminate fall in this set.

REPLICATION set. The threads that can replicate fall in this set.

Threads belonging to the YELLOW and REPLICATION sets can be scheduled for execution.
Moreover if the execution of a chunk of code ends with a tuple being output in the environment, the threads belonging to the RED set are re-inserted into the YELLOW set.
Threads belonging to the GREY set are simply ignored by the scheduler.

The scheduler can use two different strategies for deciding which thread to schedule:

FIFO. First-In-First-Out, namely the threads are scheduled according to the order they entered the queue.

RANDOM. The new thread to schedule is chosen randomly among the available ones.

In the random strategy, the random choice done for the scheduling may lead to the starvation of one of more threads, which may be then forbidden the execution for an indefinite time. In order to avoid this problem the following technique is used: the threads are ordered by the value of the expression $key = (c - c_{stamp}) + rnd$, where:

- c is a global counter, incremented by 1 each time a thread is inserted in the scheduler.
- c_{stamp} is the value of c when the thread was inserted in the scheduler.
- rnd is a random value between 1 and n, both ends included.

The scheduler always schedules the thread with the highest key.
The more a thread waits in the queue the more the value $(c - c_{stamp})$ grows and the less it is likely that a thread just added obtains a key bigger than its. In the worst case scenario a thread has to wait for the execution of at most $n - 1$ threads. In fact:

- When the "unlucky" thread A is inserted in the scheduler, $(c - c_A) = 1$, $rnd = 1$ and $key = 1 + 1$.
- Each time a new thread is inserted in the scheduler, the expression $(c - c_A)$ increments by 1.
  Thus

  $$\begin{aligned} key_A > key_{other} &\iff (c - c_A) + rnd_A > (c - c_{other}) + rnd_{other} \\ &\iff 1 + t + 1 > 1 + n \\ &\iff t > n - 1 \end{aligned}$$

  t counts the number of threads inserted after A and in the worst case scenario $rnd_{other} = n$.
- After $n - 1$ threads have been inserted in the scheduler, the key of the thread A will be always bigger than the one of any other newly inserted thread. Hence, it is guaranteed that thread A will be executed after at most $n - 1$ threads since its insertion in the scheduler.

The algorithm used by the scheduler corresponds to the one used for the "non-interactive sequential" execution implemented in the interpreter, described in Section 4.3.3.

5.5 Instruction Set Summary

This section provides an overview on the instruction set of the virtual machine. Complex instructions receive a brief analysis. All the instructions can be found in alphabetical order in Appendix C.

5.5.1 Bytecode

AspectK programs are compiled by the AspectK compiler directly into bytecode.
The layout of a bytecode instruction is rather simple. The first byte of the instruction represents the operation code. The subsequent bytes instead determine the arguments.
Arguments can be of different sizes, namely 1, 2 or 4 bytes depending on the instruction.
The type of operation code usually determines the number of arguments, but some operation codes have a *variable* number of them. In such cases the first argument reports the total number of arguments of the instruction.

5.5.2 General

In this section we present instructions used for general tasks.

Branching

The branching instructions are:

`JMP_ABS` Unconditioned jump, jumps to an absolute address provided as an argument.

`JMP_OFF` Unconditioned jump. The address is provided by mean of an offset.

`JMP_NULL` Jump if the top element of the operand stack is `null`. The address is provided by mean of an offset.

`JMP_NOTNULL` Jump if the top element of the operand stack is not `null`. The address is provided by mean of an offset.

`SWITCH` This instruction has variable number of arguments. The first argument is 1 byte long and determines the number n of subsequent arguments, thus there can be at most 256 more arguments. Each of the subsequent n argument is an offset, 2 bytes long.
Switch reads a value $v : 0 \leq v \leq 255$ and jumps to the address corresponding to the v^{th} offset (the first argument *size* is not considered in the count).

Byte Array Creation and Manipulation

The instructions for manipulating a byte array are given below:

`NEWBARRAY` Creates a new byte array, the size is given as a 4-byte-long argument. The reference of the newly allocated array is stored in the operand stack.

`BASTORE` Stores a byte provided as an argument in a byte array at a certain position.

Strings

Only one instruction is provided:

`NEWSTRING` It is a variable-length instruction. The first argument is unsigned 1 byte long and determines the length "l" of the string. The subsequent "l" arguments are 1 byte long and they encode a char in ASCII format.
This instruction allocates a byte array with the content of its arguments. The reference of the newly allocated array is stored in the operand stack.

Reference Array Creation and Manipulation

The instructions for manipulating a reference array:

`NEWARRAY` Creates a new reference array, the size is given as an argument. The reference of the newly allocated array is stored in the operand stack.

`ALOAD` Loads on the operand stack a reference from a reference array.

`ASTORE` Stores in the index provided as an argument a reference into a reference array. Both the reference and the reference array's reference are found in the operand stack.

Channel Creation and Manipulation

`NEWCHAN` Creates a new channel frame. Reads from the operand stack the reference of the string of the name of the channel. Returns on the operand stack the reference of the newly allocated channel.

`GETCHAN` Given the string's reference on the operand stack, this instruction returns the channel associated to it. The instruction traverses the *channel list* and when the channel with a matching name is found, it copies its reference on the operand stack. If no channel is found, `null` is returned on the operand stack.

`APPEND` Appends the reference of a tuple at a channel's tuple list.

`MATCH` This instruction attempts a match on the tuples located at a certain location. This instruction implements the *match* function of Figure 2.5. First it reads the reference of the tuple to match from the operand stack. Each cell of the reference array representing the tuple to match, contains one the following values:

- If the term of the tuple is a *variable definition* $!u$ the cell contains the `null` value.
- Otherwise the cell contains the reference to the constant term (string).

`MATCH` visits all the tuples of the tuple list bound to the target channel and attempts to find a matching tuple. If the tuple is found, `MATCH` returns a reference array containing the substitutions. If the tuple to match didn't contain any variable definition, the array has length 0 and simply signals the success of the operation.
In case of success, the array of substitutions contains the references to the strings that should be substituted to the variable definitions that were present the tuple to match. The substitutions are in the *same order of appearance* of the variable definitions in the original tuple to match. For example:

- $(!a, const_1, !b, const_2, const_3)$ is the tuple to match.
- `MATCH` finds a matching tuple $\langle val_1, const_1, val_2, const_2, const_3 \rangle$.
- The array returned by `MATCH` is (val_1, val_2) and val_1 is substituted to a and val_2 is substituted to b.

If a matching tuple is not found, `MATCH` returns the `null` value.

`DMATCH` This instruction operates as `MATCH`. The only difference is that whenever a matching tuple is found, it is removed from the tuple list of the channel.

Global Constants Access

The instructions that enable the access to the global constant table are given below:

`CT_STORE` Stores at a given index (provided as an argument) the reference of a string in the global constant table. The reference is read from the operand stack.

`CT_LOAD` Loads on the operand stack a reference read from the global constant table. The index is given as an argument.

Thread Creation and Manipulation

`TH_NEW` This instruction allocates a new thread frame. The reference of the new thread is copied on the operand stack.

`TH_VT_STORE` This instruction stores a reference (of a string) in the variable table of the thread in execution. The variable table is implemented as a reference array. The index of the position in the variable table is the argument of this instruction.

`TH_VT_LOAD` This instruction loads on the operand stack a reference read from the variable table of the thread currently in execution. The index of the position in the variable table is provided as argument.

`TH_VT_STORE_IND` This instruction stores a reference (of a string) in the variable table of the thread whose reference is on the top of the operand stack. The index of the position in the variable table is the argument of this instruction.

`TH_GETCHAN` This instruction loads on the operand stack the content of the *channel* field of the thread frame in execution.

Garbage Collection

`GC_CHECK` This instruction forces the garbage collector to check whether some memory should be freed. This instruction is inserted at the beginning of each code chunk and its argument specifies the number of bytes of memory that the execution of the chunk would require in the worst case scenario. The garbage collector can check *before* the execution of the chunk if there is enough memory. If not, a garbage collection is triggered.

Miscellaneous

`NOP` The traditional "No Operation".

`DNOP` "Destructive" `NOP`. Same as `NOP`, with the difference that the top element of the operand stack is removed.

`SWAP` Swaps the two top elements of the operand stack.

`DUP` Pushes on the operand stack a copy of its top element.

`RNDSEQ` Fills the stack with a randomized sequence of values that span from 0 to n. n is given as a 2 bytes argument.

`CLEARSTACK` Empties the operand stack.

`ISSTACKEMPTY` Pushes $0x00000001$ on top of the operand stack if it is empty, otherwise it pushes `null`.

Scheduling Instructions

The instructions used for scheduling are presented below:

`SCHED_YELLOW` This instruction signals the scheduler that the current thread should be inserted in the YELLOW set, because it successfully executed the last chunk of code. The instruction also copies in the *program counter* field of the thread frame the absolute address of the beginning of the next chunk. The next time that the scheduler selects this thread frame the value of the program counter field is copied in the `PC` register.

`SCHED_TPL_YELLOW` This instruction is the same as the previous one with the only difference that it signals the scheduler that in the execution of the last chunk of code a tuple was output. The consequence of this additional information is that the threads in the RED set will be reconsidered by the scheduler for the execution.

`SCHED_RED` This instruction signals the scheduler that the current thread failed to execute the chunk of code. The thread is inserted into the RED set.

`SCHED_GREY` This instruction signals the scheduler that the thread has terminated the execution of the chunk but there are no more chunks to execute. The *status* field of the thread frame currently in execution is then changed to a flag value that indicates its termination and the frame is discarded by the scheduler.

`SCHED_TPL_GREY` The same as `SCHED_GREY`, with the difference that in the last chunk of code the thread outputs a tuple and the red threads must be rescheduled.

`SCHED_REP` Signals the scheduler that the current thread replicates.

5.5.3 AOP Support

The instructions presented in this section are exploited by the weaver to inject code whenever the aspects available in the aspect pool can trap an action. More about the usage of these instructions will be presented in Section 5.7.

Tags

Tag instructions are used to facilitate the work of the trapper, the component of the virtual machine that checks whether an action can be trapped by any aspect. Generally, they provide contextual information that can be used by the trapper to perform its duty in an efficient way.

`ASP_OUTACTION_TAG` This instruction specifies that the chunk being currently executed is the one of an *out* action. This instruction has the following arguments:

- one byte that specifies the number n of arguments of the action (not of the instruction).
- for each term of the tuple, 4 bytes are added:
  - the first encodes the type of the term: variable definition, variable or constant;
  - the second specifies the code (id) of the table where such term is stored. In Section 5.7 we will see what this means.
  - The last two bytes specify the index of the argument in the table.

- At the end of the tuple terms, one more 4-bytes-long argument, with the same format shown above, is inserted, specifying the target location for the action.
- two last bytes are used to specify the index of the global constants table where it is stored the string of the location to which the process that executes this action is bound to.

All these information is needed to attempt the trap as specified in Figure 2.11.

`ASP_INACTION_TAG` Equivalent of `ASP_OUTACTION_TAG` but for the *in* action.

`ASP_READACTION_TAG` Equivalent of `ASP_OUTACTION_TAG` but for the *read* action.

`ASP_STOPACTION_TAG` Tags the current chunk of code as *stop* action. This instruction has no arguments.

`ASP_GUARD_TAG` Tags the current chunk of code as *guard* action. This instruction has no arguments and it simply informs the weaver that this action is the guard in a sum of guarded processes. As we will see in Section 5.7, guards require special treatment.

Weaving

`ASP_PREACTIONS` This instruction marks the beginning of the code implementing the action and it marks where the pre-actions should be inserted. This instruction has one argument, whose purpose will be disclosed in Section 5.7.

`ASP_POSTACTIONS` This instruction marks the border between the action code and the beginning of the area executed in case of successful execution of the action (see Figure 5.1). Ideally, it is the place where the post-actions should be hooked. This instruction has no arguments.

`ASP_VT_STORE` This instruction stores a reference in the variable table that contains the variables defined in the woven code (stored within a code frame) being currently executed.

`ASP_VT_LOAD` Loads on the operand stack the value of a variable defined in the injected code being currently executed.

`ASP_CT_STORE` This instruction stores a reference in the constant table that contains the constants defined in the injected code being currently executed.

`ASP_CT_LOAD` Loads on the operand stack the value of a constant defined in the injected code being currently executed.

`ASP_SCHED_YELLOW` Equivalent to `SCHED_YELLOW`, but the argument is an offset (2 bytes) instead of an absolute address. Used within the injected code.

`ASP_SCHED_TPL_YELLOW` Equivalent to `SCHED_TPL_YELLOW`, but the argument is an offset (2 bytes) instead of an absolute address. Used within the injected code.

`ASP_CLEAN` When the thread abandons a code frame where the injected code was contained, this instruction is the last one to be executed and it makes sure that the link (if present) to this code frame from another code frame or from the original program is removed, thus allowing the garbage collector to recall the memory used by this frame. Moreover, this instruction moves the execution of the thread back to the code that originated the weaving, to continue the execution.

5.6 The AspectK Compiler

This section presents the procedure used to compile an AspectK source file into bytecode.

There are two compilers that can compile AspectK code:

1. The first one is provided as an external utility, named *aspectkc* and it compiles net specifications into object files with extension "ako" and the set of aspects into files with extension "as". This is the compiler that is discussed in this section;

2. The second compiler is embedded inside the virtual machine and it compiles the code that the weaver weaves into to the original program. This compiler is discussed in Section 5.7.3.

The AspectK compiler is structured as a chain of three components (Figure 5.9):

1. The *Parser*.

2. The *Compiler*.

3. The *Code Generator*.

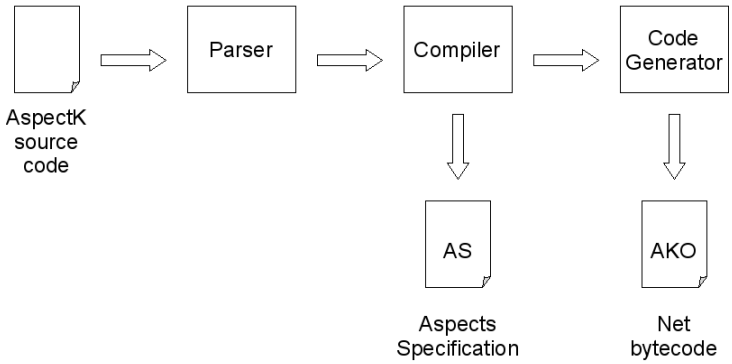


Figure 5.9: The AspectK compiler architecture.

5.6.1 The Parser

The parser is written using a parser generator for Java named *JavaCC* [17]. JavaCC generates top-down parsers and it outputs *LL(k)*-class grammars. JavaCC does not support left recursion, thus the the AspectK grammar specification had to be transformed into an equivalent one that had left recursion removed. The Net grammar of KLAIM of Figure 2.1 is transformed as shown in Figure 5.10.

$$
\begin{array}{lrcl}
N \in \mathsf{Net} & N & ::= & N_a \mid N_a || N \mid (N) \\
 & N_a & ::= & l :: P \mid l :: \langle \overrightarrow{l} \rangle \\
P \in \mathsf{Proc} & P & ::= & P_r \mid P_g \mid P_b \mid P_r|P \mid P_b|P \mid P_g|P \\
 & P_r & ::= & *P_d \mid *P_b \mid *P_r \\
 & P_g & ::= & P_d \mid P_d + P_g \\
 & P_d & ::= & a.P_d \mid a.P_b \mid a.P_r \\
 & P_b & ::= & (P) \mid 0 \\
a \in \mathsf{Act} & a & ::= & \mathbf{out}(\overrightarrow{\ell})@\ell \mid \mathbf{in}(\overrightarrow{\ell})@\ell \mid \mathbf{read}(\overrightarrow{\ell})@\ell \\
\ell, \ell^{\lambda} \in \mathsf{Loc} & \ell & ::= & u \mid l \qquad\qquad \ell^{\lambda} ::= \ell \mid !u
\end{array}
$$

Figure 5.10: KLAIM Nets and Processes syntax transformed to remove left recursion

The AspectK grammar of Figure 2.7 is modified into the grammar shown in

Figure 5.11.

$S \in \mathsf{System}$	S	$::=$	$\textbf{let}\ aspects\ \textbf{in}\ N$
	$aspects$	$::=$	$asp \mid asp, aspects$
$asp \in \mathsf{Asp}$	asp	$::=$	$A[cut] \stackrel{\triangle}{=} body$
$body \in \mathsf{Advice}$	$body$	$::=$	$\textbf{case}(cond)\ sbody;\ body \mid sbody$
	$sbody$	$::=$	$as\ \textbf{break} \mid as\ \textbf{proceed}\ as$
$as \in \mathsf{Act}^*$	as	$::=$	$a.as \mid \epsilon$
$cond \in \mathsf{BExp}$	$cond$	$::=$	$cond_b \mid cond_b \wedge cond$
	$cond_b$	$::=$	$\textbf{test}(\overrightarrow{\ell^\lambda})@\ell \mid \ell_1 = \ell_2 \mid \neg cond \mid (cond)$
$cut \in \mathsf{Cut}$	cut	$::=$	$\ell :: a$
$\ell^\lambda \in \mathsf{Loc}$	ℓ^λ	$::=$	$\ell \mid !u \mid ?u$

Figure 5.11: AspectK Syntax transformed to remove left recursion

The parser returns two data structures:

1. A tree datastructure representing the net, which is the same to the one described in Section 4.1.2.
2. A collection of objects representing the aspects. The collection of aspects can be output to a file at this phase already.

While reading the source code, the parser checks that the syntax respects the well formedness rules for nets and aspects, described in sections 2.1 and 2.2.

5.6.2 Compilation Scheme

The component coming after the parser is the compiler.
In the AspectK net we can find two class of components, the tuples and the processes. The compilation of the tuples is rather straight forward and it will be presented in Section 5.6.2.1. The compilation of the process is performed by a recursive routine, which transforms a process in a sequence of chunks of code. There are essentially three categories of chunks of code:

- The *action* chunk of code, which has the structure shown in Figure 5.1. Every action is transformed into this kind of chunk of code.

- The *operator* chunks of code, which result from the compilation of the two operators sum of guarded processes + and replication operator $*$.
- The *null process* 0 chunk.

The other operators are not compiled into separate chunks:

- the Net parallel composition || is not compiled at all.
- The sequential composition operator . can be identified into the code that links two different *action* chunks of code.
- The parallel composition of processes | operator splits the execution of one process into two or more paths. For each path a new thread is allocated: this is why this operator is basically compiled into code that instantiates new threads. This code may be present in three areas of the program:
  - the *initialization* area of the compiled program where the initial processes are created.
  - Within the *success* area of the chunk of code of an action. In cases like $a.(\ P \mid Q\ )$, after the action has been successfully executed two new parallel threads are created.
  - Within the code for the replication $*$ operator, in cases like $*(\ P \mid Q\ )$ where the process that replicate is a parallel composition of processes.

In order to clarify what discussed so far, we can see how the net reported in Figure 5.12 is transformed into chunks of code. When the compiler analyzes the tree derived from this net, it produces the chunks illustrated in Figure 5.13

```
loc1 :: <t1>
||
loc2 :: * out(t2)@loc1.( in(!a)@loc1.0 | read(!b)@loc1.0 )
||
loc3 :: out(t2)@loc1.0 + read(!b)@loc2.0
```

Figure 5.12: Example net.

The arrows indicate that the connected chunks are executed by the same thread. Whenever the thread has completed the execution of one chunk its program counter is moved to the initial address of the next pointed chunk.

The first block A, the *init* area, initializes the global constant table, allocates the tuples and allocates one thread frame for each process present in the beginning

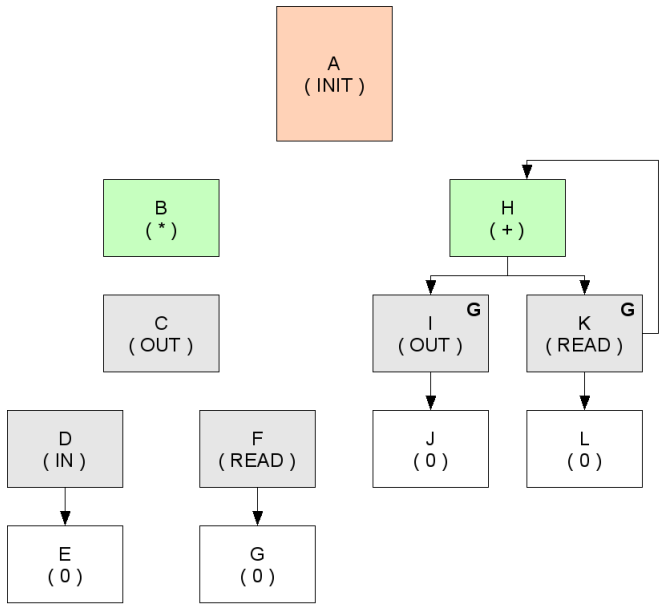


Figure 5.13: The program compiled into chunks.

of the execution (in the example there are 2). At the end of this portion of code, the instruction `SCHED_GREY` is executed and the control passes to the scheduler, which selects one of the available threads for the execution.

The first process is compiled into the following chunks:

- The replication chunk B. This code creates a new thread frame that points the block C. At the end of this block the `SCHED_REP` is present, and it reschedules the thread frame on this same chunk. Each time this chunk is executed, a new thread is spawned.

- Chunk C contains the code for the *out* action. The code placed in the "success" area of this chunk creates two thread frames for the processes starting at blocks D and F. At the end this chunk has a `SCHED_TPL_GREY` instruction, which moves the thread frame executing this chunk into the grey set (basically it terminates). This is the reason why no arrow originates from this block of code.

- chunks D and F are again two action chunks of code. At run time, they will be associated to two different thread frames, which will execute them at different times.

- chunks E and G are the "null process" chunks and they simply contain code that move the thread frames that are executing them into the grey set of the scheduler.

The second process starts with the block containing the code for the sum of guarded processes operator +. The operations performed by this portion of code are given in the sequel:

1. Chunk H selects one of the available paths (in the example there are 2).
2. The execution moves directly to that block, *without* scheduling the chunk first.
   - If the execution of the action succeeds (this happens for sure if block I is selected, because *out* actions never fail) the subsequent block is scheduled and the control is given to the scheduler.
   - If the execution of the action does not succeed (this may happen when K block is chosen) the program counter is moved back to chunk H where another path is selected.
3. If all the paths have been tested and all of them have failed, the thread is moved into the red set and the program counter of such thread is still pointing to the beginning of block H.

Blocks H, I and K can be seen as one big chunk of code, since blocks I and K are never scheduled singularly. The two blocks I and K are marked with "G" because their code is slightly different from the one of the non-guard actions.

In the next subsections we will see what the code of the single chunks contains. The code derived from the compilation of the net of Figure 5.12 will be used as an example[8]. The code is obtained with the *disassembler* utility provided along with the compiler. The instructions output by the disassembler have the format:

```
address: INSTRUCTION arguments
```

5.6.2.1 The Initialization Area

This code initializes three things.
First of all it fills the global constant table (Figure 5.14).

[8]The full code can be found in Appendix A

```
4: GC_CHECK 155
9: NEWSTRING "t2"
13: CT_STORE 3
16: NEWSTRING "t1"
20: CT_STORE 1
23: NEWSTRING "loc1"
29: CT_STORE 0
32: NEWSTRING "loc3"
38: CT_STORE 4
41: NEWSTRING "loc2"
47: CT_STORE 2
```

Figure 5.14: Init area: allocation of the strings and initialization of the global constant table.

The strings are allocated in memory and stored in predefined indexes. Secondly, once that the constants are available in the memory, the tuples are allocated (Figure 5.15)

```
50: CT_LOAD 0
53: DUP
54: GETCHAN
55: JMP_NOTNULL 7
58: NEWCHAN
59: JMP_OFF 4
62: GETCHAN
63: NEWARRAY 1
68: DUP
69: CT_LOAD 1
72: ASTORE 0
77: APPEND
```

Figure 5.15: Init area: allocation of the tuple `loc1::<t1>` .

The location/channel to which each tuple is bound to is retrieved and if it does not exist it is created (instruction `NEWCHAN` in the code). Then the reference array representing the tuple is allocated and the constants (in the example only one) are copied in the array. Finally, the tuple is appended on the tuple list of the channel.
The last task accomplished by the initialization area is to create the thread frames for the processes that are present in the net (Figure 5.16).

Similarly to what happens with the tuples, for processes too the channel is first

```
78: CT_LOAD 2
81: DUP
82: GETCHAN
83: DUP
84: JMP_NOTNULL 5
87: SWAP
88: NEWCHAN
89: TH_NEW vt size:0 ref:119
98: CT_LOAD 4
101: DUP
102: GETCHAN
103: DUP
104: JMP_NOTNULL 5
107: SWAP
108: NEWCHAN
109: TH_NEW vt size:1 ref:351
118: SCHED_GREY
```

Figure 5.16: Init area: allocation of the two processes.

retrieved and if it does not yet exist it is created. Then, a new thread frame bound to the channel is allocated.

5.6.2.2 The Replication Chunk

An example of replication chunk is shown in Figure 5.17.

```
119: GC_CHECK 31
124: TH_GETCHAN
125: TH_NEW vt size:0 ref:135
134: SCHED_REP
```

Figure 5.17: Replication chunk.

The code of this chunk simply creates a new thread frame for each process it spans over — if the parallel composition operator is present, there can be two or more.

```
356: CLEARSTACK
357: RNDSEQ 2
360: ISSTACKEMPTY
361: JMP_NULL 4
364: SCHED_RED
365: SWITCH
    2 choices
    6
    60
```

Figure 5.18: Sum of guarded processes chunk.

5.6.2.3 The Sum of Guarded Processes Chunk

This chunk works as follows (Figure 5.18):

1. It clears the operand stack.

2. A randomized sequence of n numbers $0, \cdots, n$, pairwise different, is pushed into the stack. n is the number of guarded actions[9].

3. `SWITCH` pops the top of the stack and uses it to select one of the offsets it has as arguments. It jumps to the given address, where one of the possible guarded actions is located. It does so until either one of the guards is successfully executed or the operand stack is emptied, which is equivalent to having unsuccessfully attempted all the possible paths.

5.6.2.4 The Null Process Chunk

The null process chunk is very simple, it consists of the single `SCHED_GREY` instruction. The code from the example is shown in Figure 5.19.

```
350: SCHED_GREY
```

Figure 5.19: The Null Process chunk.

[9]There can be at most 256 guarded actions, because the number of arguments allowed by `SWITCH` is at most 256.

5.6.2.5 The OUT Action Chunk

The *out* action chunk (Figure 5.20) has the structure outlined in Figure 5.1.

```
135: GC_CHECK 83
140: ASP_OUTACTION_TAG
    1 arguments
    c 3
152: ASP_PREACTIONS 0
157: CT_LOAD 0
160: DUP
161: GETCHAN
162: DUP
163: JMP_NOTNULL 5
166: SWAP
167: NEWCHAN
168: NEWARRAY 1
173: DUP
174: CT_LOAD 3
177: ASTORE 0
182: APPEND
183: ASP_POSTACTIONS
184: TH_GETCHAN
185: TH_NEW vt size:1 ref:205
194: TH_GETCHAN
195: TH_NEW vt size:1 ref:278
204: SCHED_TPL_GREY
```

Figure 5.20: The *out* action chunk.

The first area (in the example from address 140 to 183 both included) contains the code that implements the action. First we find the tags, which will be used by the weaver. They record the type of the action, the type of its arguments, the type of its target location and the name of the location that the process executing the action is bound to. Then following steps are carried out:

- The channel of the target location is retrieved or created.
- The tuple is created.
- The tuple is appended at the channel's tuple list.

After the action code, comes the "success" area, which is executed in case the

action succeeds (this is always the case for *out*). In the example this section creates (and thus schedules) two threads and then the thread terminates (it signals the scheduler to be moved in the grey set).
The out action has no "failure" section.

5.6.2.6 The IN and READ Action Chunks

The *in* and the *read* action chunks have both the structure of the chunk shown in Figure 5.1. These two actions are very similar, they differ only in the fact that *in* removes the matched tuple from the environment and *read* does not.
In Figure 5.21 we can see an example of *in* action.

```
205: GC_CHECK 9
210: ASP_INACTION_TAG
    1 arguments
    vd 0
222: ASP_PREACTIONS 0
227: CT_LOAD 0
230: GETCHAN
231: DUP
232: JMP_NOTNULL 7
235: DNOP
236: JMP_OFF 40
239: NEWARRAY 1
244: DUP
245: TH_VT_LOAD 0
248: ASTORE 0
253: DMATCH
254: DUP
255: JMP_NOTNULL 7
258: DNOP
259: JMP_OFF 17
262: ALOAD 0
267: TH_VT_STORE 0
270: ASP_POSTACTIONS
271: SCHED_YELLOW 277
276: SCHED_RED
```

Figure 5.21: The *in* action chunk.

The initial part of the "action" area is similar to the one of the other two actions.

In fact, first we can see that the compiler inserts the tags describing the action and then the code to retrieve the channel of the target location.
Then the tuple to match is created and the match against the tuples present at the target location is attempted. The `DMATCH` instruction removes the tuple from the channel, if the match is found, whilst the `MATCH` instruction (used in *read*) does not. If a match is found, the returned substitutions are stored in the thread's variable table.
In the example of Figure 5.21 the chunk has only one instruction in both the "success" area (address 271) and in the "failure" area (address 276).

```
425: ASP_GUARD_TAG 360
430: ASP_READACTION_TAG
    1 arguments
    vd 0
442: ASP_PREACTIONS 0
447: CT_LOAD 2
450: GETCHAN
451: DUP
452: JMP_NOTNULL 7
455: DNOP
456: JMP_OFF 40
459: NEWARRAY 1
464: DUP
465: TH_VT_LOAD 0
468: ASTORE 0
473: MATCH
474: DUP
475: JMP_NOTNULL 7
478: DNOP
479: JMP_OFF 17
482: ALOAD 0
487: TH_VT_STORE 0
490: ASP_POSTACTIONS
491: SCHED_YELLOW 502
496: JMP_ABS 360
```

Figure 5.22: The *read* action chunk. The action is a guard.

Figure 5.22 shows the chunk of code for the *read* action. The interesting details of this example reside in the fact that this action is a *guard*. There are some differences in the code of a guard action respect to the code of a non-guard action:

- There is one more tag instruction, `ASP_GUARD_TAG`.
- The "failure" area contains the jump instruction `JMP_ABS` (it jumps to the beginning of the + chunk) instead of the `SCHED_RED` instruction.

5.6.2.7 The STOP Action Chunk

The *stop* action chunk is outlined in Figure 5.23.

```
ASP_STOPACTION_TAG
SCHED_GREY
```

Figure 5.23: The *stop* action chunk.

The tag is inserted only to differentiate the code of the action from the code of the null process (for presentation purposes).

5.6.3 The Code Generator

After the compiler has compiled the program into code chunks, the code generator merges and streamlines them (this is shown in Figure 5.24).

The different parts of code are linked using labels, that now need to be transformed into addresses. The code generator scans the code and resolves the address of each label by computing the length of each instruction. Then, using this information, it scans again the code and it generates the binary code (bytecode). Every time a label is encountered, it is substituted with an address or with an offset, depending on the instructions.
After the code generator has finished its task, the code is output into a file that can be executed by the virtual machine.

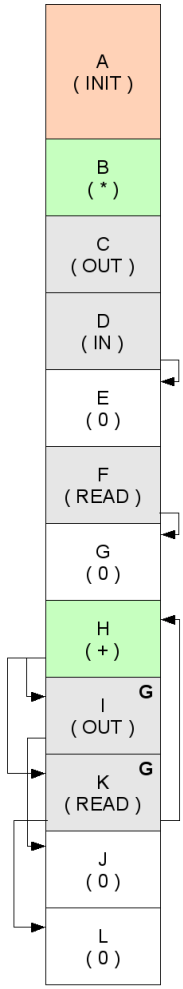


Figure 5.24: The chunks are serialized.

5.7 The Weaver

In this section we will analyze how the process of weaving of new code is performed in the AspectK virtual machine.
The weaver supports three different techniques for weaving:

- *Lazy Weaving.* This technique implements the just-in-time weaving technique, namely the trap is attempted only when the process is about to execute the action. This technique corresponds to the one specified in the AspectK semantics of Section 2.2.3. The lazy weaving was implemented in the interpreter too (see Section 4.4.1).
- *Greedy Weaving.* This weaving technique is similar to the lazy one. The difference resides in the fact that whenever an action is successfully trapped and some pre-actions or post-action are injected, the weaver recursively attempts the trap on the injected actions before injecting them.
- *Balanced Weaving.* This weaving technique attempts to optimize the greedy weaving by extending it with *dynamic* weaving techniques. The implementation of the dynamic weaving in the virtual machine is slightly different from the one implemented in the interpreter. In the virtual machine the balanced weaving can save to the virtual machine the overhead of trapping certain action. This technique extends the *greedy* weaving and it will be explained in detail in Section 5.8.3.

The differences between these weaving techniques reside mainly in the way the actions are trapped and in "when" the weaver is invoked by the virtual machine. However, all the weavers use the same infrastructure to weave the code into the original program.

The weaving process, in general, can be summarized in the following steps:

1. The weaver traps an action, and we assume that the trapping returns some pre-actions and some post-actions.
2. The weaver uses the JIT compiler (Section 5.7.3) to transform these actions into code.
3. A new *code frame* (see Section 5.3) is allocated and the code is copied into it.
4. Depending on the chosen weaving technique, either the execution is moved to the code contained in the frame or the injected code is simply linked to the original program.

An example that demonstrates these concepts is shown in Figure 5.25.

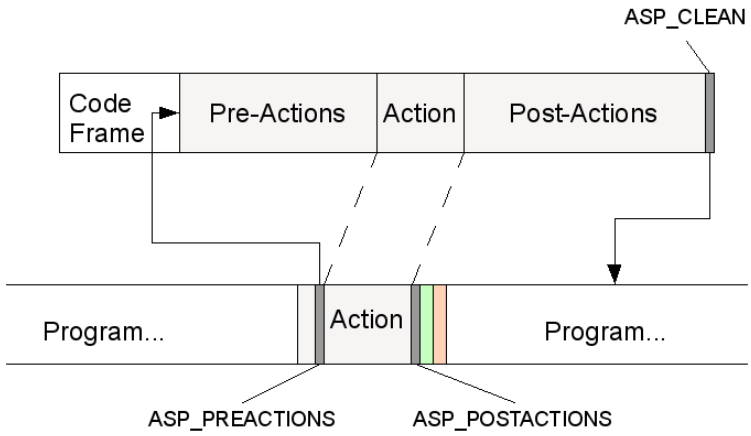


Figure 5.25: An action is trapped and as a result some code is hooked to the original program.

The tag instructions, present in the beginning of the "action" code, provide the trapper with the necessary information for performing the trap, namely:

- type of the action;
- number and type (variable definition, variable, constant) of the arguments;
- target location;
- location that the process executing the action is bound to.

Then, when the instruction `ASP_PREACTIONS` is encountered, the virtual machine triggers the weaver, which attempts the trap.

NOTE: there can be multiple "levels" of code frames: if one of the pre-actions of the post-actions contained into a code frame is trapped, a new code frame is allocated and it is linked to it.

NOTE(2): the trapped action, whenever it is included in the injected code, it is re-compiled *without* the `ASP_PREACTIONS` and `ASP_POSTACTIONS` instructions. This prevents the action from being trapped again, because the virtual machine triggers the weaver whenever it encounters an `ASP_PREACTIONS` action. The tag instructions that signal the type of the action (i.e. `ASP_OUTACTION_TAG`) could

be removed too, but they are kept for facilitate the recognition of the type of action to the memory analyzers (used in the GUI, for example).

The last instruction being executed in the code placed in the code frame is `ASP_CLEAN` which performs two operations:

1. if its argument is different from `null` it moves the program counter to the parent frame (which can be another code frame or the original program) otherwise the instruction performs the same operations as `SCHED_GREY`.
2. if the virtual machine is using the *balanced* weaver, the code frame may be linked by the argument of the `ASP_PREACTIONS` instruction. If so, `ASP_CLEAN` removes such link, enabling the garbage collector to reclaim the memory used by this code frame, the next time it is executed.

The original program physically links the injected code frame only in certain cases in the *balanced* strategy, when a reference of the code frame is stored as the argument of the `ASP_PREACTIONS` instruction. In the other weaving strategies the injection is performed when the action is trapped and there is no need to physically link the frame: it is just necessary to move the execution to the code it contains.
Conversely, the code contained into the code frame links to the parent code (which can be either the code of another code frame of the code of the program) with the `ASP_CLEAN` instruction.
The next section gives an overview of the possible cases that may arise in the trapping and injection of code and illustrates how the weaving is actually performed.

5.7.1 Weaving and Linking the Code

After the function Φ_f[10] of Figure 2.10 has been executed there are six possible alternatives for the outcome:

- no pre-actions, proceed, no post-actions;
- no pre-actions, break, no post-actions;
- pre-actions, proceed, no post-actions;

[10] In the code, the function Φ_f corresponds to the invocation of the method `tryTrap_JIT()` in the trapper implementations. There also exists a `tryTrap_Dynamic()` which, as the name suggests, attempts the dynamic version of the trap (see section5.8.3 for more information). The outcome of a trap is enclosed in the `vm.aop.TrapOutput` object.

- pre-actions, break, no post-actions;
- no pre-actions, proceed, post-actions;
- pre-actions, proceed, post-actions;

The cases where *break* is present cannot have post-actions, because the execution is interrupted by break. The number of cases is doubled when we realize that the actions that are *guards* need a different treatment from the non-guard ones. This makes a total of twelve cases , but some of the cases can be grouped together.
We analyze now each single case. In the figures the expression `ASP_CLEAN(0)` means that the argument of such instruction is `null`.

5.7.1.1 No pre-actions, proceed, no post-actions, any action

The execution is not modified.

5.7.1.2 No pre-actions, break, no post-actions, any action

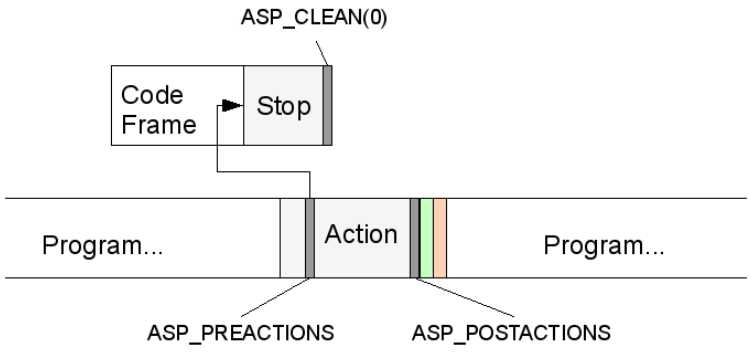


Figure 5.26: No pre-actions, break, no post-actions, any action.

The code frame (Figure 5.26) contains the *stop* action. The woven code remains the same whether the trapped action is a guard or not.

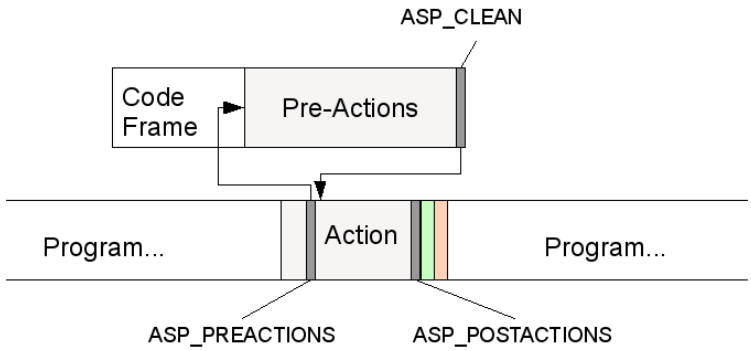


Figure 5.27: Pre-actions, proceed, no post-actions, non-guard action.

5.7.1.3 Pre-actions, proceed, no post-actions, non-guard action

The code frame contains the pre-actions code (Figure 5.27).
The last instruction executed in the frame is `ASP_CLEAN`, which jumps back to the code of the parent frame. There, the execution proceeds from the first instruction *after* `ASP_PREACTIONS`.

5.7.1.4 Pre-actions, proceed, no post-actions, guard action

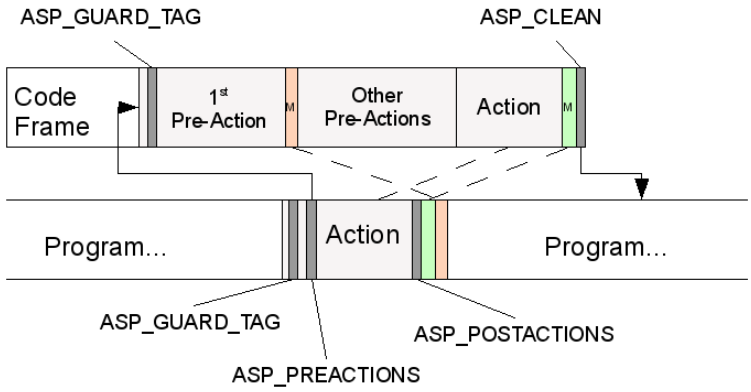


Figure 5.28: Pre-actions, proceed, no post-actions, guard action.

Figure 5.28 illustrates this case, which differs from the previous one only in the fact that the trapped action is a guard. However, the fact that the trapped

action is a guard requires some special measures. First of all, the first pre-action that is injected will be the new guard. This implies that:

1. it must contain the `ASP_GUARD_TAG`, in case this action is trapped in the future.
2. In the "failure" section the execution must jump back to the first instruction of the sum of guarded processes + chunk of code. This is achieved by using the `ASP_CLEAN` instruction, which whenever the argument is different from the `null` value it has the behavior of a jump to an absolute address (like `JMP_ABS`) besides removing the code frame it is called from.

Moreover, the trapped action needs to be changed too, since from now on it won't be a guard anymore. The action is then re-compiled without the `ASP_*` tags and the following changes are made to its "success" and "failure" sections:

- The "success" section is copied from the trapped action, but the last instruction which would pass the control to the scheduler is changed to either `ASP_SCHED_YELLOW` or `ASP_SCHED_TPL_YELLOW` which schedules the next chunk of code on the code frame, that contains only `ASP_CLEAN`. The argument of `ASP_CLEAN` contains the address that was stored in the trapped action's scheduling instruction.
- The "failure" area (unless the action is not an *out* action, which doesn't have a "failure" section) is changed to the `SCHED_RED` instruction.

5.7.1.5 Pre-actions, break, no post-actions, non-guard action

The code frame (Figure 5.29) contains two portions of code:

- the pre-actions code.
- the code for the *stop* action.

5.7.1.6 Pre-actions, break, no post-actions, guard action

This case (Figure 5.30) is very much similar to the one outlined in Section 5.7.1.4. However, in this case, after the pre-actions we do not find the code of the recompiled trapped-action bu instead a *stop* action.

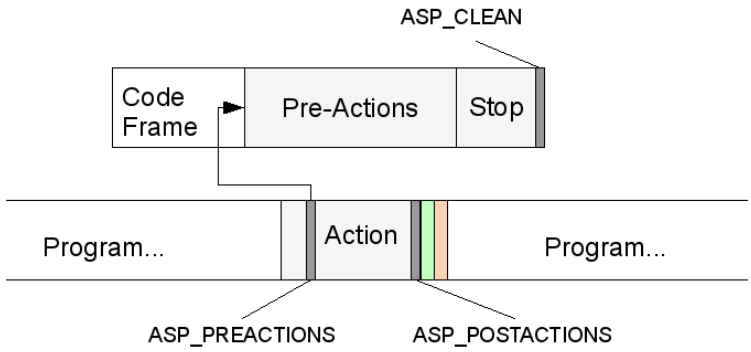


Figure 5.29: Pre-actions, break, no post-actions, non-guard action.

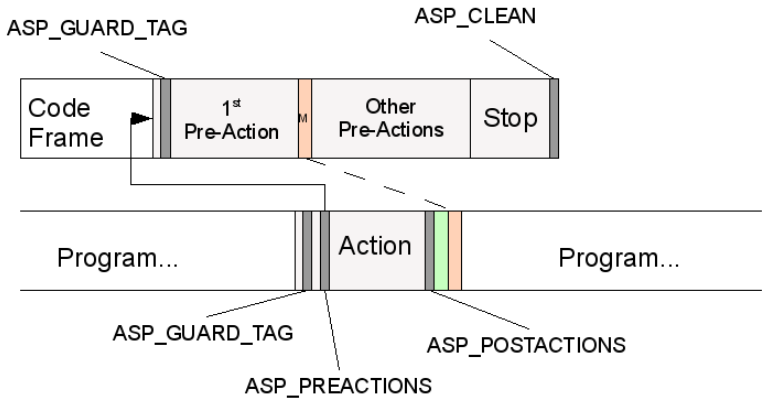


Figure 5.30: Pre-actions, break, no post-actions, guard action.

5.7.1.7 Pre-actions, proceed, post-actions, non-guard action

In this case (Figure 5.31) the code frame contains the pre-actions, the code of the recompiled trapped-action and then the post-actions.
The "success" section of the trapped action is copied in the "success" area of the last post-action. The last instruction of the copied "success" section is modified into a scheduling instruction that schedules the next chunk on the code frame, which contains the `ASP_CLEAN` instruction only.

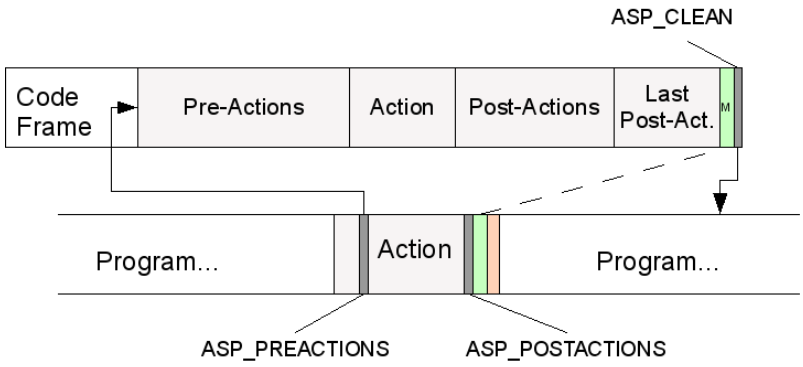


Figure 5.31: Pre-actions, proceed, post-actions, non-guard action.

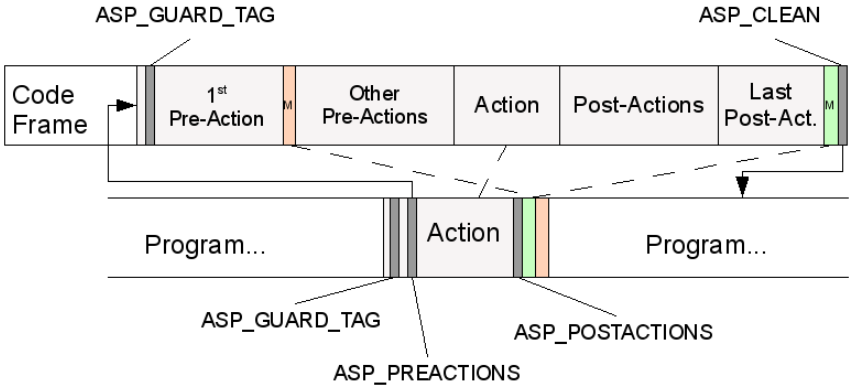


Figure 5.32: Pre-actions, proceed, post-actions, guard action.

5.7.1.8 Pre-actions, proceed, post-actions, guard action

This case (Figure 5.32) does not contain anything new, and it is simply a composition of cases already presented.

5.7.1.9 No Pre-actions, proceed, post-actions, non-guard action

In this case (Figure 5.33) the code frame contains the code of the re-compiled trapped action and the code of the post-actions.
The "success" area of the last post-action contains the "success" area of the

trapped action. The scheduling instruction in the copied "success" area is modified into one that schedules the chunk of code represented by the `ASP_CLEAN` instruction.

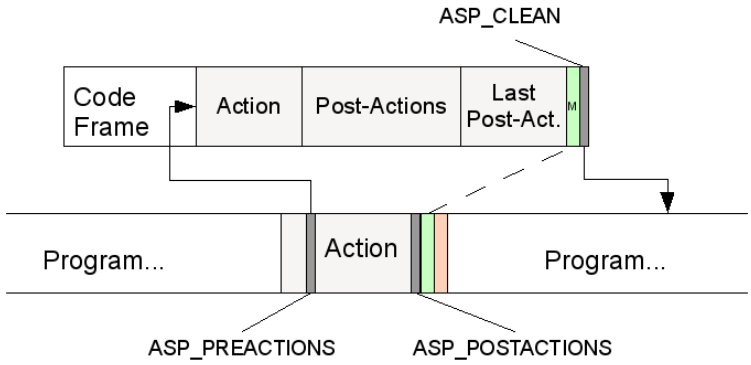


Figure 5.33: No Pre-actions, proceed, post-actions, non-guard action.

5.7.1.10 No Pre-actions, proceed, post-actions, guard action

This case (Figure 5.34) is fairly similar to the previous one.
The only real difference is in the "failure" section of the recompiled version of the trapped action is changed from the jump to an absolute address to `ASP_CLEAN`. This ensures that if the action fails to execute, the execution moves to the guarded-processes sum chunk of code and at the same time the code frame is removed.

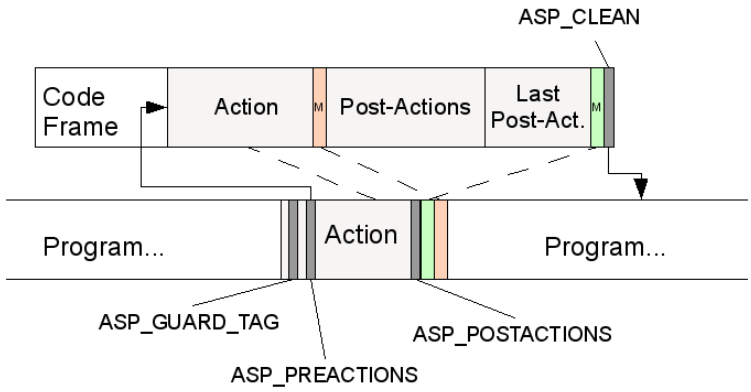


Figure 5.34: No Pre-actions, proceed, post-actions, guard action.

5.7.2 Managing Variables and Constants Introduced by the Woven Code

Sometimes the actions woven in the code introduce new variables and new constants.
The example reported below introduces 2 new constants (`const1` and `const2`) and 2 new variables (`w` and `e`).

```
A[User :: out(x)@y] ^=
    out(const1)@x.read(y,!w,!e)@DB proceed out(const2)@w
```

One possibility for handling these new temporary constants and variables could be to extend the existing global constant table and the variable table of the thread executing the code.
However, this is not an efficient solution, for several reasons:

- the global constant table and the variable table have fixed size. Extending a table to include new values implies the allocation of a new table and the copy in the new table of all the old values.
- If actions containing new constants and new variables are woven in the code often (which can easily be the case) the operation described in the previous point would be performed often too, and this would greatly affect the performances of the system and the memory would be filled quite soon.

Things are even more complicated when we think that the injected actions can be trapped too, and this may introduce even more temporary symbols.
A more efficient solution for the storage of these values is provided in the sequel. For the sake of clarity the discussion is presented for the temporary variables, but it is valid for the temporary constants too.

The overall idea is to associate a *new table* that contains the temporary variables *each time* the thread moves to a new code frame allocated by the weaver. So:

- Every time the execution moves to the the code contained in a code frame, a new table for the temporary variables is created. If there are no temporary variables, the table is not allocated and it is substituted by the `null` value.
- The (possibly `null`) reference to this new table is saved in a second table, which is defined as *aspect variable table table*, namely the table that

contains reference to the aspect variable tables. This table contains the references to the temporary variable tables created for the code frames. Moreover, the *size* of this table is always equal to the number of connected code frames that have been encountered by the thread so far.

For example we can imagine that the action of the original program is successfully trapped and a code frame with three pre-actions compiled in it is created. The pre-actions define some temporary variables. The *aspect variable table table* has size 1 and it contains the reference to another table that holds the values of the temporary variables.
Then one of the pre-actions is trapped and this leads to a new code frame. A new *aspect variable table table* for the thread is created, and it substitutes the previous one. Its first cell stores the content of the old table's first cell. The second cell contains the reference to a new table for the temporary variables of the second frame.
And so on.
When the thread moves from the second code frame back to the first one, a new *aspect variable table table* of size 1 is allocated. Its only cell contains the reference to the temporary variable table associated to the first frame.

The instructions that can access these tables are `ASP_VT_LOAD`, `ASP_VT_STORE`, `ASP_CT_LOAD` and `ASP_CT_STORE`. They are given two arguments:

- The *id* of the table, which is a number comprised between 1 and n, where n is the number of code frames stacked one on top of each other. In the previous example, when there were two code frames n was 2. The global constant table and the variable table of the thread are associated to the id *0*.
- The index of the constant or variable on the table.

5.7.3 The Just-In-Time Compiler

The Just-In-Time compiler is a compiler embodied in the virtual machine.
This compiler is a reduced version of the compiler described in Section 5.6. It is used by the weaver to compile the sequences of pre-actions and post-actions that the trapper creates whenever the trap of an action succeeds.
This compiler supports only the compilation of *sequences* of actions, because this is the only type of code that can be injected in AspectK.
The actions are compiled in the same way as in the stand-alone compiler, with some minor differences:

- The SCHED_YELLOW and SCHED_TPL_YELLOW instructions are substituted by ASP_SCHED_YELLOW and ASP_SCHED_TPL_YELLOW. These instructions use offsets to point the next chunk of code. The original instructions cannot be used because if the code frame is re-located by the garbage collector the addresses become invalid.

- The code created by this compiler can contain the instructions ASP_VT_LOAD, ASP_VT_STORE, ASP_CT_LOAD and ASP_CT_STORE which are used to access the temporary variables and constants introduced by the injected code.

- If temporary constants are present, the initial part of the injected code is used to initialize them.

- It is possible to avoid the compilation of the tags in the woven actions. This is useful when the trapped action is recompiled and inserted in the code frame — it is an underlined action — and it is also useful whenever the *greedy* trapping is used (see Section 5.8.2).

5.7.4 The Condition Evaluator

This component is used to evaluate the conditions within the conditional advices. The condition evaluator supports backtracking. Backtracking is useful because of the presence of `test` conditions, which may bind variables that may be used in the rest of the condition.
The condition evaluator tries one combination of bindings for the variables. If the condition is true for such bindings, no more combinations need to be attempted. Otherwise, the condition evaluator backtracks to find other bindings and attempt new combinations. If no new combinations are possible, the condition evaluates to false.

For example:

$$test(!a, const1)@loc1 \wedge a = const2$$

The *test* condition may evaluate to true for more than just one tuple.
If the equality condition $a = const2$ evaluates to *false* for a certain tuple matched by test, another tuple is tried and the condition is evaluated again.
The process continues until either the condition becomes *true* or *test* cannot match any other tuple and the condition is then evaluated to *false*.

5.8 Weaving Techniques

This section presents the three weaving strategies supported by the virtual machine.
The weaver, no matter what strategy is chosen, is activated by mean of the tag bytecodes:

1. If the `ASP_GUARD_TAG` is encountered, the virtual machine sets a flag within the weaver that signals that the current action is a guard.
2. When the `ASP_*ACTION_TAG` is encountered, the virtual machine sets within the weaver the type of the action, the number and the type (constant, variable or variable definition) of its arguments along with the necessary information to retrieve them, the *target* location and the location that the process executing this action is located at.
3. When the `ASP_PREACTIONS` is encountered, the virtual machine finally invokes the weaver.

5.8.1 Lazy Weaving

The *lazy* weaving approach implements the JIT weaving defined by the AspectK semantics.
The weaver, first of all, attempts a trap on the action, using the information provided by the tags. If the trap succeeds, the weaver uses the JIT compiler to compile the new code and then it copies it into a new code frame allocated on the heap.
The content of the program counter is set to the address of the first instruction of the injected code and the execution continues from there. The injected code is not physically *linked* to the original code.
If the trap does not succeed, the execution continues with the instruction coming after `ASP_PREACTIONS`.

The lazy weaving approach has some advantages:

- It uses a limited quantity of memory, since the weaving and the consequent allocation of code frames happen *only* when ultimately needed. Moreover, the trap is not directly attempted on the injected actions, and this fact keeps the size of the injected code frame rather small, if compared with the *greedy* weaving.

- This weaving technique is very convenient if the aspect pool Γ_A is expected to change frequently because the changes are immediately effective on all the actions that are trapped right then. This may be not the case in the *greedy* weaving, where some actions are compiled without tags or in the *balanced* weaving which is based on the *greedy* weaving.

On the other side, the major disadvantage of this approach is that the trap and the weaving operations can be quite demanding in terms of performances and in general it is not possible to reuse the injected code. All the computation is carried out at the last moment, when the action is about to be executed. The other two strategies try to produce some improvements in this area.

5.8.2 Greedy Weaving

The *greedy* weaving differs from the lazy weaving in the way that the trapping is performed.
The greedy trapper, in fact, does not stop with the trapping of the single action, but it recursively attempts the trap operation on the injected actions before returning the definitive set of pre-actions and post-actions. In general, we can expect four possible outcomes when the trap is attempted on one of the injected actions by a recursive call of the greedy trapper:

1. The trap succeeds and more actions are injected.
2. The trap fails because no cut matches the action.
3. The trap fails because the action being trapped uses some temporary variables whose value is undefined at the moment of the trap.
4. The trap fails because the aspect contains some conditional advices. In order to ensure the accuracy of the weaving, these must be evaluated at run time.

In the first case the trapper recursively attempts the trap on the newly injected actions.
In the second case the action is skipped.
In the third and fourth cases the trap on the injected action must be re-attempted when the execution will reach it.

The actions falling into the second category are compiled *without* the AOP tags, because with the current state of the aspect set Γ_A no aspect can trap them

and then it gives no benefit attempting the trap on them again. Of course there is a chance that the set of aspects may change just after the actions have been compiled and injected. If it must be guaranteed that any action at any time is trapped or re-trapped if the aspect pool is changed, then this weaving technique may be not appropriate.

The greedy trapper is vulnerable to recursive cycles, namely situations like the one described below:

- a first aspect traps one action and weaves some other actions;
- a second aspect traps actions woven by the first one and weaves other actions that the first one can trap.

There is no mechanism to detect such situations, however the trapper poses a limit to the depth of the recursion tree, in order to avoid infinite recursion.

In the best cases, the greedy weaving greatly reduces the number of code frames that are allocated, since it favors the injection of bigger portions of code respect to the lazy technique. This fact may produce important benefits in terms of performances. However this technique has some disadvantages:

- we already discussed the fact that it is less responsive to changes operated on the aspects pool Γ_A. Some actions may be compiled without tags, hence even if the set of aspects is modified the trap is not attempted on these actions.
- it may require a even bigger computational effort in the trap and weaving phase. If no injected action is trapped, what remains is only overhead and no benefit

In conclusion, the greedy weaving provides some advantages over the lazy weaving in those cases where the majority of aspects contain non-conditional advices, thus providing more chances that the injected actions are trapped too.

5.8.3 Balanced Weaving

The *balanced* weaving is a third approach for weaving. In the interpreter the *balanced* solution is simply an optimization: in fact, whenever the aspect pool is modified, the net tree is traversed and the *dynamic* trapping is attempted

on the actions. If the trapping *for the single aspect* succeeds, the substitutions returned by *check* are cached, for being used when the action is encountered and just-in-time trapped during the execution.
The virtual machine implementation of the *balanced* weaving, instead, is more evoluted. Whenever the aspect pool is modified[11], the program is scanned and the *dynamic* version[12] of the *greedy* trap is attempted on every action. There are three possible outcomes:

- The trap succeeds.
- The trap fails because no aspect can trap the action.
- The trap fails because run time knowledge is required, namely:
  - The cut of one or more of the aspects requires the knowledge of the value of one or more variables.
  - One or more of the aspects contain conditional advices.

In the third case there is not much that can be done, since the no weaving can be attempted until the needed run time information is available.
In the first case the weaver allocates a *static* code frame in the heap, that can be used and reused until the Γ_A set does not change. In fact, the code contained in static code frames is derived from aspects that do not require any run time condition or information. However, when the Γ_A set is changed the code contained in the static code frames may be not valid, since some new aspects might have been introduced.
Then the weaver scans the entire program again and creates, when possible, new static code frames. Frames that are being executed by threads cannot be removed and recomputed, they must be left untouched in order to avoid the corruption of the execution process.

Static frames need to be distinguished from the frames allocated and used only once. Frames contain a special flag-field named "static flag" signaling whether they are static or not. Moreover, code frames store in the field "freshness" a number, which is used to determine whether a static frame is *updated* to the current configuration of the aspect pool. This number is indeed obtained from the *aspect pool*. Whenever the *aspect pool* is modified, the "freshness" of the

[11] The aspects may be added into the virtual machine at *any* time. However, the information needed by the weaver to check if the cuts match a certain joinpoint is available in memory only *after* the execution of the initialization area of the program. For this reason, the dynamic weaving is always delayed by the virtual machine until the initialization part of the program has been executed.

[12] With *dynamic version* we mean an implementation containing functions that have the properties of the prototypes outlined in section 2.3.

aspect pool is modified too.
When a thread encounters a static frame, it checks if the value contained in the aspect pool and the one contained in the code frame differ: if so the code frame must be removed and the trap re-attempted.
Whenever an action is trapped and a static frame is created, its reference is stored as an argument of the `ASP_PREACTIONS` instructions. In this way any further execution of the same action reuses the same code frame.

Static frames can be used not only for actions successfully trapped at dynamic time, but also for actions which *failed* to be trapped because no aspect could trap them (and not because run time information was required).
In these cases the weaver allocates and links an *empty static code frame*, which is used as a "flag" that signals that with the current state of the aspect pool the flagged action is not trapped by any aspect.

Whenever a thread executes the `ASP_PREACTIONS` one of the following possible choices is taken:

- If its argument is `null`, the weaver is invoked and it attempts a JIT (greedy) trap.
- Otherwise, the argument is the address of a static code frame:
  - If the static code frame is outdated, namely its freshness value is different from the one of the aspect pool, the *dynamic* trap is attempted again. If it succeeds, the new static code frame is linked to the code and the execution continues on the code it contains.
    If it fails, a JIT (greedy) trap is attempted and if it succeeds (and a code frame is allocated) the execution continues on the code it contains. In this case, the code frame created is not static, thus it is not linked to the original program.
  - If the static code frame is updated and it is a non-empty code frame, the execution moves on the code it contains. Otherwise, if the code frame is empty, the execution simply continues with the next instruction.

The balanced weaving is an optimization of the greedy weaving technique that exploits the *dynamic* weaving of aspects rather than limiting itself to the just-in-time solution. In fact it provides better performances in those cases where the aspects contain a non-conditional advice and the aspect pool is not modified frequently. In the best case the weaving is purely dynamic and no trapping is performed during the execution, because all the injected actions are compiled without tags. But if the aspects contain conditional advices or the aspect pool is modified frequently the overhead may become unacceptable.

5.9 Garbage Collection

The execution of processes leads to the creation of a large number of frames. Some of these frames are persistent, namely they are used throughout the entire evolution of the net. Some other frames, instead, have a transient life, like

- the reference frames created for the execution of `MATCH` and `DMATCH` instructions.
- Thread frames of threads not active anymore (in the *grey* set).
- Frames of tuples that have been removed by an *in* action.
- Code frames containing code that was injected by the weaver and that now is not required anymore.

The *garbage collector* is the component of the virtual machine which takes care of recalling the memory which is used by dead frames.
It adopts a *moving mark-and-sweep* strategy. The process of collection operates as follows:

1. All the frames that are reachable from active memory are marked (a bit of the descriptor is set).
2. The heap is scanned and all the non-marked frames are removed.
3. The heap is compacted.

The garbage collector can be executed only in the window of time that intercurs between the execution of two chunks of code. That is the only moment when it is guaranteed that the operand stack does not contain information useful during the execution and that would be otherwise lost if the garbage collection were triggered at a different time.
The `GC_CHECK` instruction is used to achieve this goal. It is placed by the compiler at the beginning of every chunk and its argument contains an over-estimation of the number of bytes required for the execution of the chunk it tops. Whenever `GC_CHECK` is executed, the virtual machine checks whether there is enough memory available to execute the chunk. If not, the garbage collector is executed.
This dedicated instruction also makes it more efficient to test the necessity of garbage collection, because this task is carried out only at certain moments of the execution and not for example after every instruction.

However this solution is not perfect, because it is not possible to predict at compile time the memory used by the weaver when it injects code. Moreover, for the reasons explained before, the weaver *cannot* activate the garbage collector but it can only force its execution when the next `GC_CHECK` instruction will be encountered. In order to mitigate this limitation, the garbage collector is triggered when more than 2/3s of the memory are used. This is done in the attempt to guarantee that at any time there is at least 1/3 of memory that can be used for the injection of new code. When the execution reaches the point that more than 2/3 of the memory are permanently used and cannot be freed, the garbage collector is disabled, because executing it would not provide any benefit. It is re-enabled when either new aspects are added or enabled/disabled, since these facts might provide the opportunity to free some memory.

5.10 Optimizations

The virtual machine contains few optimizations:

- The weaver keeps a cache of the strings extracted from the memory when the trap operation is performed.
- The weaver caches the tuples extracted from memory when conditions containing *test* constructs are evaluated. This caching system increments the performances of the backtracker that can save some of the overhead necessary to reconstruct the tuple from the memory.

Both the caches are cleared when the garbage collector executes, because after it has been executed the frames storing the strings and the tuples may have been moved or removed.

Chapter 6

Supporting Other Calculi

The virtual machine presented in this thesis was designed and implemented for the execution of the AspectK coordination language. Many of the structures and the instructions are specific to the AspectK language and may not be suitable for other languages. Some examples are:

- The *channel* frame, which may be not suitable to be used as a communication channel in other languages. In these channels there is no notion of synchronous communication which instead is featured in languages such as CSP.
- The `MATCH` and `DMATCH` instructions, which implement the logic of the AspectK *match* function and rely on a specific structure of the memory, namely the presence of channel frames bound to tuple lists.

Moreover the AspectK language uses only one data type, namely the *string*. Other languages may require instructions for the comparison of strings, their concatenation and so on. Other process calculi may also support the declaration of other data types, such as integers or floats, which the current virtual machine does not support at all.
In general, in order to support a new process calculus the virtual machine should be updated with new instructions and/or new memory allocation units.

However it is possible to determine a general set [18] of operators for processes modelling, which are supported by numerous process calculi such as CSP, CCS and π-calculus. The operators are:

- *Basic Process.* The basic process is of two kinds: the *inactive* process, which usually appears as *0*, *nil* or *stop* and the *termination* process which is denoted by *skip* or *stop*.
- *Action Prefixing.*
- *Sequentialization.*
- *Choice.* Choice can be non-deterministic, external or internal.
- *Parallel Composition.* The parallel composition of processes may also involve *interaction*: processes may synchronize their execution and pass messages between them.
- *Abstraction.* Abstraction can be *restriction*, *hiding* and *renaming*.
- *Infinite Behaviours.* Namely *recursion*, *replication* and *iteration*.

One direction to simplify the support of other process calculi could be to extend the design of the AspectK virtual machine to support all of these operators. Providing native structures to natively support these operators may require the addition of new frames, new instructions or the extension of the existent ones. On the other hand some of the operators may be handled at compile time and they would not affect anyhow the design of the virtual machine.
For example, the virtual machine does not provide support for synchronous interaction of processes. The support of this kind of interaction (present in CSP) may require the design of a new channel frame and/or the addition of new structures that can provide means for *connecting* two processes to operate the synchronization.
Conversely, *restriction* could be implemented at compile time by renaming the restricted channels with new names that are not used for any other channel present in the net: hence the support of restriction would not affect the virtual machine design.
Almost the totality of the process calculi do not have AOP support. However, had any new AOP-oriented process calculus to be supported in the virtual machine, this would require the creation of a new trapper and weaver and possibly the extension of the current set of *tag* instructions.

Chapter 7

Final Considerations

In the first part of the thesis we analyzed the syntax and the semantics of AspectK. We also presented a possible extension/modification of the semantics of the language that can be used to optimize, under certain conditions, the dynamic weaving of aspects. Then, we overviewed some basic concepts about virtual machines, and we presented some existing solutions based on virtual machines that support the dynamic weaving of aspects.

In the subsequent part of the thesis we presented the interpreter. The interpreter is a prototype software that was conceived for obtaining an understanding of what the challenges in creating the virtual machine could be at an early stage of the work. The time spent on this prototype payed off, because by the time I had built up the knowledge necessary to develop the AspectK virtual machine, I was already aware of what the most sensible points in the design were. While working on the interpreter I developed the thread scheduling strategy based on sets, I identified possible issues in trapping actions that are *guards* in a guarded-processes sum and I refined the *lazy* and the *balanced* weaving techniques.

In the third part of the thesis we studied the design of the virtual machine and the working of its internals. We analyzed how a net is compiled into a chunk-of-code based structure and we presented how the net is stored in memory and then executed. Then we discussed the weaver and the three dynamic weaving strategies it features. Finally we presented the garbage collector and explained

its role in ensuring the availability of memory for the injection of new code.

In the final chapter we briefly considered what should be done for supporting more process calculi in the virtual machine.

7.1 Analysis of the Results

The work on the AspectK virtual machine highlighted some interesting facts concerning the native support of Aspect-Oriented Programming in a virtual machine.

First of all, the virtual machine seems to be an efficient solution for implementing the dynamic weaving of aspects. The process of identification of the joinpoints and injection of new code can be triggered by the virtual machine itself which can efficiently activate the weaver whenever certain instructions or certain conditions are encountered during the execution.

The AspectK virtual machine ships with three different strategies for dynamically weaving aspects and none of them can be considered optimal under *every* type of workload. This fact suggests that better performances can be achieved by matching the most convenient weaving strategy with the workload that the virtual machine is undertaking. If the set of aspects is invariant in time and the majority of aspects are *non-conditional*, the *balanced* technique may be the most appropriate. Conversely, if the set of aspects is expected to change frequently, it may be more convenient to use the *lazy* weaver.

Weaving aspects at run-time in general requires the knowledge about the architecture of the binary code of the original program. In the rather simple case of AspectK the actions were compiled into chunks that have a well defined structure. It was then relatively easy to define strategies for weaving new code and linking it to the original one. The situation may become very different if the structure of the code is arbitrarily changed by compiler optimizations or other causes. In such case the weaving of new code may become very complex or even impossible. This consideration seems to explain why there exist many solutions for dynamic weaving of aspects in the Java virtual machine (AspectWerkz [3] or Steamloom [2]) since the *class* format is well specified, whereas this is not the case for other languages like C++ where the structure of the compiled code may be much more difficult to derive.

7.2 Conclusions

The initial goal of this thesis was to explore how Aspect-Oriented Programming and virtual machines could be combined to implement the dynamic weaving of aspects in the code. This objective was further extended into the analysis of the properties of the AspectK language and the identification of ways to extend it to support different solutions for the dynamic weaving.

The first challenge to face was to build the necessary knowledge base that could enable me to confront this project and exploit it in the right direction. It was very important to keep the design simple and avoid excessively complex solutions. I encountered some obstacles in specifying a satisfactory design of the bytecode, because it had to be expressive enough to permit the compilation of the net and at the same time simple enough to facilitate the operations of trapping and weaving. The compromise was found in moving some of the complexity into the design of the instructions, and the result is that some of them execute rather sophisticated operations (`MATCH` for example). On the other hand this choice pays off for at least two reasons: first of all the structure of the compiled program is more compact and easier to manipulate, secondly the porting of the virtual machine to other process calculi is facilitated, because the business logic of new operations can be almost fully confined within new instructions. As a matter of fact this kind of design choice was made for the same reason by the developers of other virtual machines for process calculi, for example TyCO [16].

The subsequent challenge was represented by the development of the weaver. The *lazy* weaving solution was designed by closely following the semantics of AspectK, whereas for the *greedy* and *balanced* strategies a new semantics was derived from the original one and then implemented.

There was not enough time to attempt the port of the virtual machine to other process calculi however, some possible directions for this kind of task were considered and reported in the thesis.

Overall, the project resulted in a fully functional implementation of a virtual machine for AspectK.

7.3 Further Work

There are several directions that can be taken to further extend the project presented in this thesis. Some of them have already been mentioned, namely the porting of the virtual machine to other process calculi or the improvement of the design of the bytecode for allowing the execution of the garbage collector at any time.

Other possibilities are the improvement of the design of the aspects tags. For example, the tag instruction that "informs" the weaver about the type of action, the number and type (constant, variable, ...) of its arguments and the type of its target location can consume quite a lot of space. Maybe the same information can be obtained by other means.

Most of the woven code is discarded once it has been used. It might be possible to *cache* part of it for future re-use. This solution would be beneficial for the performances of the virtual machine.

The virtual machine could be equipped with a component capable of automatically selecting the most adequate weaver given the detected workload and set of aspects. Again, the performances of the virtual machine would obtain discrete benefits from this solution.

Finally, an important improvement would be the creation of a static analyzer that detects the presence of cycles in the set of aspects. Currently, the virtual machine can partially control this situation but it can not avoid it. If a cycle is present the virtual machine keeps allocating code frames, and after some cycles all the available virtual memory is filled.

Appendix A

Example Code

This appendix contains the output of the disassembled bytecode for the compiled net

```
loc1 :: <t1>
||
loc2 :: * out(t2)@loc1.( in(!a)@loc1.0 | read(!b)@loc1.0 )
||
loc3 :: out(t2)@loc1.0 + read(!b)@loc2.0
```

The code is reported below.

```
Size of the Constants Table: 5


4: GC_CHECK 155
9: NEWSTRING "t2"
13: CT_STORE 3
16: NEWSTRING "t1"
20: CT_STORE 1
23: NEWSTRING "loc1"
```

```
29: CT_STORE 0
32: NEWSTRING "loc3"
38: CT_STORE 4
41: NEWSTRING "loc2"
47: CT_STORE 2
50: CT_LOAD 0
53: DUP
54: GETCHAN
55: JMP_NOTNULL 7
58: NEWCHAN
59: JMP_OFF 4
62: GETCHAN
63: NEWARRAY 1
68: DUP
69: CT_LOAD 1
72: ASTORE 0
77: APPEND
78: CT_LOAD 2
81: DUP
82: GETCHAN
83: DUP
84: JMP_NOTNULL 5
87: SWAP
88: NEWCHAN
89: TH_NEW vt size:0 ref:119
98: CT_LOAD 4
101: DUP
102: GETCHAN
103: DUP
104: JMP_NOTNULL 5
107: SWAP
108: NEWCHAN
109: TH_NEW vt size:1 ref:351
118: SCHED_GREY

119: GC_CHECK 31
124: TH_GETCHAN
125: TH_NEW vt size:0 ref:135
134: SCHED_REP

135: GC_CHECK 83
140: ASP_OUTACTION_TAG
        1 arguments
        c 3
```

```
152: ASP_PREACTIONS 0
157: CT_LOAD 0
160: DUP
161: GETCHAN
162: DUP
163: JMP_NOTNULL 5
166: SWAP
167: NEWCHAN
168: NEWARRAY 1
173: DUP
174: CT_LOAD 3
177: ASTORE 0
182: APPEND
183: ASP_POSTACTIONS
184: TH_GETCHAN
185: TH_NEW vt size:1 ref:205
194: TH_GETCHAN
195: TH_NEW vt size:1 ref:278
204: SCHED_TPL_GREY

205: GC_CHECK 9
210: ASP_INACTION_TAG
        1 arguments
        vd 0
222: ASP_PREACTIONS 0
227: CT_LOAD 0
230: GETCHAN
231: DUP
232: JMP_NOTNULL 7
235: DNOP
236: JMP_OFF 40
239: NEWARRAY 1
244: DUP
245: TH_VT_LOAD 0
248: ASTORE 0
253: DMATCH
254: DUP
255: JMP_NOTNULL 7
258: DNOP
259: JMP_OFF 17
262: ALOAD 0
267: TH_VT_STORE 0
270: ASP_POSTACTIONS
271: SCHED_YELLOW 277
```

```
276: SCHED_RED
277: SCHED_GREY

278: GC_CHECK 9
283: ASP_READACTION_TAG
        1 arguments
        vd 0
295: ASP_PREACTIONS 0
300: CT_LOAD 0
303: GETCHAN
304: DUP
305: JMP_NOTNULL 7
308: DNOP
309: JMP_OFF 40
312: NEWARRAY 1
317: DUP
318: TH_VT_LOAD 0
321: ASTORE 0
326: MATCH
327: DUP
328: JMP_NOTNULL 7
331: DNOP
332: JMP_OFF 17
335: ALOAD 0
340: TH_VT_STORE 0
343: ASP_POSTACTIONS
344: SCHED_YELLOW 350
349: SCHED_RED
350: SCHED_GREY

351: GC_CHECK 40
356: CLEARSTACK
357: RNDSEQ 2
360: ISSTACKEMPTY
361: JMP_NULL 4
364: SCHED_RED
365: SWITCH
        2 choices
        6
        60
371: ASP_GUARD_TAG 360
376: ASP_OUTACTION_TAG
        1 arguments
        c 3
```

```
388: ASP_PREACTIONS 0
393: CT_LOAD 0
396: DUP
397: GETCHAN
398: DUP
399: JMP_NOTNULL 5
402: SWAP
403: NEWCHAN
404: NEWARRAY 1
409: DUP
410: CT_LOAD 3
413: ASTORE 0
418: APPEND
419: ASP_POSTACTIONS
420: SCHED_TPL_YELLOW 501
425: ASP_GUARD_TAG 360
430: ASP_READACTION_TAG
        1 arguments
        vd 0
442: ASP_PREACTIONS 0
447: CT_LOAD 2
450: GETCHAN
451: DUP
452: JMP_NOTNULL 7
455: DNOP
456: JMP_OFF 40
459: NEWARRAY 1
464: DUP
465: TH_VT_LOAD 0
468: ASTORE 0
473: MATCH
474: DUP
475: JMP_NOTNULL 7
478: DNOP
479: JMP_OFF 17
482: ALOAD 0
487: TH_VT_STORE 0
490: ASP_POSTACTIONS
491: SCHED_YELLOW 502
496: JMP_ABS 360
501: SCHED_GREY
502: SCHED_GREY
```

Appendix B

Software

This appendix presents some information about the software developed while working on this thesis.

B.1 The AspectK Compiler

The AspectK compiler described in Section 5.6 is implemented by the *aspectKc* application.

The aspectKc program can perform two tasks: it can either compile an AspectK source file or it can disassemble a compiled net.
When it is used to compile, aspectKc produces two files:

- One file with extension "ako", containing the compiled net.
- One file with extension "as", containing the compiled aspects.

If aspectKc used to disassemble a compiled program, aspectKc can output the disassembled representation either to the standard output or into a file.

The contextual help of this program can be invoked by passing either the `-h` or `--help` switch to the program from the command line.

B.1.1 Syntax Accepted by the Parser

The parser accepts the syntax of AspectK defined in Chapter 2, with minor modifications:

- Aspects are defined using the `^=` symbol, which corresponds to $\stackrel{\triangle}{=}$.
- Multiple aspect definitions are separated by comma.
- Constants, within cuts, must be prefixed by the `$` symbol, in order to differentiate them from the variables.
- Within conditions, the `and` keyword is used to define *and* conditions.
- The parser does not accepts tabs.
- Variables cannot start with a number.
- Comments are enclosed between `\* *\`

B.2 The AspectK GUI

aspectKgui is a graphical user interface to the virtual machine.
Thus tool may be used to load in the virtual machine an AspectK net compiled into bytecode and a collection of aspects. Then, the net can be executed and the interface allows to inspect the virtual machine internals to see how the different parts change during the execution.

Figure B.1 shows the main window of aspectKgui. There are three menus:

- *Main.* This menu contains the sub-menus:
  - *Options.* Here it is possible to control some options of the virtual machine, like the size of the virtual memory or the weaving strategy to use.
  - *About.* Shows information about the author.
  - *Exit.* Exits the application.

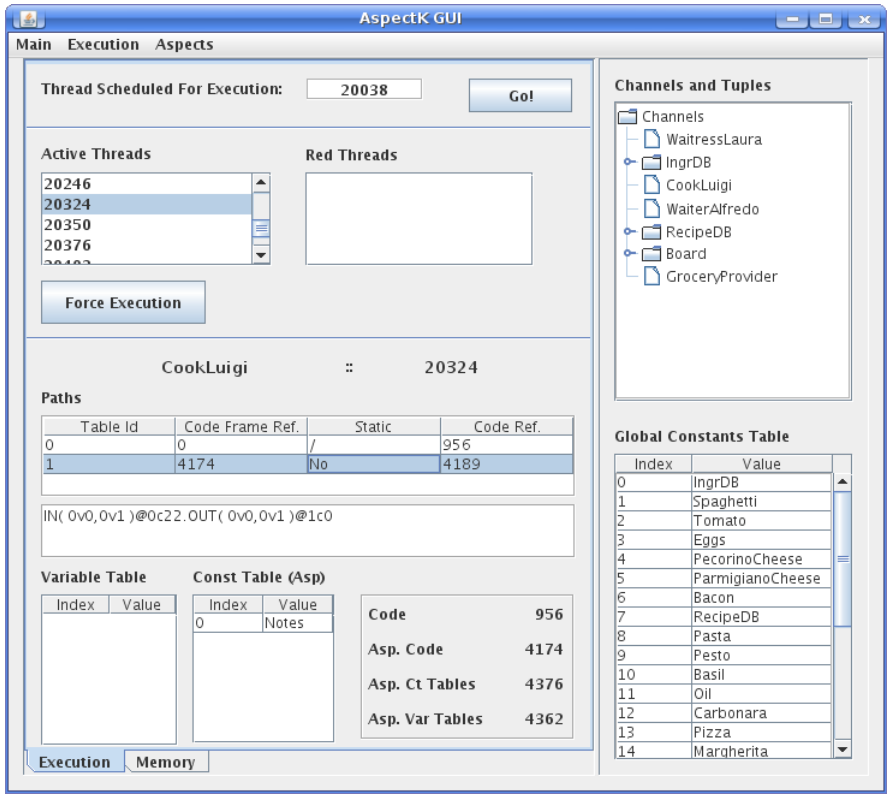


Figure B.1: aspectKgui, *Execution* tab.

- *Execution.* Contains three sub-menus:
  - *Load Program.* Creates a virtual machine instance and loads a program in it. If aspects had been previously loaded, they are loaded in the aspect pool of the virtual machine too.
  - *Program Bytecode.* Opens a window containing the disassembled representation of the program.
  - *Restart VM.* Restarts the virtual machine.
- *Aspects.* Contains two sub-menus:
  - *Add.* Permits to load aspects in the virtual machine.
  - *Manage.* Opens a windows where it is possible to consult the loaded aspects and enable/disable them.

The body of the application is divided by a split panel into two parts. The left part contains the main body of the application, distributed into two tab panels. The right part of the interface displays the content of the channel list and the content of the global constant table.
Figure B.1 shows the *Execution* tab. The upper part of the tab panel displays the active threads and the red threads. All the numbers displayed are memory addresses.
The *Go!* button triggers the execution of the chunk scheduled for execution.
The *Force Execution* button changes the thread selected for execution with the one selected in the *Active Threads* list.
If the thread selected for the execution or one of the active threads or one of the read threads is clicked over, the lower part of the window is filled with information about such thread. The *Paths* table shows the execution paths for the thread. There is always at least one path, representing the original program. Whenever some code is woven and new frames are allocated, they are appended at the end of this table, so it is possible to consult the new code frames.
In the figure, one code frame was allocated, and the code it contains is shown in the text area under the table. The text representation of the locations have this encoding: $t\{c,v\}i$.

- t is the table id. 0 for the global constant table or for the variable table of the thread, a number greater or equal to 1 for constants and variables that are temporary introduced by the injected code.
- $\{c,v\}$: c if the location is a constant, v if it is a variable.
- i is the index of the symbol on the table

For example, in the figure, the last location is *1c0*, which indicates:

- a constant;
- stored in the temporary constant table allocated for the code frame number 1, in this case the only code frame allocated;
- the constant is stored at index 0 of such table.

There are two tables in the bottom part of *Execution* panel:

- *Variable Table.* Shows the content of the variable table of the process, if the path corresponding to the main program is selected, otherwise it displays the content of the temporary variable table of the selected code frame.

- *Const Table (Asp).* Displays the content of the temporary constant table associated to the selected code frame. When the user selects the path corresponding to the main program, the table is obviously empty.

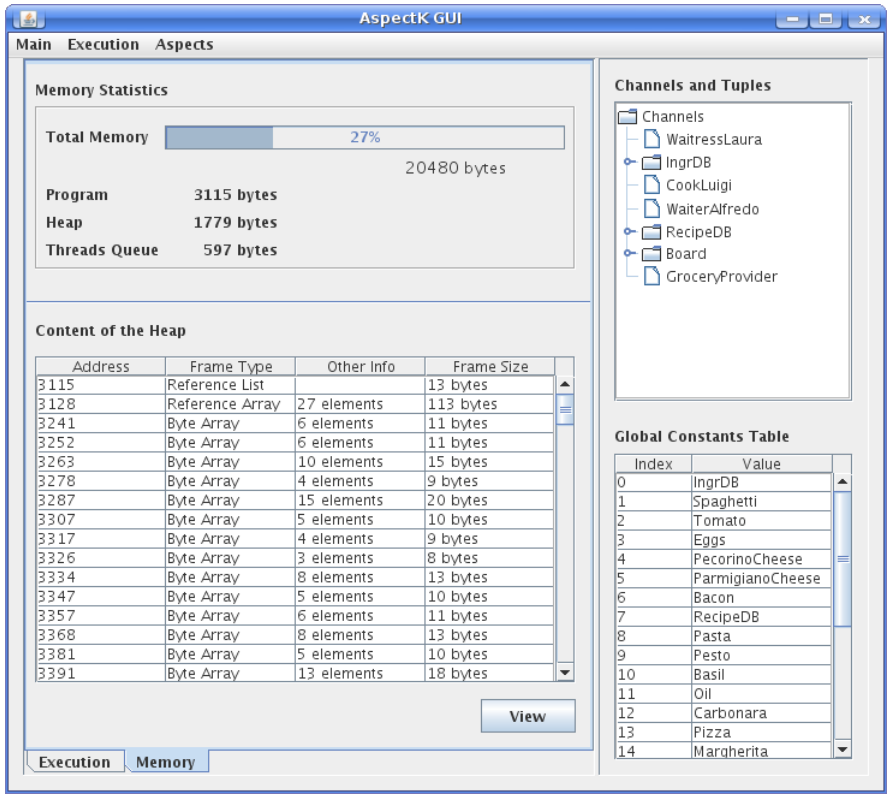


Figure B.2: aspectKgui, *Memory* tab.

Figure B.2 shows the content of the *Memory* tab of the application.
The upper part displays some general information about the memory usage, whereas the bottom part displays the content of the heap.
The *View* button opens a window that displays the content of the selected frame.

B.3 The AspectK Interpreter

Figure B.3 shows how the interpreter appears to the user. The interface does not allow to perform many operations.

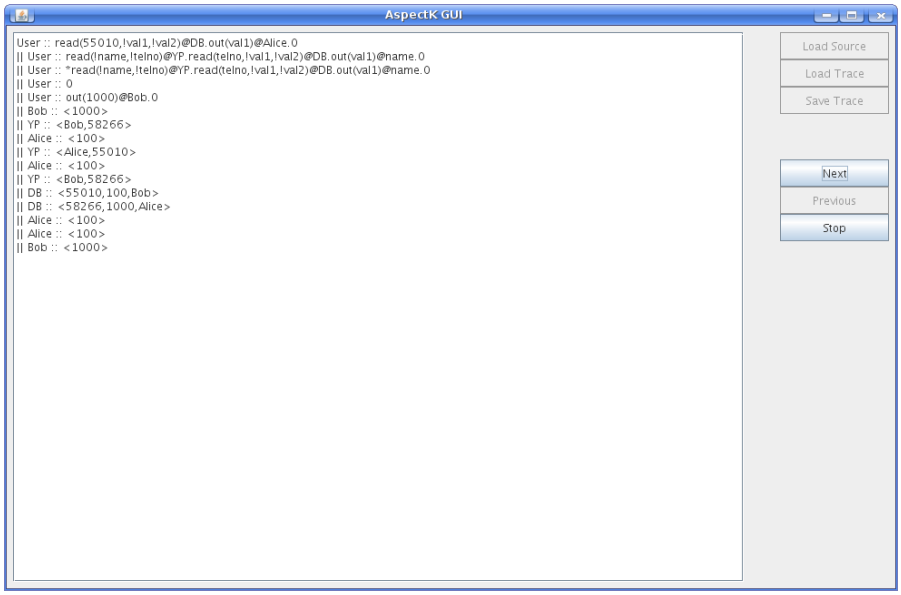


Figure B.3: The AspectK Interpreter

The *Load Source* program permits to locate an AspectK source file and to load it. After the source has been loaded, the execution is controlled by the two buttons *Next* — it executes the next task of the process scheduled for execution — and *Stop* — which stops the execution.
After *Stop* has been pressed, the evolution of the net is stopped and it can be saved into a trace by using the *Save Trace* button.

A trace can be loaded by using the *Load Trace* button. It is possible to move forward or backward in the evolution of the net recorded in the trace by using the *Next* and the *Previous* buttons.
The trace is closed by pressing the *Stop* button.

NOTE: The interpreter is simply a prototype and it was neither developed nor tested with accuracy, hence it may show malfunctions.

Appendix C

Instruction Set

ALOAD

Operation Loads on the operand stack a reference from a reference array

Format

ALOAD
index_byte1
index_byte2
index_byte3
index_byte4

Opcode 12 (0x0c)

Stack ...,array_reference ⇒
...,reference

APPEND

Operation Appends a tuple (array of strings) to the end of the tuple list of the given channel. The reference of the channel and of the tuple are on the operand stack.

Format

APPEND

Opcode 16 (0x10)

Stack ...,chan_reference,tuple_reference ⇒
...,

ASP_CLEAN

Operation Removes the link to the aspect's code in the program code (if present) an jumps to the "parent" code (it could be another code frame or the original code) or schedules the process for termination

Format

ASP_CLEAN
reference_byte1
reference_byte2
reference_byte3
reference_byte4

Opcode 46 (0x2e)

Stack No change

ASP_CT_LOAD

Operation Loads on the operand stack a reference of a string from the constants table associated to the code frame being executed by the thread

Format

ASP_CT_LOAD
table_id
index_byte1
index_byte2

Opcode 43 (0x2b)

Stack ..., ⇒
...,reference

ASP_CT_STORE

Operation Stores a reference (loaded on the operand stack) of a string in the constants table associated to the code frame being executed by the thread

Format

ASP_CT_STORE
table_id
index_byte1
index_byte2

Opcode 42 (0x2a)

Stack ...,reference ⇒
...,

ASP_GUARD_TAG

Operation Tags the fact that the action encoded in the current chunk of code is a guard, in a guarded process sum

Format

ASP_GUARD_TAG
switch_ref_byte1
switch_ref_byte2
switch_ref_byte3
switch_ref_byte4

Opcode 37 (0x25)

Stack No change

ASP_INACTION_TAG

Operation Tags the beginning of the “in” action

Format

ASP_INACTION_TAG
number_of_arguments
table id
type_arg1
arg1_byte1
arg1_byte2
...
table id
type_argn
argn_byte1
argn_byte2
table id
type_tgtloc
tgtloc_byte1
tgtloc_byte2
loc_byte1
loc_byte2

Opcode 34 (0x22)

Stack No change

ASP_OUTACTION_TAG

Operation Tags the beginning of the “out” action

Format

ASP_OUTACTION_TAG
number_of_arguments
table id
type_arg1
arg1_byte1
arg1_byte2
...
table id
type_argn
argn_byte1
argn_byte2
table id
type_tgtloc
tgtloc_byte1
tgtloc_byte2
loc_byte1
loc_byte2

Opcode 33 (0x21)

Stack No change

ASP_READACTION_TAG

Operation Tags the beginning of the “read” action

Format

ASP_READACTION_TAG
number_of_arguments
table id
type_arg1
arg1_byte1
arg1_byte2
...
table id
type_argn
argn_byte1
argn_byte2
table id
type_tgtloc
tgtloc_byte1
tgtloc_byte2
loc_byte1
loc_byte2

Opcode 35 (0x23)

Stack No change

ASP_POSTACTIONS

Operation Tags the place where post-actions should be inserted, in the code.

Format

ASP_POSTACTIONS

Opcode 39 (0x27)

Stack No change

ASP_PREACTIONS

Operation Tags the place where post-actions should be inserted, in the code. It also triggers the weaver, which attempts the trap and if this succeeds the code is injected.

Format

ASP_PREACTIONS
reference_byte1
reference_byte2
reference_byte3
reference_byte4

Opcode 38 (0x26)

Stack No change

ASP_SCHED_TPL_YELLOW

Operation Schedule the next chunk of code in this thread (within a code frame) and signal that a tuple was created.

Format

ASP_SCHED_TPL_YELLOW
offset_byte1
offset_byte2

Opcode 45 (0x2d)

Stack No change

ASP_SCHED_YELLOW

Operation Schedule the next chunk of code in this process, within a code frame.

Format

ASP_SCHED_YELLOW
offset_byte1
offset_byte2

Opcode 44 (0x2c)

Stack No change

ASP_STOPACTION_TAG

Operation Tags the stop action

Format

ASP_STOPACTION_TAG

Opcode 36 (0x24)

Stack No change

ASP_VT_LOAD

Operation Loads on the operand stack a reference from the variable table associated to the code frame being executed by the thread

Format

ASP_VT_LOAD
table_id
index_byte1
index_byte2

Opcode 41 (0x29)

Stack ..., ⇒
...,reference

ASP_VT_STORE

Operation Stores a reference (loaded on the operand stack) in the symbol table associated to the code frame being executed by the thread

Format

ASP_VT_STORE
table_id
index_byte1
index_byte2

Opcode 40 (0x28)

Stack ...,reference ⇒
...,

ASTORE

Operation Stores a reference (loaded on the operand stack) in a reference array

Format

ASTORE
index_byte1
index_byte2
index_byte3
index_byte4

Opcode 13 (0x0d)

Stack ...,array_reference,reference ⇒
...,

BASTORE

Operation Stores a byte in a byte array

Format

BASTORE
index_byte1
index_byte2
index_byte3
index_byte4
byte

Opcode 10 (0x0a)

Stack ...,array_reference ⇒
...,

CLEARSTACK

Operation Empties the operand stack

Format

CLEARSTACK

Opcode 48 (0x30)

Stack ..., ⇒
\

CT_LOAD

Operation Loads on the operand stack a reference of a string from the global constant table

Format

CT_LOAD
index_byte1
index_byte2

Opcode 21 (0x15)

Stack ..., ⇒
...,reference

CT_STORE

Operation Stores the reference (loaded on the operand stack) of a string in the global constant table

Format

CT_STORE
position_byte1
position_byte2

Opcode 20 (0x14)

Stack ...,reference ⇒
...,

DMATCH

Operation Returns an array of substitutions, if a match is found, otherwise `null`. Additionally, it removes the matched tuple from the channel.

Format

DMATCH

Opcode 18 (0x12)

Stack ...,chan_reference,args_reference ⇒
...,subs_reference

DNOP

Operation No operation. It also removes the top element of the operand stack.

Format

DNOP

Opcode 1 (0x01)

Stack No change

DUP

Operation Duplicate the top operand stack word.

Format

DUP

Opcode 3 (0x03)

Stack ...,word ⇒
...,word,word

GC_CHECK

Operation Checks whether the garbage collector should be executed.

Format

GC_CHECK
nbytes_byte1
nbytes_byte2
nbytes_byte3
nbytes_byte4

Opcode 50 (0x32)

Stack No change

GETCHAN

Operation Returns the reference of the channel bound to the given string.

Format

GETCHAN

Opcode 15 (0x0f)

Stack ...,string_reference ⇒
...,chan_reference

ISSTACKEMPTY

Operation pushes 1 on the top of the operand stack if the stack is empty, otherwise null.

Format

ISSTACKEMPTY

Opcode 49 (0x31)

Stack ..., ⇒
...,value

JMP_ABS

Operation Jump to an absolute address.

Format

JMP_ABS
address_byte1
address_byte2
address_byte3
address_byte4

Opcode 4 (0x04)

Stack No change

JMP_OFF

Operation Jump to an offset-address.

Format

JMP_OFF
offset_byte1
offset_byte2

Opcode 5 (0x05)

Stack No change

JMP_NOTNULL

Operation Jump to an offset-address if the value is not null.

Format

JMP_NOTNULL
offset_byte1
offset_byte2

Opcode 7 (0x07)

Stack No change

JMP_NULL

Operation Jump to an offset-address if the value is null (0x00000000)

Format

JMP_NULL
offset_byte1
offset_byte2

Opcode 6 (0x06)

Stack No change

MATCH

Operation Returns an array of substitutions, if a match is found, otherwise null.

Format

MATCH

Opcode 17 (0x11)

Stack ...,chan_reference,array_reference ⇒
...,array_reference

NEWARRAY

Operation Instantiates a new array of references

Format

NEWARRAY
size_byte1
size_byte2
size_byte3
size_byte4

Opcode 11 (0x0b)

Stack ..., ⇒
...,array_reference

NEWBARRAY

Operation Instantiates a new array of bytes

Format

NEWBARRAY
size_byte1
size_byte2
size_byte3
size_byte4

Opcode 9 (0x09)

Stack ..., ⇒
...,array_reference

NEWCHAN

Operation Instantiates a new channel bound to a given name (string)

Format

NEWCHAN

Opcode 14 (0x0e)

Stack ...,string_reference ⇒
...,chan_reference

NEWSTRING

Operation Instantiates a new string (short cut for using byte arrays)

Format

NEWSTRING
size_byte1
char1
...
charN

Opcode 19 (0x13)

Stack ..., ⇒
...,array_reference

NOP

Operation No operation

Format

NOP

Opcode 0 (0x00)

Stack No change

RNDSEQ

Operation Pushes on the Operand Stack the sequence of values $0, \cdots, n$ in random order ($n \leq 255$)

Format

RNDSEQ
n_byte1
n_byte2

Opcode 47 (0x2f)

Stack ..., ⇒
...,valuen,...,value1

SCHED_GREY

Operation Signals the scheduler to move the current thread in the "grey" set.

Format

SCHED_GREY

Opcode 31 (0x1f)

Stack No change

SCHED_REP

Operation Signals the scheduler to move the current thread in the "replication" set

Format

SCHED_REP

Opcode 32 (0x20)

Stack No change

SCHED_RED

Operation Signals the scheduler to move the current thread in the "red" set

Format

SCHED_RED

Opcode 28 (0x1c)

Stack No change

SCHED_TPL_GREY

Operation Signals the scheduler to move the current thread in the "grey" set. Signals the scheduler to move the red threads in the "yellow" set.

Format

SCHED_TPL_GREY

Opcode 29 (0x1d)

Stack No change

SCHED_TPL_YELLOW

Operation Signals the scheduler to move the current thread in the "yellow" set. Schedules the next chunk of code for this thread. Signals the scheduler to move the red threads in the "yellow" set.

Format

SCHED_TPL_YELLOW
address_byte1
address_byte2
address_byte3
address_byte4

Opcode 30 (0x1e)

Stack No change

SCHED_YELLOW

Operation Signals the scheduler to move the current thread in the "yellow" set. Schedules the next chunk of code for this thread.

Format

SCHED_YELLOW
address_byte1
address_byte2
address_byte3
address_byte4

Opcode 27 (0x1b)

Stack No change

SWAP

Operation Swaps the top 2 values of the operand stack.

Format

SWAP

Opcode 2 (0x02)

Stack
...,value1,value2 ⇒
...,value2,value1

SWITCH

Operation Reads the top value of the operand stack and selects one of the offsets provided as arguments. Them, it jumps to the address derived from the selected offset.

Format

SWITCH
n_choices
offset1_byte1
offset1_byte2
.
.
.
offsetN_byte1
offsetN_byte2

Opcode 8 (0x08)

Stack
...,value ⇒
...,

TH_GETCHAN

Operation Pushes on the operand stack the reference of the channel associated to this thread.

Format

TH_GETCHAN

Opcode 26 (0x1a)

Stack ..., ⇒
chan_reference

TH_NEW

Operation Instantiates a new thread. The arguments determine the size of the variable table and the address of the initial chunk of code.

Format

TH_NEW
vt_size_byte1
vt_size_byte2
vt_size_byte3
vt_size_byte4
address_byte1
address_byte2
address_byte3
address_byte4

Opcode 22 (0x16)

Stack ...,chan_reference ⇒
..., thread_reference

TH_VT_LOAD

Operation Loads the value of a variable from the variable table of the thread in execution.

Format

TH_VT_LOAD
index_byte1
index_byte2

Opcode 24 (0x18)

Stack ..., ⇒
...,reference

TH_VT_STORE

Operation Stores a reference in the variable table of the thread in execution.

Format

TH_VT_STORE
index_byte1
index_byte2

Opcode 23 (0x17)

Stack ...,reference ⇒
...,

TH_VT_STORE_IND

Operation Stores a reference in the variable table of a thread whose reference is on the operand stack.

Format

TH_VT_STORE_IND
index_byte1
index_byte2

Opcode 25 (0x19)

Stack ...,th_reference,reference ⇒
...,

Bibliography

[1] Jonas Bonér Joakim Dahlstedt, Alexandre Vasseur. Java virtual machine support for aspect-oriented programming. In *5th International Conference on Aspect-Oriented Software Development (AOSD'2006)*.

[2] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In *AOSD*, pages 83–92, 2004.

[3] Aspectwerkz home page. `http://aspectwerkz.codehaus.org/`, retrieved 15-08-2008.

[4] Chris Hankin, Flemming Nielson, Hanne Riis Nielson, and Fan Yang. Advice for coordination. In *COORDINATION*, pages 153–168, 2008.

[5] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999.

[6] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, 1974.

[7] Dieter Gollmann. *Computer security*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[8] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. pages 220–242. Springer-Verlag, 1997.

[9] Geri Georg, Indrakshi Ray, and Robert France. Using aspects to design a secure system. In *ICECCS '02: Proceedings of the Eighth International*

Conference on Engineering of Complex Computer Systems, page 117, Washington, DC, USA, 2002. IEEE Computer Society.

[10] L. Bettini, V. Bono, R. Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The klaim project: Theory and practice, 2003.

[11] Rocco de Nicola, Gian Luigi Ferrari, and R. Pugliese. klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering (Special Issue on Mobility and Network Aware Computing)*, 1998.

[12] Rocco De Nicola, GianLuigi Ferrari, and Rosario Pugliese. Programming access control: The KLAIM Experience. *Lecture Notes in Computer Science*, 1877:48–??, 2000.

[13] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[14] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, June 2005.

[15] Javassist home page. `http://www.csg.is.titech.ac.jp/~chiba/javassist/`, retrieved 20-08-2008.

[16] L. Lopes. *On the Design and Implementation of a Virtual Machine for Process Calculi*. PhD thesis, Faculty of Science, University of Porto, Portugal, 1999.

[17] Javacc home page. `https://javacc.dev.java.net/`, retrieved 20-08-2008.

[18] M. Wirsing. Process algebra: Operators for process algebras, WS 07/08. Set of slides.

[19] Michael Engel and Bernd Freisleben. Using a low-level virtual machine to improve dynamic aspect support in operating system kernels. In *Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Chicago, USA, 2005.

[20] M. Haupt and M. Mezini. Virtual Machine Support for Aspects with Advice Instance Tables. *L'Objet*, 11(3):9–30, 2005.

[21] Terracotta Jonas Bonér and Terracotta Eugene Kuleshov. Clustering the java virtual machine using aspect-oriented programming. In *AOSD Conference 2007*.

[22] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[23] S. Gao. Applying aspect-orientation in designing security systems: A case study, 2004.

[24] Luis M. B. Lopes, Fernando M. A. Silva, and Vasco Thudichum Vasconcelos. A virtual machine for a process calculus. In *PPDP '99: Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming*, pages 244–260, London, UK, 1999. Springer-Verlag.

[25] J. E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32–38, 2005.

[26] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness, 2000.

[27] J.T. Bloch J. Viega and P. Chandri. Applying Aspect-Oriented Programming to Security. *Cutter IT Journal*, 14(2):31–39, 2001.

[28] Robert P. Goldberg. Survey of Virtual Machines Research. *IEEE Computer*, pages 34–45, 1974.

[29] Aspectj home page. `http://www.eclipse.org/aspectj/`, retrieved 15-08-2008.