A theory of distributed aspects

Nicolas Tabareau

A theory of distributed aspects

Nicolas Tabareau
INRIA, Ascola team
École des mines de Nantes, France

ABSTRACT

Over the last five years, several systems have been proposed to take distribution into account in Aspect Oriented Programming. While being fruitful to develop or improve distributed component infrastructures or application servers, those systems are not given with a formal semantics and so do not permit to establish properties on the code to be executed. This paper introduces the aspect join calculus – an aspect oriented and distributed language based on the join calculus – to provide a first formal theory of distributed AOP in which we can express many features of previous distributed AOP models.

The semantics of the aspect join calculus is given by a (chemical) operational semantics and a type system is developed to ensure properties satisfied by aspect during the execution of a process. We also give a translation of the aspect join calculus into the core join calculus. The translation is proved to be correct by a bisimilarity argument. In this way, we provide a well-defined version of a weaving algorithm which constitutes the main step towards an implementation of the aspect join calculus directly in JoCaml.

We conclude this paper by showing that despite its minimal definition, the aspect join calculus is a convenient language in which existing distributed AOP models can be formalized. Indeed, many features of previous distributed AOP models (such as remote pointcut, distributed advices, migration of aspects, asynchronous and synchronous aspects, re-routing of messages, distributed control flow) can be defined in this simple language.

Categories and Subject Descriptors

D.3.1 [**Programming languages**]: Formal definitions and theory; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*

General Terms

Languages, Theory

1. INTRODUCTION

Distributed applications are more complex to develop than sequential applications, mainly because of synchronization issues and distribution of the code across the network. It has been advocated that traditional programming languages do not allow to separate distribution concerns from standard functional concerns in a satisfactory way. For instance, data replication, transactions, security, and fault tolerance often crosscut the business code of a distributed application. Aspect-Oriented Programming (AOP) promotes better separation of concerns in software systems by introducing aspects for the modular implementation of crosscutting concerns. Indeed, AOP provides the facility to intercept the flow of control in an application and perform new computation at that point. In this approach, computation at certain execution events, called *join points*, may be intercepted by a particular condition, called *pointcut*, and modified by a piece of code, called an *advice*.

But even though AOP is routinely used in distributed component infrastructures (such as EJB) or application servers (such as JBoss), most AOP models do not support the remote definition or application of aspects. For instance, in Spring or JBoss, non-distributed aspects are used to manipulate distributed infrastructures. As pointed out by Tanter et al. [15], AOP in distributed systems is not distributed AOP. Over the past five years, we assisted to the emergence of several systems in the field of disttributed AOP: JAC [10], DJcutter [9], ReflexD [15], AWED [8], a Scheme-like language with distribution and aspects [14]. Those languages introduces new concepts for distributed AOP such as remote pointcut (join point depending on execution host), distributed advice (advice executed on a remote host), migration of aspects, asynchronous and synchronous aspects, distributed control flow. Most of those systems are based on Java and RMI in order to promote the role of AOP on commonly-used large-scale distributed applications. But the temptation of using a rich language to develop interesting applications has a fundamental drawback: it makes almost impossible the definition of a formal semantics for distributed aspects. Therefore, to date, no theory of distributed aspects has been developed.

In this paper, we propose a theory for distributed AOP based on a well formalized distributed language, the distributed join calculus [3]. The join calculus, founded on the chemical abstract machine and fully implemented in JoCaml, has been developed to fill in the gap between easy-to-reason-

about distributed languages such as the π-calculus and easy-to-program-with distributed languages such as Java-RMI. To stay closed to the object oriented tradition of AOP, we use an object oriented and distributed variant of the join calculus. This full variant is just a mix between the objective join calculus [5] and the distributed join calculus [4]. We define a notion of aspects on this calculus and give its formal semantics by altering the main reduction rule of the join calculus. The resulting aspect join calculus is provided with a type system that guarantees safety properties such as the absence mismatch in the number or type of arguments when an aspect returns to the original program using the keyword proceed, or the absence of host duplication in the network.

We additionally provide a translation of this aspect join calculus into the core join calculus. This translation implements the weaving algorithm that remains implicit in the abstract semantics and makes possible an implementation of the aspect join calculus directly in JoCaml. The correctness of the translation is given by a bisimilarity proof. Note that translating the aspect part of the calculus into the core calculus corresponds to the original version of AspectJ were aspects were directly compiled into Java code. We conclude the paper by showing how our calculus can be a cornerstone for the formulation of semantics for existing distributed oriented languages. Indeed, many feature of distributed AOP models can be defined in this simple language.

Section 2 presents the distributive and objective join calculus. Section 3 introduces the aspect join calculus. Section 4 defines the translation of the aspect join calculus into the core join calculus. Section 5 shows how basic concepts of distributed AOP can be described in the aspect join calculus. The reader that wants a quick overview of the aspect join calculus should only read Sections 3 and 5.

2. THE DISTRIBUTED OBJECTIVE JOIN-CALCULUS

The original version of the join calculus is a simple name-passing calculus related to the π-calculus but with a functional flavor [3, 4]. In this calculus, communication channels are statically defined: channels are created together with a set of reaction rules that specify, once and for all, how messages sent on these names will be synchronized and processed. We decide here to present an object-oriented version of the join-calculus [5] with an explicit notion of location to account for distribution [4].

2.1 Message passing and internal states

Before going into the details of the distributed objective join calculus, we begin with the example of the class *buffer* presented in [5]. The basic operation of the join-calculus is asynchronous message passing and, accordingly, the definition of an object describes how messages received on some labels can trigger processes. For instance, the term

$$\mathsf{obj}\ \mathit{continuation} = \mathit{reply}(n) \triangleright \mathit{out}.\mathit{print\_int}(n)$$

defines an object that reacts to messages on its own label *reply* by sending a message with label *print_int* and content n to an object named *out* that prints integers on the terminal.

$\mathrm{fn}(l(\vec{v}))$	=	$\{v_i / i \in I\}$
$\mathrm{fn}(M \,\&\, M')$	=	$\mathrm{fn}(M) \cup \mathrm{fn}(M')$
$\mathrm{fn}(M \triangleright P)$	=	$\mathrm{fn}(P) \setminus \mathrm{fn}(M)$
$\mathrm{fn}(D \,\mathsf{or}\, D')$	=	$\mathrm{fn}(D) \cup \mathrm{fn}(D')$
$\mathrm{fn}(H[D : P])$	=	$\mathrm{fn}(D) \cup \mathrm{fn}(P)$
$\mathrm{fn}(c)$	=	$\{c\}$
$\mathrm{fn}(0)$	=	$\emptyset$
$\mathrm{fn}(\mathsf{self}(z)\ D)$	=	$\mathrm{fn}(D) \setminus \{z\}$
$\mathrm{fn}(x.M)$	=	$\{x\} \cup \mathrm{fn}(M)$
$\mathrm{fn}(\mathsf{go}\, H; P)$	=	$\mathrm{fn}(P)$
$\mathrm{fn}(P \,\&\, Q)$	=	$\mathrm{fn}(P) \cup \mathrm{fn}(Q)$
$\mathrm{fn}(\mathsf{obj}\, x = C\, \mathsf{init}\, P\, \mathsf{in}\, Q)$	=	$(\mathrm{fn}(C) \cup \mathrm{fn}(P) \cup \mathrm{fn}(Q)) \setminus \{x\}$
$\mathrm{fn}(\mathsf{class}\, c = C\, \mathsf{in}\, P)$	=	$\mathrm{fn}(C) \cup (\mathrm{fn}(P) \setminus \{c\})$

Figure 1: Definition of free names $\mathrm{fn}(\cdot)$

But labels may also convey messages representing the internal state of an object, rather than an external method call. This is the case of label *some* in the following definition of a buffer :

$$\begin{array}{l}\mathsf{obj}\ \mathit{buffer} = \\ \quad \mathit{put}(n) \,\&\, \mathit{empty}() \triangleright \mathit{buffer}.\mathit{some}(n) \\ \mathsf{or}\ \mathit{get}(r) \,\&\, \mathit{some}(n) \triangleright r.\mathit{reply}(n) \,\&\, \mathit{buffer}.\mathit{empty}() \\ \mathsf{init}\ \mathit{buffer}.\mathit{empty}()\end{array}$$

Such a buffer can either be empty or contain one element. The state is encoded as a message pending on *empty* or *some*, respectively. Object *buffer* is created empty, by sending a first message on *empty* in the (optional) init part of the obj construct.

In our definition of pointcut for distributed aspects, we will have to consider labels that are common to different objects. This will be the case if we want to define a replication buffer aspect that will intercept the synchronization on the label *get* on any *buffer* object. This means that we need a notion of class instantiations. A buffer can then be defined as the instantiation of a class *buffer*:

$$\begin{array}{l}\mathsf{class}\ \mathit{buffer} = \mathsf{self}(z) \\ \quad \mathit{put}(n) \,\&\, \mathit{empty}() \triangleright z.\mathit{some}(n) \\ \mathsf{or}\ \mathit{get}(r) \,\&\, \mathit{some}(n) \triangleright r.\mathit{reply}(n) \,\&\, z.\mathit{empty}() \\ \mathsf{in}\ \mathsf{obj}\ b = \mathit{buffer}\ \mathsf{init}\ b.\mathit{empty}()\end{array}$$

Note that to keep the buffer object consistent, there should be a single message pending on either *empty* or *some*. This remains true as long as external processes cannot send messages on these labels directly. This can be enforced by a privacy discipline, as described in [5]. Nevertheless, we found it superfluous to describe the basics of distributed aspects.

2.2 Syntax

We use four disjoint countable sets of identifiers for object names $x, y, z \in \mathcal{O}$, classes $c \in \mathcal{C}$, labels $l \in \mathcal{L}$ and hosts $H \in \mathcal{H}$. Tuples are written $(v_i)^{i \in I}$ or simply $\vec{v}$. The grammar of the distributive objective join calculus is given in Figure 2; it has syntactic categories for configurations C, processes P, definitions D, and patterns M. A reaction rule $M \triangleright P$ associates a pattern M with a guarded process P.

$P ::=$		**Processes**
	0	null process
	$x.M$	message sending
	$\mathsf{go}\, H; P$	migration request (on host H)
	$P \,\&\, P$	parallel composition
	$\mathsf{obj}\, x = C\, \mathsf{init}\, P\, \mathsf{in}\, P$	object definition
	$\mathsf{class}\, c = C\, \mathsf{in}\, P$	class definition
$C ::=$		**Classes**
	c	class variable
	$\mathsf{self}(z)\ D$	self binding
$D ::=$		**Definitions**
	$M \triangleright P$	reaction rule
	$H[D : P]$	sub-location (named H)
	$D\, \mathsf{or}\, D$	disjunction
$M ::=$		**Patterns**
	$l(\vec{v})$	message
	$M \,\&\, M$	synchronization

Figure 2: Syntax for the core objective join calculus

Every message pattern $l(\vec{v})$ in M binds the object names $\vec{v}$ with scope P. In the join-calculus, it is required that every pattern M guarding a reaction rule be linear, that is, labels and names appear at most once in M. Class definitions $\mathsf{class}\, c = C\, \mathsf{in}\, P$ are the only binders for class names c, with scope P. The scoping rules appear in Figure 1. In addition, the object definition $\mathsf{obj}\, x = C\, \mathsf{init}\, P\, \mathsf{in}\, Q$ binds the name x to C. The scope of x is every guarded process in C (here x means "self") and the processes P and Q. The term $H[D : P]$ hosts the definition D and process P at location H. Migration request is described by $\mathsf{go}\, H; P$. Free names in processes and definitions, written $\mathrm{fn}(\cdot)$, are defined accordingly; a formal definition of free names appears in Figure 1. We suppose that class name definitions are unique. Objects are taken modulo renaming of bound names (or α-conversion).

2.3 Semantics

The operational semantics is given as a *reflexive chemical abstract machine* [3]. Each rewrite rule applies to configurations of objects and processes (usually called *chemical solutions*). A machine $\mathcal{D} \Vdash^{\varphi} \mathcal{P}$ consists of a set of named definitions $\mathcal{D}$ (representing objects in the machine) and of a multiset of processes $\mathcal{P}$ running in parallel at location φ. We write $x.D$ for a named definition in $\mathcal{D}$, and always assume that there is at most one definition for x in $\mathcal{D}$. Chemical reductions are obtained by composing rewrite rules of two kinds: structural rules $\equiv$ represent the syntactical rearrangement of terms; reduction rules $\longrightarrow$ represent the basic computation steps. The rules for the objective join calculus are given in Figure 3, with side conditions for rule Red: σ is a substitution with domain $\mathrm{fn}(M)$; the processes $M\sigma$ and $P\sigma$ denote the results of applying σ to M and P, respectively.

Rules Par and Nil make parallel composition of processes associative and commutative, with unit 0. Rule Obj describes the introduction of an object (up-to α-conversion, we can consider that any definition of an object x appears only once in a configuration). Rule Join gathers messages sent to the same object. Rule Red states how messages can be jointly consumed and replaced by a copy of a guarded process, in which the contents of these messages are substituted for the formal parameters of the pattern. In chemical semantics, each rule usually mentions only the components that participate to the rewriting, while the rewriting applies to every chemical solution that contains them. More explicitly, we provide two context rules Context and Context-Obj. In Rule Context, the symbol $\longrightarrow\ /\ \equiv$ stands for either $\longrightarrow$ or $\equiv$. In Rule Context-Obj, the side condition $x \notin \mathrm{fn}(D) \cup \mathrm{fn}(P)$ prevents name capture when introducing new objects (the sets $\mathrm{fn}(D)$ and $\mathrm{fn}(P)$ are defined in Figure 1). Rule Comm is reminiscent of distributed systems, where routing is a different step from actual computation. This rule states that a message emitted in a given location φ on a channel name x that is remotely defined can be forwarded to the machine at location ψ that contains the definition of x. Later on, this message can be used within ψ to assemble a pattern of messages and to consume it locally, using a local React step. Rule Loc introduces a new machine at sub-location H of φ with D as initial definitions and P as initial processes. The machine at φ continues but discharged of the sublocation H. The side condition "H frozen" means that there is no other machine of the form $\Vdash^{\varphi H \psi}$ in the configuration. The notation $\{x.D\}$ and $\{P\}$ states that there are no extra definitions or processes machine at location φH. Finally, rule Move gives the semantics of migration. A sublocation φH_1 of φ wants to move to a sublocation ψH_2 of ψ. On the right hand side, the machine $\Vdash^{\varphi}$ is fully discharged of the location H_1. Note that P can be executed at any time, whereas Q can only be executed after the migration.

In the following, we consider only configurations where every name is defined in at most one local solution. This condition is preserved by the semantics, and simplifies the usage of rule Comm: for every message, the rule applies at most once, and delivers the message to a unique receiving location.

Structural rules

PAR
$\Vdash^{\varphi} P \,\&\, Q \quad\equiv\quad \Vdash P, Q$

NIL
$\Vdash^{\varphi} 0 \quad\equiv\quad \Vdash^{\varphi}$

OR
$x.(D \,\mathsf{or}\, D') \Vdash^{\varphi} \quad\equiv\quad x.D, x.D' \Vdash^{\varphi}$

JOIN
$\Vdash^{\varphi} x.(M \,\&\, M') \quad\equiv\quad \Vdash^{\varphi} x.M, x.M'$

OBJ
$\Vdash^{\varphi} \mathsf{obj}\, x = \mathsf{self}(z)\ D \,\mathsf{init}\, P \,\mathsf{in}\, Q \quad\equiv\quad x.D[x/z] \Vdash^{\varphi} P, Q$

LOC
$x.H[D : P] \Vdash^{\varphi} \quad\equiv\quad \{x.D\} \Vdash^{\varphi H} \{P\} \qquad (H \text{ frozen})$

Reduction rule

CLASS-RED
$\Vdash^{\varphi} \mathsf{class}\, c = C \,\mathsf{in}\, P \quad\longrightarrow\quad \Vdash^{\varphi} P[C/c]$

RED
$x.M \triangleright P \Vdash^{\varphi} x.M\sigma \quad\longrightarrow\quad x.M \triangleright P \Vdash^{\varphi} P\sigma$

COMM
$\Vdash^{\varphi} x.M \quad\|\quad x.D \Vdash^{\psi} \quad\longrightarrow\quad \Vdash^{\varphi} \quad\|\quad x.D \Vdash^{\psi} x.M$

MOVE
$x.H_1[D : (P \,\&\, \mathsf{go}\, H_2; Q)] \Vdash^{\varphi} \quad\|\quad \Vdash^{\psi H_2} \quad\longrightarrow\quad \Vdash^{\varphi} \quad\|\quad x.H_1[D : (P \,\&\, Q)] \Vdash^{\psi H_2}$

Context rules

CONTEXT
$$\frac{\mathcal{D}_0 \Vdash^{\varphi} \mathcal{P}_1 \longrightarrow / \equiv\ \mathcal{D}_0 \Vdash^{\varphi} \mathcal{P}_2}{\mathcal{D}, \mathcal{D}_0 \Vdash^{\varphi} \mathcal{P}_1, \mathcal{P} \longrightarrow / \equiv\ \mathcal{D}, \mathcal{D}_0 \Vdash^{\varphi} \mathcal{P}_2, \mathcal{P}}$$

CONTEXT-OBJ
$$\frac{\Vdash^{\varphi} P \quad\equiv\quad x.D \Vdash^{\varphi} \mathcal{P}' \qquad x \notin \mathrm{fn}(\mathcal{D}) \cup \mathrm{fn}(\mathcal{P})}{\mathcal{D} \Vdash^{\varphi} P, \mathcal{P} \quad\equiv\quad \mathcal{D}, x.D \Vdash^{\varphi} \mathcal{P}', \mathcal{P}}$$

Figure 3: Chemical semantics

Rules for names and messages

OBJECT-VAR
$\Gamma, x : A \vdash x : A$

MESSAGE
$$\frac{\Gamma \vdash x : [l : \vec{A}; B]}{\Gamma \vdash x.l : \vec{A}}$$

Rules for patterns

PATTERN
$$\frac{(\Gamma \vdash x_i : A_i)^{i \in I}}{\Gamma \vdash l(x_i^{i \in I}) :: (l : A_i^{i \in I})}$$

SYNCHRONIZATION
$$\frac{\Gamma \vdash M_1 :: B_1 \qquad \Gamma \vdash M_2 :: B_2}{\Gamma \vdash M_1 \,\&\, M_2 :: B_1 \oplus B_2}$$

Rules for definitions and classes

REACTION
$$\frac{\Gamma' \vdash M :: B \qquad \Gamma, \Gamma' \vdash P \qquad \mathrm{dom}(\Gamma') = \mathrm{fn}(J)}{\Gamma \vdash M \triangleright P :: B}$$

DISJUNCTION
$$\frac{\Gamma \vdash D_1 :: B_1 \qquad \Gamma \vdash D_2 :: B_2}{\Gamma \vdash D_1 \,\mathsf{or}\, D_2 :: B_1 \oplus B_2}$$

LOC
$$\frac{\Gamma \vdash D :: B \qquad \Gamma \vdash P \qquad \Gamma \vdash H : \mathrm{loc}}{\Gamma \vdash H[D : P] :: B \oplus \{H : \mathrm{loc}\}}$$

SELF-BINDING
$$\frac{\Gamma, z : [B] \vdash D :: B}{\Gamma \vdash \mathsf{self}(z)\ D : [B]}$$

Rules for processes

NULL
$\Gamma; \vdash 0$

GO
$$\frac{\Gamma \vdash H : loc \qquad \Gamma \vdash P}{\Gamma \vdash \mathsf{go}\, H; P}$$

PATTERN
$$\frac{\Gamma \vdash x.l : A_i^{i \in I} \qquad (\Gamma \vdash x_i : A_i)^{i \in I}}{\Gamma \vdash x.l(x_i^{i \in I})}$$

PARALLEL
$$\frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1 \,\&\, P_2}$$

JOIN-PARALLEL
$$\frac{\Gamma \vdash x.M_1 \qquad \Gamma \vdash x.M_2}{\Gamma \vdash x.(M_1 \,\&\, M_2)}$$

CLASS
$$\frac{\Gamma \vdash C : [B] \qquad \Gamma, c : [B] \vdash P}{\Gamma \vdash \mathsf{class}\, c = C \,\mathsf{in}\, P}$$

OBJECT
$$\frac{\Gamma \vdash C : [B] \qquad \Gamma, x : [B] \vdash P \qquad \Gamma, x : [B] \vdash Q}{\Gamma \vdash \mathsf{obj}\, x = C \,\mathsf{init}\, P \,\mathsf{in}\, Q}$$

Rules for configurations

CONFIGURATION
$$\frac{(\Gamma \vdash \Vdash^{\varphi_i} P_i)_{i \in I} \qquad (\varphi_i)_{i \in I} \text{ is a tree}}{\Gamma \vdash \prod_{i \in I} \Vdash^{\varphi_i} P_i}$$

LOCAL SOLUTION
$$\frac{\Gamma \vdash P \qquad \Gamma \vdash H : \mathrm{loc}}{\Gamma \vdash \Vdash^{\varphi H} P}$$

Figure 4: Typing rules

2.4 An encoding of (a part of) Java

The objective join calculus is inherently asynchronous. Nevertheless, it is folklore that synchronous call can be encoded by use of continuations.

For instance, we can encode a large part of Java (without inheritance) in the objective join calculus. Fields are translated into labels containing the value of the field (as for labels *some* and *empty* of the class *buffer*). An expression

$$\texttt{x.m}(\vec{\texttt{v}});\texttt{P}$$

is translated into

$$\mathsf{obj}\ k = return() \triangleright P\ \mathsf{in}\ m(k, \vec{v})$$

A method

$$\texttt{m}(\vec{\texttt{v}})\ \{\texttt{return e}\}$$

using fields $f_1, \ldots, f_n$ is translated into the reaction rule

$$m(k, \vec{v})\ \&\ (f_i(x_i))_{i\in I} \triangleright k.return(e)\ \&\ (f_i(x_i))_{i\in I}$$

where k is the explicit continuation passed to m and e is a base expression. To illustrate this, we present the translation of the walking example of the seminal paper on Featherweight Java [6]. This example can be written, in absence of inheritance, as:

```
class Pair {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
    super(); this.fst=fst; this.snd=snd;
  }
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd);
  }
```

The class `Pair` has two fields `fst` and `snd`, an initialization method (also written `Pair`) and a method `setfst` that returns a new `Pair` with different first element. In the join calculus, this class becomes:

$$\begin{array}{l}\mathsf{class}\ Pair = \mathsf{self}(z)\\ \quad Pair(fst, snd) \triangleright z.fst(fst)\ \&\ z.snd(snd)\\ \mathsf{or}\ snd(snd)\ \&\ setfst(newfst, k) \triangleright z.snd(snd)\ \&\\ \qquad \mathsf{obj}\ x = Pair\ \mathsf{init}\ Pair(newfst, snd)\\ \qquad \mathsf{in}\ k.return(x)\end{array}$$

The two fields have been translated in two labels fst and snd. This two labels are used in a synchronization pattern to communicate the internal value of the fields. On the right hand side of a rule, those labels have to be present again to maintain the value of the fields in the local solution. The initialization is performed by the reaction rules $Pair(fst, snd) \triangleright z.fst(fst)\ \&\ z.snd(snd)$. The method `setfst` is now seen as a reaction that synchronizes the label $setfst$ with the label snd (to get the value of the field `snd`) and then produces the label snd again and creates a new pair object that is passed to the continuation k. So the continuation comes in the picture when a method returns.

Naturally, we can encode much more than Featherweight Java, the pure functional part of Java. As sketched in Section 2.1 with the class *buffer*, it is also possible to capture the imperative flavour of Java by encoding internal states with labels. For instance, we refer the interesting reader to [4] for a description of references in the join calculus.

Note that if we add inheritance in the join calculus, as done in [5], then most of Java can be encoded.

2.5 Migration in the calculus

In contrast with some models of distributed systems [12], the explicit routing of messages is not described by the calculus. Rule Comm applies at most once and delivers the message to a unique receiving location. Since every object is located, the interpretation of the remote object mechanism of Java-RMI, implemented using *stub* and *skeleton*, is transparent.

Rule Move says that the migration process on the network is based on sub-locations but not objects nor processes. When a process decides to move, it moves with all the definitions and processes present at the same sub-location. Nevertheless, we can encode object/process migration by defining a fresh sub-location and uniquely attaching an object/process to it. Then the migration of the sub-location will be equivalent to the migration of the object/process.

In contrast with Java-RMI, there is no mechanism of serialization, with copy before migration. Jeffrey has considered a distributed object calculus where serialization is explicit [7]. Nevertheless, the continuation passing style encoding presented above is closed to the semantics given by Ahern and Yoshida for method invocation in a core Java with RMI [1]. Indeed, most of the mechanism of Java-RMI (as formalized in [1]) can be encoded in the distributed join calculus. One just has to create a new location for each object such that the migration of an object becomes the migration of the sub-location attached to this object. So except from serialization concerns, the distributed join calculus and Java-RMI have the same flavour.

2.6 Typing

The grammar of type expression is given by

$$\begin{array}{l}A ::= \texttt{int} \mid \texttt{bool} \mid [B]\\ B ::= \emptyset \mid l : \vec{A}; B\end{array}$$

We build types out of the two base types `int` for natural numbers and `bool` for booleans. As we want to keep our core language and typing derivation as simple as possible, we do not consider polymorphism in this paper, although polymorphism *à la ML* can be defined without difficulty [5].

Object types $[B]$ collect the types of labels. For instance, the type of the object *continuation* is

$$continuation : [reply : \texttt{int}]$$

and the object *buffer* can be typed for example with

$$buffer : [put : \texttt{int}; empty : (); get : [reply : \texttt{int}]; some : \texttt{int}].$$

Again, the absence of polymorphism certainly make this type system look a bit odd. Indeed, one could expect the

type system to be able to type the object *buffer* with the polymorphic type

$$\forall\alpha,\beta : [put : \alpha; empty : (); get : [reply : \alpha; \beta]; some : \alpha].$$

We accept this peculiar nature for the sake of simplicity.

A typing judgment for a process P is of the form

$$\Gamma \vdash P$$

where Γ is the typing environment. As usual in typing of object oriented languages (see for instance [6]), we make use of a class table CT that collects every class definition. To lighten the notation in what follows, we always assume a fixed class table CT. A typing judgment for messages has the form

$$\Gamma \vdash M :: B$$

where we write $B_1 \oplus B_2$ for the union of B_1 and B_2, with the statement that B_1 and B_2 coincide on their common labels. We do not describe in detail the typing rules given in Figure 4 and we refer the reader to [4, 5] for more details.

Note that rules CONFIGURATION and LOCAL SOLUTION are only defined in case of configurations of the form $\Vdash^{\varphi} P$. Up-to structural congruence, a more general rule can be easily deduced.

2.7 Safety property

We now present the interest of types with respect to the chemical semantics. Namely, the type system ensures that no well-typed configurations can present a *runtime failure* in the sense defined below. To state this safety property, we first need subject reduction for our type system.

THEOREM 1 (SUBJECT REDUCTION). *Chemical reductions preserve typing : let $\mathcal{C}$ be configuration*

$$\mathcal{C} = \mathcal{D}_1 \Vdash^{\varphi_1} \mathcal{P}_1 \parallel \cdots \parallel \mathcal{D}_n \Vdash^{\varphi_n} \mathcal{P}_n,$$

Γ an environment. If $\Gamma \vdash \mathcal{C}$ and $\mathcal{C} \equiv \mathcal{C}'$ or $\mathcal{C} \longrightarrow \mathcal{C}'$, then there exists an environment Γ' such that $\Gamma' \vdash \mathcal{C}'$.

PROOF. The proof goes by induction on structural and reduction rules. A detailed description of this induction (except for locations) can be found in [5], Appendix B. A formal (and more general) treatment of location can be found in [13]. □

Runtime failure: We say that a configuration $\mathcal{C}$ fails when one of the following holds:

- ARITY MISMATCH: for some message $l(\vec{v})$ in $\mathcal{P}_i$, $l(\vec{u})$ appears in a pattern of $\mathcal{D}_j$ with different arities or types for $\vec{v}$ and $\vec{u}$.
- HOST DUPLICATION: there is two machines at the same location, for instance $\mathcal{D}_i \Vdash^{\varphi H} \mathcal{P}_i$ and $\mathcal{D}_j \Vdash^{\psi H} \mathcal{P}_j$
- IMPOSSIBLE MIGRATION: there is a process $\mathtt{go}\, H; P$ where H is not a location name.

The first runtime failure is usual and does not rely on distribution. It is about method invocation with a right number of arguments and the right types. The second one models IP address duplication. Avoiding this duplication in a configuration makes the semantics of migration unambiguous. The last failure corresponds to a wrong IP address when trying to move a process to another hosting machine.

THEOREM 2 (SAFETY). *Well-typed configurations do not fail.*

PROOF. We check that no failure listed above can occur for a well-typed configurations. The conclusion then follows from subject reduction. □

2.8 Basic extension of the language

To make the definition of processes more digestible in the rest of the paper, we flavor our language with some usual primitive. We add a notion of string and list of strings, equality test on strings, a traditional `let` binding, a conditional `if then else` instruction and a particular variable `localhost` that represents the current location on which a process is executing. Except for equality testing, all those constructions can be easily encoded into the join-calculus. A string is written "name", and a list is written $[l_1; \ldots; l_n]$ with concatenation noted $L_1 \cdot L_2$, constructor $l :: L$ and empty list $[\,]$.

Since the join-calculus has lexical scoping, programs being executed on different machines do not initially share any port name; therefore, they would normally not be able to interact with one another. To bootstrap a distributed computation, it is necessary to exchange a few names, and this is achieved using a built-in library called the name server. Once this is done, these first names can be used to communicate some more names and to build more complex communication patterns.

The interface of the name server mostly consists of two functions to register and look up arbitrary values in a "global table" indexed by plain strings. A process that wants to associate the string "foo" to the name x simply executes ns.register(x,"foo"). Then, an object y on a remote location can ask to the name server which name is associated to "foo" by executing ns.lookup("foo", $y, return$). The name server will then send the name x on $y.return$.

3. A JOIN CALCULUS WITH ASPECTS

In Java, the basic event of the language is the call of a method. In ML, the basic event of the language is the application of a function. Therefore, it is not surprising to see those events as basic blocks for the definition of join points in AspectJ or AspectML. To define the notion of aspects in the objective join calculus, we must understand what is a basic event of this language. It makes no doubt that it consists in the application of Rule RED to a synchronization pattern M. So a pointcut in the aspect join calculus will rely on a synchronization pattern.

Before going into the details of the syntax, we present a basic example of distributed aspects as defined in the language AWED [8], and show how we want to define them in aspect join calculus. In AWED, one likes to define an aspect for buffer replication in the following manner :

$Pc ::=$		**Pointcuts**
	$\mathsf{rule}(c.M)$	call, arguments
	$Pc \wedge Pc^{opt}$	conjunction
$Pc^{opt} ::=$		**Optional Pointcuts**
	$\mathsf{flow}(l)$	control on flow
	$\mathsf{host}(H)$	control on host
	$\neg Pc^{opt}$	negation
	$Pc^{opt} \wedge Pc^{opt}$	conjunction
$Ad ::=$		**Advice bodies**
	$\ldots$	other process definitions
	$\mathsf{proceed}(\vec{v})$	proceed
$A ::=$		**Advices and aspects**
	$Pc\ \{Ad\}$	advice definition
	$\mathsf{aspect}\ a = C\ \mathsf{init}\ P\ \mathsf{intercept}\ (Pc_i\ \{Ad_i\})_{i\in I}$	aspect definition

Figure 5: Syntax for distributed aspects

```
all aspect BufferReplication{
   pointcut bufferPcut(Object k, Object o):
    call(* Buffer.put(Object,Object))
        && args(k,o) && !on(jphost) &&
    !within(BufferReplication);

   before(Object k, Object o): bufferPcut(k,o){
     Buffer.getInstance().put(k,o); }
}
```

This aspect realizes a replication of the buffer each time the method *buffer.put* is called. The replication takes place on every machine except the machine where the method *put* has been caught (!`on(jphost)`). To prevent from an infinite replication of buffers, the condition !`within`(*bufferRepl*) guarantees that the join point is not inside an execution of the aspect *bufferRepl*. In our setting, using the definition of the class *buffer* given in Section 2.1, we can write a similar aspect:

$$\Vdash^{\varphi} \mathsf{aspect}\ \mathit{bufferRepl} =$$
$$\mathsf{intercept} : \mathsf{rule}(\mathit{buffer}.(\mathit{put}(n)\ \&\ \mathit{empty}())) \wedge \neg\, \mathsf{host}(\varphi)$$
$$\{\mathsf{obj}\, b = \mathit{buffer}\, \mathsf{init}\, b.\mathit{empty}()\, \mathsf{in}(b.\mathit{put}(n)\ \&\ \mathsf{proceed}(n))\}$$

The join point now relies on the interception of the synchronization pattern $put(n)$ & $empty()$ of the class *buffer*. The advice body makes an explicit use of the keyword proceed. This is because `Before` advices do not exist in the aspect join calculus. Indeed, in an asynchronous setting, there is no notion of before or after the execution of a method body, so the only possible advice is something that looks like the `Around` advice of AspectJ. Note that it might seem unsatisfactory to define an aspect by explicitly mentioning the channel *empty*. This can be handled with a careful management of privacy that we don't want to consider here. The basic idea is to say that an aspect should only mention public labels and will be implicitly quantified over all private labels. In that setting, the pattern of interest for the advice aspect above would simply be defined by $\mathsf{rule}(\mathit{buffer.put}(n))$.

The condition $\neg\, \mathsf{host}(\varphi)$ guarantees that the replication does not hold when the reaction is taking place on a sub-location of the location where the aspect has been hosted. In particular, this prevent the aspect to be deploy on its own invocation of method *put*.

So this single aspect behave as a single aspect `BufferReplication`. Nevertheless, if one tries to define such an aspect on each host of interest, then the aspects will interfere and recursively copy the buffer copied by the other aspect. In AWED, the way we can prevent this livelock to append is by the use of `within`. Unfortunately, this notion of "within the execution of an aspect" makes no sense in an asynchronous setting. Nevertheless, we will see later that `within` can be encoded with flow when we are in a fully synchronous setting.

3.1 Syntax

Figure 5 presents the syntax for distributed aspects in the objective join calculus. We use a countable set of identifiers for aspect names $a \in \mathcal{A}$,

An aspect $\mathsf{aspect}\, a = C\, \mathsf{init}\, P\, \mathsf{intercept}\ (Pc_i\ \{Ad_i\})_{i\in I}$ consists in a class definition C, an initialization process P and a list of advices $Pc_i\ \{Ad_i\}$. The class C and process P are here to define inner fields and methods of an aspect seen as an object. Advices are defined by a pointcut Pc and an advice body Ad.

A pointcut is defined by a term $\mathsf{rule}(c.M)$ that selects any reaction rule that has the pattern M of the class c as left hand part. A pointcut can also be defined by conditions on the history of reaction rule (flow), on the host where the join point has been selected (host). Finally, a pointcut can be constructed by negations and conjunctions of those two conditions. Note that in contrast with AspectJ, we do not need to type the intercepted pattern in $\mathsf{rule}(c.M)$ as we explicitly mention the class to which M belongs.

An advice body Ad is a process to be executed when then rule is intercepted. This process may contain the special keyword proceed. Definition of processes are extended with advices and aspects.

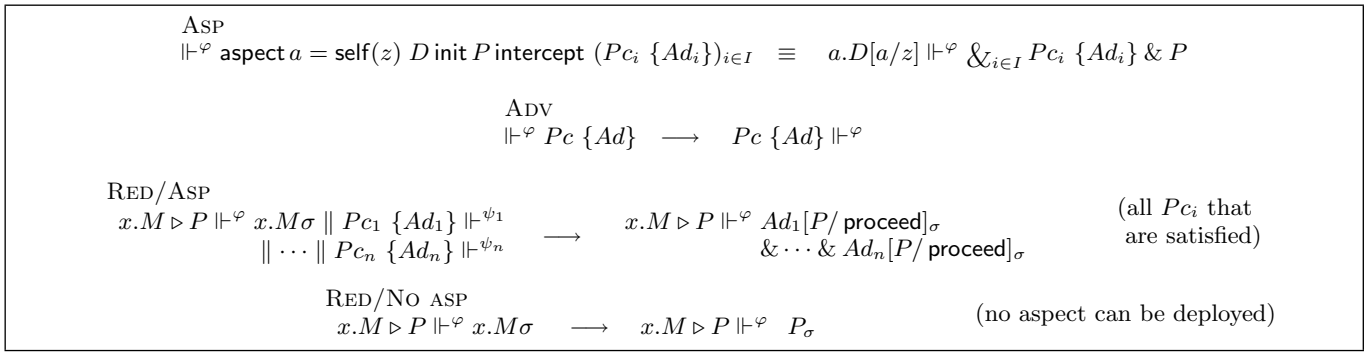

Figure 6: Semantics of aspects

3.2 Semantics

Figure 6 presents the semantics of aspects. All rules of Figure 3 are conserved, expect for Rule RED that is split in two rules. Rule ASP describes the introduction of an aspect. It is similar to Rule OBJ. Rule ADV corresponds to the activation of an advice. Note that activation of advices is asynchronous.

Rule RED/ASP defines the modification of Rule RED in presence of aspects. If an advice definition $Pc\ \{Ad\}$ has a pointcut Pc that is satisfied, then the advice Ad is deployed while substituting the process P to the keyword proceed. Note that all advices that have a satisfied pointcut are executed in parallel. The side condition of this rule is that $Pc_1, \ldots, Pc_n$ are all the pointcuts that are satisfied at this join point. For a given pointcut Pc, this means that it intercepts the right pattern and that the condition Pc^{opt} is satisfied. The proposition $\mathsf{flow}(l)$ is satisfied when the message l appears in the reaction tree of the intercepted reaction rule. The proposition $\mathsf{host}(H)$ is satisfied when the intercepted reaction rule is executed on a sub-location of H.

Rule RED/NO ASP is a direct reminiscence of Rule RED in case where no aspect can be deployed.

3.3 Typing rules

In contrast with [11], we do not type pointcuts. They are just boolean expressions that describe the applicability of an advice.

Rule ASPECT is similar to Rule OBJECT, it further checks that all advices are well-typed. Rule ADVICE checks that the body of the advice is well typed once we have substituted the keyword proceed by the pattern $x.M$ of any object x of the class c.

Of course, substituting proceed by the pattern $x.M$ is not correct with respects to the semantics of proceed. Indeed, proceed should rather be replaced by the right hand of the reaction rule involving M. Nevertheless, as one consider typing judgment only, this substitution is safe and makes the typing derivation easier to define.

We get subject reduction and a safety theorem similar to Theorem 2. In particular, we have the following corollary, which is not the case in AspectJ,

COROLLARY 1. *A well-typed configuration makes use of* proceed *with the good number and types of arguments.*

4. REDUCTION TO THE CORE CALCULUS

In this section, we present a translation of the aspect join calculus into the core join calculus. In this way, we give an implementation of the weaving algorithm with a bisimilarity proof that this translation has the same behaviour than the original configuration with aspects. A non-objective version of the aspect join calculus can thus be implemented directly in Jocaml (http://jocaml.inria.fr/), a combination of Ocaml and the join calculus. It will provide an expressive distributed AOP platform were formal reasoning about aspect properties is possible. This implementation has not been develop yet and is the subject of an ongoing work.

Given a typed aspect join calculus configuration

$$\emptyset \vdash (\quad \Vdash^{\varphi_1} P_1) \parallel \cdots \parallel (\quad \Vdash^{\varphi_n} P_n),$$

we construct a distributed join calculus configuration without aspects by translating processes and aspects and introducing a weaver process W

$$\emptyset \vdash (\quad \Vdash^{\varphi_1} [\![P_1]\!]) \parallel \cdots \parallel (\quad \Vdash^{\varphi_n} [\![P_n]\!]) \parallel (\quad \Vdash^{H_W} W).$$

The idea is to introduce an explicit join point in every reaction rule. This join point consists in a dialogue with the weaver to wonder whether advices could intercept the rule and be deployed. To make the weaver able to decide if an advice can be deployed or not, the message sends by the process to the weaver must contains information about flow and host. We do this by passing the flow information has an argument all over the execution.

4.1 The translation processes

The translation of processes is quite transparent. Any reaction rule $M \triangleright P$ is replaced by a call to the weaver and a return method that performs the actual computation P. The flow of previous synchronized labels is passed has an argument for every label. For example, the class *buffer* would by translated by

$$\textsc{Aspect}\quad \frac{\Gamma \vdash C : [B_1] \qquad \Gamma, a : [B_1] \vdash P \qquad (\Gamma, a : [B_1] \vdash Pc_i\ \{Ad_i\})_{i\in I}}{\Gamma \vdash \mathsf{aspect}\ a = C\ \mathsf{init}\ P\ \mathsf{intercept}\ (Pc_i\ \{Ad_i\})_{i\in I}}$$

$$\textsc{Advice}\quad \frac{CT \vdash c : [B_1 \oplus B_2] \qquad \Gamma' \vdash M :: B_1 \qquad \Gamma, \Gamma', x : [B_1] \vdash Ad[x.M\ /\ \mathsf{proceed}] \qquad x \text{ fresh in } \Gamma, \Gamma'}{\Gamma \vdash (\mathsf{rule}(c.M) \wedge Pc^{opt})\ \{Ad\}}$$

Figure 7: Typing rules for aspect join calculus

$$\begin{array}{l}
\mathsf{class}\ \mathit{buffer} = \mathsf{self}(z)\\
\quad \mathit{put}(f_1, n)\ \&\ \mathit{empty}(f_2)\ \&\ \mathrm{Weaver}_1(w) \triangleright\\
\qquad w.\mathit{weave}(z, 1, f_1 \cdot f_2 \cdot [\text{"}\mathit{put}\text{"}, \text{"}\mathit{empty}\text{"}],\\
\qquad \mathtt{localhost}, n)\ \&\ z.\mathrm{Weaver}_1(w)\\
\mathsf{or}\ \mathit{resume}_1(f, n) \triangleright z.\mathit{some}(f, n)\\
\mathsf{or}\ \mathit{get}(f_1, r)\ \&\ \mathit{some}(f_2, n)\ \&\ \mathrm{Weaver}_2(w) \triangleright\\
\qquad w.\mathit{weave}(z, 2, f_1 \cdot f_2 \cdot [\text{"}\mathit{get}\text{"}, \text{"}\mathit{some}\text{"}],\\
\qquad \mathtt{localhost}, r, n)\ \&\ z.\mathrm{Weaver}_2(w)\\
\mathsf{or}\ \mathit{resume}_2(f, r, n) \triangleright r.\mathit{reply}(f, n)\ \&\ z.\mathit{empty}(f)
\end{array}$$

Each reaction rule of the class *buffer* has been divided into two reactions. The first one sends a message to the weaver with label *weave*. The original reaction is blocked. When an advice do a proceed (in case of Rule RED/ASP) or when the weaver detects that no aspect can be deployed (in case of Rule RED/NO ASP), the message resume_1 is sent to *buffer* and the original reaction is performed (with potentially new variables). The flow of performed reactions is passed along reaction rules with the used of dedicated variable $f_1, f_2, \ldots$ The location were the reaction rule is performed is sent using the `localhost` keyword. The location of the weaver is stored in the label Weaver.

Figure 8 describes the details of the translation for processes. Note that each object initializes its own labels Weaver_M for each pattern M appearing in C. To construct the flow information, we use the function `listof` that builds out a list of messages from a pattern and an object.

$$\begin{array}{c}
\mathtt{listof}(l(\vec{v}), x) = [\text{"x.l(}\mathrm{v_1}, \ldots, \mathrm{v_n}\text{)"}]\\
\mathtt{listof}(M_1\ \&\ M_2, x) = \mathtt{listof}(M_1, x) \cdot \mathtt{listof}(M_2, x)
\end{array}$$

Note that in this translation, "P" stands for a (supposed to be unique) string identifier attached to the process P. This string informs weaver and advices that they must send a message on the label $\mathit{resume}_{M,\text{"}P\text{"}}$ to proceed.

4.2 The weaver

We define, for each possible pattern M present in a class defined in CT, a weaver W_M dedicated to M. To know which advices can be deployed, the weaver maintains the list of all advices Ad (first argument of *adviceList*) that have a pointcut that intercepts the rule M. The weaver also stores aspects defining the advices of Ad by the list A (second argument of *adviceList*) and the associated pointcut list Pc (third argument of *adviceList*). Note that pointcuts and advices can not be passed directly as arguments of messages but we take this liberty for clarity (this approximation can be solved by an encoding).

When an aspect sends a message $\mathit{add\_advice}(a, pc, ad)$ to register a new advice, the weaver updates *adviceList* accordingly. When the message $\mathit{weave}(x, p, f, H, \vec{v})$ is captured, the weaver tests the validity of the list pointcuts Pc of the different advices described by the list Ad by executing

$$\mathtt{let}\ (B = \mathit{test}(Pc, f, H)).$$

We do not detail here the test function *test* as it basically performs the boolean test described in each Pc_i^{opt} based on the flow information f and the host information H. Note that an other possibility would have been to differ the test to each aspect, but then it raises synchronization issue that would have make the translation much harder.

If no advice can be deployed (`is_false`(B) is true) then the weaver executes the original process by sending the message $\mathit{resume}_{M,p}(f, \vec{v})$ to the object x. This corresponds to Rule RED/NO ASP. Otherwise, the weaver asynchronously deploys any applicable advice Ad_i (that is advice for which b_i is true) by sending the message $a_i.\mathit{deployAd}_i(x, p, f, \vec{v}))$ to the associated aspect a_i. This corresponds to Rule RED/ASP.

The central weaver W is just the parallel composition of each local weaver

$$W = \mathop{\&}_{M\in CT} W_M$$

for every pattern M appearing in the class table CT.

4.3 The translation of aspects

In our translation, an aspect is seen as a classical object that receives messages from the weaver to execute particular methods that are reminiscences of advice bodies. This is closed to the CaesarJ point of view that aspects are just objects that happen to have some pointcuts as attributes [2]. A call to proceed is translated into a message $\mathit{resume}_{M,\text{"}P\text{"}}(f, \vec{v})$ that is sent to the object whose pattern M has been intercepted. More precisely, given an aspect

$$\mathsf{aspect}\ A = \mathsf{self}(z)\ D\ \mathsf{init}\ P\ \mathsf{intercept}\ (Pc_i\ \{Ad_i\})_{i\in I},$$

the translation produces the object

$$\begin{array}{l}
\mathsf{obj}\ a = \mathsf{self}(z)\ [\![D]\!]\\
\qquad \mathsf{or}\ \mathit{deployAd}_i(x, p, f, \vec{v})\ \triangleright\ [\![Ad_i\{z/a\}]\!]_{M_i}\\
\qquad \mathsf{or}\ \ldots\\
\qquad \mathsf{init}\ [\![P]\!]\ \&\ \mathit{Add}(Ad, a, Pc)
\end{array}$$

where the translation of processes is extending to proceed by

$$[\![\mathsf{proceed}(\vec{v})]\!]_M \equiv\ x.\mathit{resume}_{M,p}(f, \vec{v})$$

The definitions D of the object part of a are extended with reaction rules that deploy an advice Ad_i when the weaver send the message $\mathit{deployAd}_i(x, p, f, \vec{v})$. The initialization sends asynchronously every advice appearing in the definition of the aspect a to its associated weaver by using the

Rules for processes

$$[\![0]\!] \equiv 0 \qquad [\![x.M]\!] \equiv x.[\![M]\!] \qquad [\![\mathsf{go}\, H; P]\!] \equiv \mathsf{go}\, H; [\![P]\!] \qquad [\![P_1 \,\&\, P_2]\!] \equiv [\![P_1]\!] \,\&\, [\![P_2]\!]$$

$$[\![\mathsf{obj}\, x = C\, \mathsf{init}\, P\, \mathsf{in}\, Q]\!] \equiv \mathsf{obj}\, x = [\![C]\!]\ \mathsf{init}\ \&_{M \in C}\, \mathrm{ns.lookup}(\text{"weaver\_M"}, x, \mathrm{Weaver}_M) \,\&\, [\![P]\!]\ \mathsf{in}\ [\![Q]\!]$$

$$[\![\mathsf{class}\, c = C\, \mathsf{in}\, P]\!] \equiv \mathsf{class}\, c = [\![C]\!]\, \mathsf{in} [\![P]\!]$$

Rules for definitions and classes

$$[\![M \triangleright P]\!]_z \equiv [\![M]\!]_1 \,\&\, \mathrm{Weaver}_M(w) \triangleright w.weave(z, \text{"}P\text{"}, f_1 \,.\, \cdots \,.\, f_{\#M} \,.\, \mathtt{listof}(M, z), \mathtt{localhost}, \vec{v_1}, \ldots, \vec{v_n}) \,\&\, z.\mathrm{Weaver}_M(w)$$
$$\mathsf{or}\ \ resume_{\text{"}P\text{"}}(f, \vec{v_1}, \ldots, \vec{v_n}) \triangleright [\![P]\!]$$

$$[\![H[D:P]]\!]_z \equiv H[[\![D]\!]_z : [\![P]\!]] \qquad [\![D_1\, \mathsf{or}\, D_2]\!]_z \equiv [\![D_1]\!]_z\, \mathsf{or} [\![D_2]\!]_z \qquad [\![c]\!] \equiv c \qquad [\![\mathsf{self}(z)\ D]\!] \equiv \mathsf{self}(z)\ [\![D]\!]_z$$

Rules for patterns

$$[\![l(\vec{v}) \,\&\, M]\!] \equiv l(f, \vec{v}) \,\&\, [\![M]\!] \qquad [\![l(\vec{v}) \,\&\, M]\!]_i \equiv l(f_i, \vec{v}) \,\&\, [\![M]\!]_{i+1}$$

Figure 8: Translation of processes

$$\begin{array}{lll}
\mathsf{obj} & W_M = & add_advice(ad, a, pc) \,\&\, adviceList(Ad, A, Pc) \triangleright adviceList(ad : Ad, a : A, pc : Pc) \quad (* \text{ store new advice } *) \\
 & & weave(x, p, f, H, \vec{v}) \,\&\, adviceList(Ad, A, Pc) \triangleright \quad (* \text{ receive join point } *) \\
 & & \quad \mathtt{let}\ (B = test(Pc, f, H))\ \mathtt{in}\ adviceList(Ad, A, Pc) \,\& \quad (* \text{ test applicability of aspects } *) \\
 & & \quad\ \ \mathtt{if\ is_false}(B)\ \mathtt{then}\ x.resume_{M,p}(f, \vec{v}) \quad (* \text{ if no aspect, return to join point } *) \\
 & & \quad\ \ \mathtt{else}\ \big(\&_{b_i \in B}\ \mathtt{if}\ b_i\ \mathtt{then}\ a_i.deployAd_i(x, p, f, \vec{v})\big) \quad (* \text{ forall advices if } Pc_i, \text{ deploy } Ad_i\ *) \\
 & \mathsf{init} & \mathrm{ns.register}(W_M, \text{"weaver_M"}) \,\&\, adviceList([\,], [\,], [\,]) \quad (* \text{ register weaver_M and initiliazes lists } *)
\end{array}$$

Figure 9: Definition of the weaver for pattern M

dedicated process $Add(Ad, a, Pc)$. We do not detail this process here has it is just a matter of bureaucracy using *add_advice*.

4.4 Bisimilarity of abstract and concrete definitions

The main interest of translating the aspect join calculus into the core join calculus is that it provides a direct implementation of the weaving algorithm that can be proved to be correct. As usual in concurrent programming languages, the correction of the algorithm is given by a proof of bisimilarity. Namely, we prove that the original configuration with aspects is bisimilar to the translated configuration that has no aspect. The idea of bisimilarity is to express that, at any stage of reduction, both configuration can performed the same actions in the future. More formally, in our setting, a simulation $\mathcal{R}$ is a relation between configurations such that when $\mathcal{C}_0\ \mathcal{R}\ \mathcal{C}_1$ and $\mathcal{C}_0$ reduces in one step to $\mathcal{C}'_0$, there exists $\mathcal{C}'_1$ such that $\mathcal{C}'_0\ \mathcal{R}\ \mathcal{C}'_1$ and $\mathcal{C}_1$ reduces to $\mathcal{C}'_1$. We illustrate this with the following diagram

$$\begin{array}{ccc}
\mathcal{C}_0 & \overset{\mathcal{R}}{\text{---}} & \mathcal{C}_1 \\
\big\downarrow & & \big\downarrow{}^{*} \\
\mathcal{C}'_0 & \underset{\mathcal{R}}{\text{---}} & \mathcal{C}'_1
\end{array}$$

A bisimulation is a simulation whose converse is also a simulation.

To relate a configuration $\mathcal{C}_0$ with its translation $[\![\mathcal{C}_0]\!]$, we need to tackle three difficulties:

1. During the evolution of $[\![\mathcal{C}_0]\!]$, auxiliary messages that have no correspondents in $\mathcal{C}$ are sent for communication between processes, weaver and aspects.

2. In the execution of $\mathcal{C}_0$, proceed is substituted by the process P to be executed, whereas in $[\![\mathcal{C}_0]\!]$, P is executed throw a communication with the object where the reaction has been intercepted.

3. Initial communications with the name server in $[\![\mathcal{C}_0]\!]$ disappears during the reduction.

To see the auxiliary communication as part of a reduction rule of the aspect join calculus, we define a notion of standard form for the translated configurations. Let

$$\mathbb{T} = \{\mathcal{C} \mid \exists \mathcal{C}_0, [\![\mathcal{C}_0]\!] \longrightarrow^* \mathcal{C}\}$$

be the set of configurations that comes from a translated configuration. We construct a rewriting system $\longrightarrow_{\mathbb{T}}$ for $\mathbb{T}$, based on the reduction rule of the join calculus. Namely, we take Rule RED restricted to the case were the pattern contains either of the dedicated labels: *weave*, *resume*, *deployAd*, *replyM* or *add_advice*. In $\mathbb{T}$, those labels only interact one-by-one with constant labels (a constant label is a label that appears identically on the left and right hand side of every reaction rule) such as Weaver or aspL(). So the order in which reaction rules are selected has no influence on the synchronized pattern, that is the rewriting system $\longrightarrow_{\mathbb{T}}$ is confluent. Furthermore, it is not difficult to check that this rewriting system is also terminating. Therefore, it makes sense to talk about the normal form of $\mathcal{C} \in \mathbb{T}$, noted $\widetilde{\mathcal{C}}$.

We note $\mathcal{C} \overset{\text{proc}}{\sim} \mathcal{C}'$ when $\mathcal{C}'$ is equal to $\mathcal{C}$ were every message $resume_{M,"P"}(f,\vec{v})$ is substituted by the process $P(\vec{v})$.

Given a configuration $\mathcal{C}_0$, we note $[\![\mathcal{C}_0]\!]_{\text{init}}$ the translated configuration where every initial communication with the name server has been performed. That is, every message of the form ns.lookup("a", x, l) and ns.register(a, "a") have been consumed.

THEOREM 3. *The relation* $\mathcal{R} = \{(\mathcal{C}_0, \mathcal{C}_1) \mid \widetilde{\mathcal{C}_1} \overset{\text{proc}}{\sim} [\![\mathcal{C}_0]\!]_{\text{init}}\}$ *is a bisimulation. In particular, any typed configuration is bisimilar to its translation.*

PROOF. The fact that $\mathcal{R}$ is a simulation just says that the communication between aspects, processes and the weaver simulates the abstract semantics of aspects. More precisely, we show that for any reduction $\mathcal{C}_0 \longrightarrow \mathcal{C}_0'$ using Rule RED/ASP, RED/NO ASP or ADV, one can find a corresponding reduction chains from $[\![\mathcal{C}_0]\!]_{\text{init}}$ to $[\![\mathcal{C}_0']\!]_{\text{init}}$

$$\begin{array}{ccccc} \mathcal{C}_0 & \overset{\mathcal{R}}{\text{---}} & \mathcal{C}_1 & \overset{*}{\underset{\mathbb{T}}{\longrightarrow}} \widetilde{\mathcal{C}_1} \overset{\text{proc}}{\sim} & [\![\mathcal{C}_0]\!]_{\text{init}} \\ \downarrow & & & & \vdots \\ \mathcal{C}_0' & \text{- - - -} \overset{\mathcal{R}}{\text{- -}} & \mathcal{C}_1' & \overset{\text{proc}}{\sim} & [\![\mathcal{C}_0']\!]_{\text{init}} \end{array}$$

Consider the case of Rule RED/ASP (the others are easier):

$$x.l(\vec{v}) \quad \longrightarrow \quad Ad_1[P/\,\mathsf{proceed}] \,\&\, \cdots \,\&\, Ad_n[P/\,\mathsf{proceed}]$$

This rule is simulated by the chain (we omit argument on the right for saving place)

$$\begin{array}{lll} x.l(f,\vec{v}) & \longrightarrow & w.weave \\ & \longrightarrow & a_1.deployAd_1 \,\&\, \cdots \,\&\, a_n.deployAd_n \\ & \longrightarrow & [\![Ad_1\{z/a\}]\!]_A \,\&\, \cdots \,\&\, [\![Ad_n\{z/a\}]\!]_A \end{array}$$

leading the normal form $\mathcal{C}_1' \overset{\text{proc}}{\sim} [\![\mathcal{C}_0']\!]_{\text{init}}$.

The converse direction says that any reduction in the translated configuration can be seen as a step in the simulated reduction of a Rule RED/ASP or RED/NO ASP of the original configuration. More precisely, we have to show that any reduction $\mathcal{C}_1 \longrightarrow \mathcal{C}_1'$ can be seen as a reduction between their normal forms. This expressed by the following diagram:

$$\begin{array}{ccccc} \mathcal{C}_0 & \overset{\mathcal{R}}{\text{---}} & \mathcal{C}_1 & \overset{*}{\underset{\mathbb{T}}{\longrightarrow}} \widetilde{\mathcal{C}_1} \overset{\text{proc}}{\sim} & [\![\mathcal{C}_0]\!]_{\text{init}} \\ \vdots & & \downarrow & & \vdots \\ \mathcal{C}_0'^{*} & \text{-} \overset{\mathcal{R}}{\text{-}} & \mathcal{C}_1' & \overset{*}{\underset{\mathbb{T}}{\longrightarrow}} \widetilde{\mathcal{C}_1'} \overset{\text{proc}}{\sim} & [\![\mathcal{C}_0']\!]_{\text{init}} \end{array}$$

If the reduction is $\mathcal{C}_1 \longrightarrow_{\mathbb{T}} \mathcal{C}_1'$, then $\widetilde{\mathcal{C}_1} = \widetilde{\mathcal{C}_1'}$ and $\mathcal{C}_0 = \mathcal{C}_0'$. If it introduces a message $resume_{M,"P"}(f,\vec{v})$, then $\widetilde{\mathcal{C}_1'} \overset{\text{proc}}{\sim} \widetilde{\mathcal{C}_1} \overset{\text{proc}}{\sim} [\![\mathcal{C}_0]\!]_{\text{init}}$. If it introduces a message $add\_advice(ad,a,pc)$, then $\mathcal{C}_0 \overset{Adv}{\longrightarrow} \mathcal{C}_0'$ and $\widetilde{\mathcal{C}_1'} \overset{\text{proc}}{\sim} [\![\mathcal{C}_0']\!]_{\text{init}}$. Otherwise, the reduction consumes a pattern $x.M_\sigma$ and produces a message of the form

$$w.weave(z, "P", f, \texttt{localhost}, \vec{v}).$$

Then, if some aspects can be deployed, $\mathcal{C}_0'$ is obtained by applying Rule RED/ASP to $x.M \triangleright P \Vdash^{\varphi} x.M_\sigma$, and if no aspect can be deployed, $\mathcal{C}_0'$ is obtained by applying Rule RED/NO ASP to $x.M \triangleright P \Vdash^{\varphi} x.M_\sigma$. The fact that the diagram above commutes is a direct consequence of the confluence of $\longrightarrow_{\mathbb{T}}$.

We conclude the proof of the theorem by noting that $[\![\mathcal{C}_0]\!]_{\text{init}}$ is a normal form for $\longrightarrow_{\mathbb{T}}$, so that $\mathcal{C}_0 \;\mathcal{R}\; [\![\mathcal{C}_0]\!]_{\text{init}}$. □

The crux of the proof lies in the confluence of $\longrightarrow_{\mathbb{T}}$ which means that once the message $weave(x,p,f,H,\vec{v})$ is send to the weaver, the translation introduces no further choice in the configuration. That is, every possible choice in $[\![\mathcal{C}]\!]$ corresponds directly to the choice of a reduction rule in $\mathcal{C}$.

Note that the bisimulation we have defined is not barbed-preserving nor context-closed. This is not surprising as a context would be able to distinguish between the original and translated configuration by using the flow information. But we are interesting in equivalent behaviour of two closed configurations, not of two terms that can appear in any context, so a simple bisimulation is sufficient in our case.

5. CONNECTIONS TO EXISTING DISTRIBUTED AOP

Local weavers. In our implementation of the weaving algorithm, we have made the choice of a central weaver. Even if this was enough for the correction of the algorithm, it might be inefficient from a practical point of view as every machine would have to connect to the same server at each reduction step. A more realistic algorithm would be to attach each local weaver and its corresponding aspects to the location where the intercepted method will be executed. This implementation will prevent network congestion at the weaver location.

Migration of aspects and advices. In the aspect join calculus, one can attach an aspect to an object. It suffices to host the aspect at a sub-location of the object, and thus the aspect will migrate with its object. The property that an aspect is attached to an object has been discussed in [14].

Note that it also possible to define more general migration of aspects and advices. For example, one can define an aspect with an advice that migrates at each deployment to an host that possesses the resource of interest.

Grouping host. In some distributed aspect oriented languages such as AWED, there is a notion of group of hosts that can be dynamically managed by adding and removing host. A group of hosts can then be used to define pointcuts. This mechanism can be translated in the aspect join calculus by a creation of a location for each group of hosts and a migration of the hosts to that location. Adding or removing an host will also be perform by migration. Then, a pointcut can be defined on the location of the group. Nevertheless, this point of view is not completely similar to the mechanism used in AWED as it presents the drawback to force a tree-like configuration of hosts.

Synchronous aspects in sequence. Our deployment of aspects is, as the join calculus, eminently asynchronous. Nevertheless, we code encode sequential execution by adding a

channel a.proceedAd and a definition

$$\mathsf{proceedAd}(\vec{v}) \triangleright \mathsf{proceed}(\vec{v})$$

for every advice Ad of every aspect a. A normal call to proceed in Ad is then replaced by a call to a.proceedAd. We will use those new labels to trigger the execution of aspects. Suppose that two advices $Pc_1\ \{Ad_1\}$ and $Pc_2\ \{Ad_2\}$ interrupt the same rule $c.M$. Then, in traditional synchronous setting, the user has to define the order of execution between both aspects, let say $Pc_1\ \{Ad_1\}$ before $Pc_2\ \{Ad_2\}$. In the join calculus, we will define two advices for translating $Pc_2\ \{Ad_2\}$: one that triggers $c.M$ with the optional condition that $Pc_2^{opt} \wedge \neg Pc_1^{opt}$ is satisfied, and one that triggers the call to proceed of $PC_1\ \{Ad_1\}$ with the optional argument that Pc_2^{opt} is satisfied.

$$Pc_2\ \{Ad_2\} \quad \Rightarrow \quad \begin{cases} \mathsf{rule}(c.M(\vec{v})) \wedge Pc_2^{opt} \wedge \neg Pc_1^{opt}\ \{Ad_2\} \\ \mathsf{rule}(a_1.\mathsf{proceedAd}(\vec{v}) \wedge Pc_2^{opt}\ \{Ad_2\} \end{cases}$$

After and Before advices. In an asynchronous setting, `After` and `Before` advices do not really make sense. Nevertheless, in an synchronous setting where every method has an entry point and a return value, those two kind of advices can be easily encoded. `Before` is encoded by an advice that intercepts the call of a method m and where exactly one proceed is performed at the end of the advice body

$$\mathsf{before}(c.m)\{P\} \equiv \mathsf{intercept\ rule}(c.m(\vec{v}))\ \{P; \mathsf{proceed}(\vec{v})\}$$

In the same way, `After` is encoded by an advice that intercepts the return label of the method and where exactly one proceed is performed at the end of the advice body.

Call and execute pointcuts. We can not interpret the difference between `call` and `execute` pointcuts in our setting. This is due to the absence of inheritance in our model. But as we have already said, inheritance can be added smoothly following the work of [5].

Distributed control flow. In the objective join calculus, there is no notion of method body and return value. The flow of execution just indicates that a method has been called, but says nothing about termination. Then, the pointcut $\mathsf{flow}(l)$ just says that l has been called at least once during the reduction, an information that appears quickly to be useless.

Nevertheless, in a synchronous setting, we can extract the traditional notion of control flow from this "flat" notion of flow already present in the calculus. Indeed, when every method has a call and return discipline, we can decide which method calls have not been completed yet. This corresponds to parsing the flow of execution and detecting the called methods that have not returned yet. More precisely, with the continuation passing encoding, a call $x.m(k, \vec{v})$ to method $x.m$ has not returned when the message $k.return(\vec{w})$ does not appear after in the flow. This enables to define the well-known AspectJ pointcut designator `Cflow`.

In the same way, we can construct a within pointcut designator.

Changing the route of messages. A common use of aspects in a distributed setting is the re-routing of messages. For example, one would like to intercept and re-route every a message sent to a machine that has crashed. This interception of message is not directly possible in the aspect join calculus. This is because explicit routing of messages is implicit and does not constitute an observable event of the language. However, we can recover routing information after the following encoding. Any emission of the the message $y.l(\vec{v})$ by an object x is replaced by the emission of the message $x.send\_l(y, \vec{v})$ and the reaction rule

$$send\_l(y, \vec{v}) \triangleright y.l(\vec{v}).$$

Then, an aspect can prevent routing of the message $y.l$ to the host were y is situated by intercepting the message $x.send\_l(y, \vec{v})$.

6. CONCLUSIONS

In this paper, we have developed the first formal theory of distributed aspects. Based on the distributed and objective join calculus, our calculus is presented with a type system and a (chemical) operational semantics. We have also defined a translation of the aspect join calculus into the core join calculus and shown the correctness of the translation with a proof of bisimilarity. In this way, we provide a well-defined version of a weaving algorithm that makes possible an implementation of the aspect join calculus directly in JoCaml. This paper has also shown that the main features of distributed AOP can be modeled by a few relatively simple constructs in a core calculus. Those key features are: distributing cut, distributed action, migration of aspects, asynchronous and synchronous aspects, re-routing of messages, distributed control flow.

7. ACKNOWLEDGMENTS

The author wants to thank Remi Douence, Hervé Graal and Mario Südholt for valuable discussions and comments.

8. REFERENCES

[1] A. Ahern and N. Yoshida. Formalising Java RMI with explicit code mobility. *Theoretical Computer Science*, 389(3):341–410, 2007.

[2] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I*, 3880:135–173, 2006.

[3] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385. ACM New York, NY, USA, 1996.

[4] C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. *Lecture Notes in Computer Science*, 2395:268–332, 2002.

[5] C. Fournet, C. Laneve, L. Maranget, and D. Remy. Inheritance in the join calculus. *Journal of Logic and Algebraic Programming*, 57(1):23–70, 2003.

[6] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[7] A. Jeffrey. A distributed object calculus. In *Proceedings of the 25th workshop on foundations of object-oriented languages*, 2000.

[8] L. Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvee. Explicitly distributed AOP using AWED. In *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62. ACM New York, NY, USA, 2006.

[9] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut: a language construct for distributed AOP. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 7–15. ACM New York, NY, USA, 2004.

[10] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC: an aspect-based distributed dynamic framework. *Software: Practice and Experience*, 34(12), 2004.

[11] M. Prasad and B. Chaudhary. A Type System for an Aspect Oriented Programming Language. In *International Conference on Distributed Computing and Internet Technology*, pages 266–272. Springer, 2004.

[12] J. Riely and M. Hennessy. A typed language for distributed mobile processes. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–390. ACM New York, NY, USA, 1998.

[13] A. Schmitt. Safe Dynamic Binding in the Join Calculus. In *Proc. IFIP TCS*, pages 563–575. Citeseer, 2001.

[14] É. Tanter, J. Fabry, R. Douence, J. Noyé, and M. Südholt. Expressive scoping of distributed aspects. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 27–38. ACM, 2009.

[15] E. Tanter and R. Toledo. A versatile kernel for distributed AOP. *Lecture Notes in Computer Science*, 4025:316–331, 2006.