Noname manuscript No.
(will be inserted by the editor)

Chemical Foundations of Distributed Aspects

Nicolas Tabareau · Éric Tanter

the date of receipt and acceptance should be inserted later

Abstract Distributed applications are challenging to program because they have to deal with a plethora of concerns, including synchronization, locality, replication, security and fault tolerance. Aspect-oriented programming (AOP) is a paradigm that promotes better modularity by providing means to encapsulate crosscutting concerns in entities called aspects. Over the last years, a number of distributed aspect-oriented programming languages and systems have been proposed, illustrating the benefits of AOP in a distributed setting.

Chemical calculi are particularly well-suited to formally specify the behavior of concurrent and distributed systems. The join calculus is a functional name-passing calculus, with both distributed and object-oriented extensions. It is used as the basis of concurrency and distribution features in several mainstream languages like C# (Polyphonic C#, now Cω), OCaml (JoCaml), and Scala Joins. Unsurprisingly, practical programming in the join calculus also suffers from modularity issues when dealing with crosscutting concerns.

We propose the Aspect Join Calculus, an aspect-oriented and distributed variant of the join calculus that addresses crosscutting and provides a formal foundation for distributed AOP. We develop a minimal aspect join calculus that allows aspects to advise reactions, and then extend it in several directions. We show how to deal with causal relations in pointcuts and how to support advanced customizable aspect weaving semantics. We also provide the foundation for a decentralized distributed aspect weaving architecture. Finally, we extend the calculus so that aspects can intervene upon migration of localities.

The semantics of the aspect join calculus is given by a chemical operational semantics. We give a translation of the aspect join calculus into the core join calculus, and prove this translation correct by a bisimilarity argument. This translation is used to implement Aspect JoCaml on top of JoCaml.[1]

N. Tabareau
Ascola, INRIA, France
E-mail: nicolas.tabareau@inria.fr

É. Tanter
PLEIAD Lab, Computer Science Dept (DCC),
University of Chile, Chile
E-mail: etanter@dcc.uchile.cl

1 Introduction

Distributed applications are complex to develop because of a plethora of issues related to synchronization, distribution, and mobility of code and data 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 [24,15]. Indeed, the pointcut/advice mechanism of AOP provides the facility to intercept the flow of control when a program reaches certain execution points (called *join points*) and perform new computation (called *advice*). The join points of interest are denoted by a predicate called a *pointcut*.

AOP is frequently used in distributed component infrastructures such as Enterprise Java Beans, appli-

[1] Implementation available online at:
`http://tabareau.fr/aspect_jocaml`

cation frameworks (such as Spring[2]) and application servers (such as JBoss[3]), Recently, there is a growing interest in the use of AOP for Cloud computing [32, 9], including practical infrastructures such as CloudStack[4]. In all these cases however, AOP systems do not support the remote definition or application of aspects. Rather, non-distributed aspects are used to manipulate distributed infrastructures [39].

To address these limitations, distributed AOP has been the focus of several practical developments: JAC [36], DJcutter [33], QuO's ASL [14], ReflexD [46], AWED [4, 5], Lasagne [47], as well as a higher-order procedural language with distribution and aspects [45]. These languages introduce new concepts for distributed AOP such as remote pointcut (advice triggered by remote join points), distributed advice (advice executed on a remote host), migration of aspects, asynchronous and synchronous aspects, distributed control flow, etc. Almost all these 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 the drawback that it makes it almost impossible to define the formal semantics of distributed aspects. While the formal foundations of aspects have been laid out in the sequential setting [49,12], to date, no theory of distributed aspects has been developed.[5]

This paper develops the formal foundations of distributed AOP using a chemical calculus, essentially a variant of the distributed join calculus [17]. The join calculus is a functional name-passing calculus founded on the chemical abstract machine and implemented in several mainstream languages like OCaml [19], C# [6] and Scala [21]. Chemical execution engines are also being developed for Cloud computing [37,34]. Due to its chemical nature, the join calculus is well-suited to describe parallel computation. The explicit treatment of localities and migration in the distributed join calculus make it possible to express distribution-related concerns directly.

In the join calculus, communication channels are created together with a set of reaction rules that specify, once and for all, how messages sent on these names are synchronized and processed. The crosscutting phenomena manifests in programs written in this style, just as they do in other languages. The reason is that reactions in the join calculus are scoped: it is not possible to define a reaction that consumes messages on external channels. Therefore, extending a cache process with replication implies modifying the cache definition itself. Similarly, establishing alternative migration policies based on the availability of locations requires intrusively modifying components.

The Aspect Join Calculus developed in this paper addresses crosscutting issues in the join calculus by introducing the possibility to define aspects that can react to chemical reactions. In doing so, it provides a formal foundation that can be used to understand and describe the semantics of existing and future distributed aspect languages. We also use it to describe interesting features that have not (yet) been implemented in practical distributed AOP systems.

Section 2 presents the distributed objective join calculus, which serves as the basis for the aspect join calculus. The core of the aspect join calculus is described in Section 3. In Section 5, we extend the expressive power of pointcuts with causality, based on a temporal logic. Section 6 describe an extension of the calculus with different weaving semantics per reaction. The issue of decentralized aspect weaving is addressed in Section 7 by developing the concept of weaving registries. Section 8 shows how to extend the core aspect join calculus to expose migration of localities as join points. Section 9 describes Aspect JoCaml, an implementation of the aspect join calculus on top of JoCaml. The weaving is based on a transformation from the aspect join calculus into the core join calculus; the correctness of this translation is given by a bisimilarity proof, in Appendix A. Finally, Section 10 discusses related work and Section 11 concludes.

2 The distributed objective join calculus

We start by presenting a distributed and object-oriented version of the join calculus.[6] This calculus, which we call the *distributed objective join calculus*, is an original, slightly adapted combination of an object-oriented version of the join calculus [18] and an explicit notion of location to account explicitly for distribution [16].

[2] `http://www.springsource.org`

[3] `http://www.jboss.org`

[4] `http://cloudstack.apache.org/`

[5] This article builds upon a previous conference publication [43]. Much of the text has been completely rewritten. The aspect join calculus has been simplified and clarified, in particular by removing the type system and the management of classes, because they are orthogonal to the extensions considered in this work. In addition, the technical treatment has been extended in many ways: advanced quantification (in particular temporal pointcuts of Section 5), per-reaction weaving (Section 6), decentralized weaving (Section 7), and migration join points (Section 8) are all new. Finally, an implementation based on JoCaml is presented (Section 9) and provided online.

[6] There is a good reason why we choose a variant of the join calculus with objects; we discuss it later in Section 3.3, once the basics of aspects in the calculus are established.

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 object *buffer* presented in [18]. 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

$$\mathsf{obj}\ r = reply(n) \triangleright out.print(n)$$

defines an object that reacts to messages on its own label *reply* by sending a message with label *print* and content n to an object named *out* that prints on the terminal. In the definition of an object, the '$\triangleright$' symbol defines a reaction rule that consumes the messages on its left hand side and produces the messages on its right hand side.

Note that labels may also be used to represent the internal state of an object. Consider for instance the definition of a one-place buffer object:

$$\begin{array}{ll}\mathsf{obj}\ b = & put(n)\ \&\ empty() \triangleright b.some(n)\\ \mathsf{or} & get(r)\ \&\ some(n) \triangleright r.reply(n)\ \&\ b.empty()\\ \mathsf{init}\ b.empty()\end{array}$$

A buffer can either be empty or contain one element. The buffer state is encoded as a message pending on *empty* or *some*, respectively. A buffer object is created empty, by sending a first message $b.empty$ in the init clause. 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 internal labels directly. This can be enforced by a privacy discipline, as described in [18].

2.2 Syntax

We use three disjoint countable sets of identifiers for object names $x, y, z \in \mathcal{O}$, labels $l \in \mathcal{L}$ and host names $H \in \mathcal{H}$. Tuples are written $(v_i)^{i\in I}$ or simply $\bar{v}$. We use v to refer indifferently to object or host names, *i.e.* $v \in \mathcal{O} \bigcup \mathcal{H}$. The grammar of the distributive objective join calculus is given in Figure 1; it has syntactic categories for processes P, definitions D, patterns M, and named definitions $\mathcal{D}$. The main construct is object definition $\mathsf{obj}\, x = D\, \mathsf{init}\, P\, \mathsf{in}\, Q$ that binds the name x to the definitions of D. The scope of x is every guarded process in D (here x means "self") and the processes P and Q. Objects are taken modulo renaming of bound names (or α-conversion). When P (*resp.* Q) is 0, init (*resp.* in) can be omitted.

$P ::=$		**Processes**
	0	null process
	$x.M$	message sending
	$\mathsf{obj}\, x = D\, \mathsf{init}\, P\, \mathsf{in}\, Q$	object definition
	$\mathsf{go}(H); P$	migration request
	$H[P]$	situated process
	$P \,\&\, P$	parallel composition
$D ::=$		**Definitions**
	$M \triangleright P$	reaction rule
	$D\, \mathsf{or}\, D$	disjunction
$M ::=$		**Patterns**
	$l(\bar{v})$	message
	$M \,\&\, M$	synchronization
$\mathcal{D} ::=$		**Named Definitions**
	$x.D$	object definition
	$H[\mathcal{D}: P]$	sub-location definition
	$\mathcal{D}\, \mathsf{or}\, \mathcal{D}$	disjunction
	$\top$	void definition

Fig. 1 Syntax of the distributed objective join calculus (a combination of simplified versions of the distributed join calculus [16] and the objective join calculus [18])

Definitions D are a disjunction of reaction rules. A reaction rule $M \triangleright P$ associates a pattern M with a guarded process P. Every message pattern $l(\bar{v})$ in M binds the object names and/or hosts $\bar{v}$ 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. Named definitions $\mathcal{D}$ are a disjunction of object definitions $x.D$ and sub-location definitions $H[\mathcal{D}: P]$, hosting the named definitions $\mathcal{D}$ and process P at host H. Note that each object is associated to exactly one named definition. $H[P]$ is the process that starts a fresh new location with process P. Note that $H[P]$ acts as a binder for creating a new host. A migration request is described by $\mathsf{go}(H'); P$. It is subjective in that it provokes the migration of the current host H to location $\psi H'$ with continuation process P. The definitions of free names (noted $\mathsf{fn}(.)$) for processes, definitions, patterns and named definitions are given in Figure 2.

2.3 Semantics

The operational semantics of the distributed objective join calculus is given as a *reflexive chemical abstract machine* [17]. A machine $\mathcal{D} \Vdash^{\varphi} \mathcal{P}$ consists in a set of named definitions $\mathcal{D}$ and of a multiset of processes $\mathcal{P}$ running in parallel at location $\varphi = H_1 \cdots H_n$. Each rewrite rule applies to a configuration $\mathcal{C}$, called a *chemical solution*, which is a set of machines running in par-

Structural rules

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

EMPTY
$\top \Vdash^{\varphi} \equiv \Vdash^{\varphi}$

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

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

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

SUB-LOC
$H[\mathcal{D}: P] \Vdash^{\varphi} \equiv \{\mathcal{D}\} \Vdash^{\varphi H} \{P\}$ (H frozen)

OBJ-DEF
$\Vdash^{\varphi} \mathsf{obj}\, x = D \,\mathsf{init}\, P \,\mathsf{in}\, Q \equiv x.D \Vdash^{\varphi} P, Q$ (x fresh)

LOC-DEF
$\Vdash^{\varphi} H[P] \equiv H[\top: P] \Vdash^{\varphi}$ (H fresh)

Reduction rules

RED
$x.[M \rhd P] \Vdash^{\varphi} x.M\sigma \longrightarrow x.[M \rhd P] \Vdash^{\varphi} P\sigma$

MESSAGE-COMM
$\Vdash^{\varphi} x.M \parallel x.D \Vdash^{\psi} \longrightarrow \Vdash^{\varphi} \parallel x.D \Vdash^{\psi} x.M$

MOVE
$H[\mathcal{D}: (P \,\&\, \mathsf{go}(H'); Q)] \Vdash^{\varphi} \parallel \Vdash^{\psi H'} \longrightarrow \Vdash^{\varphi} \parallel H[\mathcal{D}: (P \,\&\, Q)] \Vdash^{\psi H'}$

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
$$\frac{\Vdash^{\varphi} P \equiv x.D \Vdash^{\varphi} \mathcal{P}' \qquad x \notin \mathsf{fn}(\mathcal{D}) \cup \mathsf{fn}(\mathcal{P})}{\mathcal{D} \Vdash^{\varphi} P, \mathcal{P} \equiv \mathcal{D}, x.D \Vdash^{\varphi} \mathcal{P}', \mathcal{P}}$$

Fig. 3 Chemical semantics of the distributed objective join calculus (adapted from [16,18])

$\mathsf{fn}(0)$	=	$\varnothing$
$\mathsf{fn}(x.M)$	=	$\{x\} \cup \mathsf{fn}(M)$
$\mathsf{fn}(\mathsf{obj}\, x = D \,\mathsf{init}\, P \,\mathsf{in}\, Q)$	=	$(\mathsf{fn}(D) \cup \mathsf{fn}(P) \cup \mathsf{fn}(Q)) \setminus \{x\}$
$\mathsf{fn}(\mathsf{go}(H); P)$	=	$\{H\} \cup \mathsf{fn}(P)$
$\mathsf{fn}(H[P])$	=	$\mathsf{fn}(P) \setminus H$
$\mathsf{fn}(P \,\&\, Q)$	=	$\mathsf{fn}(P) \cup \mathsf{fn}(Q)$
$\mathsf{fn}(M \rhd P)$	=	$\mathsf{fn}(P) \setminus \mathsf{fn}(M)$
$\mathsf{fn}(D \mathsf{\ or\ } D')$	=	$\mathsf{fn}(D) \cup \mathsf{fn}(D')$
$\mathsf{fn}(l(\bar{v}))$	=	$\{v_i / i \in I\}$
$\mathsf{fn}(M \,\&\, M')$	=	$\mathsf{fn}(M) \cup \mathsf{fn}(M')$
$\mathsf{fn}(x.D)$	=	$\{x\} \cup \mathsf{fn}(D)$
$\mathsf{fn}(H[\mathcal{D}: P])$	=	$(\mathsf{fn}(\mathcal{D}) \cup \mathsf{fn}(P)) \setminus H$
$\mathsf{fn}(\mathcal{D} \mathsf{\ or\ } \mathcal{D}')$	=	$\mathsf{fn}(\mathcal{D}) \cup \mathsf{fn}(\mathcal{D}')$
$\mathsf{fn}(\top)$	=	$\varnothing$

Fig. 2 Definition of free names $\mathsf{fn}(\cdot)$

allel:

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

Intuitively, a root location H can be thought of as an IP address on a network and a machine at host/root location H can be thought of as a physical machine at this address. Differently, a machine at sub-location HH' can be thought of as a system process H' executing on a physical machine (whose location is H). This includes for example the treatment of several threads, or of multiple virtual machines executing on the same physical machine.

A chemical reduction is the composite of two kinds of rules: (i) structural rules $\equiv$ that deal with (reversible) syntactical rearrangements, (ii) reduction rules $\longrightarrow$ that deal with (irreversible) basic computational steps. The rules for the distributed objective join calculus are given in Figure 3.

Rules OR and EMPTY make composition of named definitions associative and commutative, with unit $\top$. Rules PAR and NIL do the same for parallel composition of processes. Rule JOIN gathers messages that are meant to be matched by a reaction rule. Rule OBJ-DEF 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)Note that P and Q are treated identically here, but we distinguish between them in the definition to make clear which process deals with initialization of the object and which deals with remaining computation. Rule RED states how a message $x.M'$ interacts with a reaction rule $x.[M \rhd P]$. The notation $x.[M \rhd P]$ means that the unique named definition $x.D$ in the solution contains reaction rule $M \rhd P$. The message $x.M'$ reacts when there exists a substitution σ with domain $\mathsf{fn}(M)$ such that $M\sigma = M'$. In that case, $x.M\sigma$ is consumed and replaced by a copy of the substituted guarded process $P\sigma$.

In chemical semantics, each rule is local in the sense that it mentions only messages involved in the reaction; but it can be applied to a wider chemical solution that contains those messages. This is formalized by two context rules CONTEXT and CONTEXT-OBJ. In Rule CONTEXT, the symbol $\longrightarrow / \equiv$ stands for either $\longrightarrow$ or $\equiv$ (the same in premise and conclusion). In Rule CONTEXT-OBJ, the side condition $x \notin \mathsf{fn}(D) \cup \mathsf{fn}(P)$ prevents name capture when introducing new objects.

Distribution. Rule MESSAGE-COMM 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 holds 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 RED step. Note that in contrast to some models of distributed systems [38], the explicit routing of messages is not described by the calculus.

Rule LOC-DEF describes the introduction of a sub-location (up-to α-conversion, we can consider that any host appears only once in a configuration). Rule SUB-LOC introduces a new machine at sub-location φH of φ with $\mathcal{D}$ as initial definitions and P as initial process. When read from right-to-left, the rule can be seen as a serialization process, and conversely as a deserialization process. The side condition "H frozen" means that there is no other machine of the form $\Vdash^{\varphi H\psi}$ in the configuration (*i.e.* all sub-locations of H have already been "serialized"). The notation $\{\mathcal{D}\}$ and $\{P\}$ states that there are no extra definitions or processes at location φH.

Rule MOVE gives the semantics of migration. A sub-location φH of φ is about to move to a sub-location $\psi H'$ of ψ. On the right hand side, the machine $\Vdash^{\varphi}$ is fully discharged of the location H. Note that P can be executed at any time, whereas Q can only be executed after the migration. Rule MOVE says that migration on the network is based on sub-locations but not objects nor processes. When a migration order is executed, the continuation process moves with all the definitions and processes present at the same sub-location. Nevertheless, we can encode object (or 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.

Names and configuration binding. In the distributed join calculus, every name is defined in at most one local solution; rule MESSAGE-COMM hence applies at most once for every message, delivering the message to a unique location [16]. Similarly, we also assume that the rightmost host H_n defines the location φ uniquely.

In the semantics, the rule OBJ-DEF (*resp.* LOC-DEF) introduces a fresh variable x (*resp.* H) that is free in the definitions and processes of the whole configuration. But the fact that x (*resp.* H) appears on the left hand side of the machine definition means that the free variable is bound in the configuration. More precisely, for a configuration $\mathcal{C} = (\mathcal{D}_1 \Vdash^{\varphi_i} \mathcal{P}_i)_i$, we say that x is bound in $\mathcal{C}$, noted $\mathcal{C} \vdash x$, when there exists i such that $x.D$ appears in $\mathcal{D}_i$. Similarly, we say that H is bound in $\mathcal{C}$, noted $\mathcal{C} \vdash H$, when there exists i such that $H[\mathcal{D}{:}\mathcal{P}]$ appears in $\mathcal{D}_i$. This notion of configuration binding will be used in the definition of the semantics of pointcuts in Section 3.

2.4 A companion example

In the rest of the paper, we will use a cache replication example. To implement the running example, we assume a dictionary library *dict* with three labels:

- $create(x)$ returns an empty dictionary on $x.getDict$;
- $update(d, k, v, x)$ updates the dictionary d with value v on key k, returning the dictionary on $x.getDict$;
- $lookup(d, k, r)$ returns the value associated to k in d on $r.reply$

We also assume the existence of strings, which will be used for keys of the dictionary, written "`name`".

The cache we consider is similar to the buffer described in Section 2.1 but with a permanent state containing a dictionary and a *getDict* label to receive the (possibly updated) dictionary from the *dict* library:

$$\begin{array}{l}\mathsf{obj}\ c = \ put(k,v)\ \&\ state(d)\ \triangleright\ dict.update(d,k,v,c)\\ \qquad\ \mathsf{or}\ \ get(k,r)\ \&\ state(d)\ \triangleright\ dict.lookup(d,k,r)\ \&\ c.state(d)\\ \qquad\ \mathsf{or}\ \ getDict(d)\ \triangleright\ c.state(d)\\ \mathsf{init}\ dict.create(c)\\ \mathsf{in}\ \ \ldots\end{array}$$

For the moment, we just consider a single cache and a configuration containing a single machine as follows:

$$\begin{array}{l}c.[put(k,v)\ \&\ state(d)\ \triangleright\ dict.update(d,k,v,c),\\ \quad get(k,r)\ \&\ state(d)\ \triangleright\ dict.lookup(d,k,r)\ \&\ c.state(d),\\ \quad getDict(d)\ \triangleright\ c.state(d)],\\ r.[reply(n)\ \triangleright\ out.print(n)]\\ \Vdash^{H}\ c.state(d_0)\ \&\ c.get(\text{“}\texttt{foo}\text{”},r)\ \&\ c.put(\text{“}\texttt{bar}\text{”},5)\end{array}$$

At this point, two reactions can be performed, involving $c.state(d_0)$ and either $c.get(\text{“}\texttt{foo}\text{”},r)$ or $c.put(\text{“}\texttt{bar}\text{”},5)$. Suppose that *put* is (non-deterministically) chosen. The configuration amounts to:

$$Rules \Vdash^{H} dict.update(d_0, \text{“}\texttt{foo}\text{”}, 5, c)\ \&\ c.get(\text{“}\texttt{foo}\text{”}, r)$$

where *Rules* represents the named definitions introduced so far. $c.get(\text{“}\texttt{foo}\text{”},r)$ can no longer react, because there are no $c.state$ messages in the solution anymore. *dict* passes the updated dictionary d_1, which is passed in the message $c.state$ using reaction on label $c.getDict$.

$$Rules \Vdash^{H} c.state(d_1)\ \&\ c.get(\text{“}\texttt{foo}\text{”}, r)$$

Now, $c.get(\text{“}\texttt{foo}\text{”},r)$ can react with the new message $c.state(d_1)$, yielding:

$$Rules \Vdash^{H} c.state(d_1)\ \&\ r.reply(5)$$

Finally, 5 is printed out (consuming the $r.reply$ message) resulting in the terminal configuration:

$$Rules \Vdash^{H} c.state(d_1)$$

$D ::=$	$\ldots$	**Definitions**
	$\langle Pc, Ad \rangle$	pointcut/advice pair
$Pc ::=$		**Pointcuts**
	$\mathsf{contains}(x.M)$	sub-pattern M in x
	$\mathsf{host}(H)$	in a sub-location of φH
	$\neg Pc$	negation
	$Pc \wedge Pc$	conjunction
	$Pc \vee Pc$	disjunction
$Ad ::=$		**Advices**
	P	any process
	$\mathsf{proceed}\{(l_i : \bar{v_i} \mapsto \bar{v_i}')_i\}$	proceed

Fig. 4 Syntax of aspects in the aspect join calculus

$\mathsf{fn}(\mathsf{contains}(x.M))$	$=$	$\mathsf{fn}(x.M)$
$\mathsf{fn}(\mathsf{host}(H))$	$=$	H
$\mathsf{fn}(\neg Pc)$	$=$	$\mathsf{fn}(Pc)$
$\mathsf{fn}(Pc \wedge Pc')$	$=$	$\mathsf{fn}(Pc) \cup \mathsf{fn}(Pc')$
$\mathsf{fn}(Pc \vee Pc')$	$=$	$\mathsf{fn}(Pc) \cup \mathsf{fn}(Pc')$
$\mathsf{fn}(\mathsf{proceed}\{(l_i : \bar{v_i} \mapsto \bar{v_i}')_i\})$	$=$	$\{v_{ij}\} \cup \{v'_{ij}\}$
$\mathsf{fn}(\langle Pc, Ad \rangle)$	$=$	$\mathsf{fn}(Ad) \setminus \mathsf{fn}(Pc)$

Fig. 5 Definition of free names for aspects

2.5 Bootstrapping distributed communication

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, using a name server. The name server NS offers a service to associate a name to a constant string ($NS.register(\text{“}\texttt{my\_x}\text{”}, x)$), and to look up a name based on string ($NS.lookup(\text{“}\texttt{my\_x}\text{”}, r)$, where the value is sent on $r.reply$).

3 The aspect join calculus

We now describe the *aspect join calculus*, an extension of the distributed objective join calculus with aspects. Support for crosscutting in a programming language is characterized by its join point model [31]. A join point model includes the description of the points at which aspects can potentially intervene, called *join points*, the means of specifying the join points of interest, here called *pointcuts*, and the means of effecting at join points, called *advices*. We first describe each of these elements in turn, from a syntactic and informal point of view, before giving the formal semantics of aspect weaving in the aspect join calculus. The syntax of aspects is presented in Figure 4.

3.1 Defining the join point model

Join points. Dynamic join points reflect the steps in the execution of a program. For instance, in AspectJ [25] join points are method invocations, field accesses, etc. In functional aspect-oriented programming languages, join points are typically function applications [13,48].

The central computational step of any chemical language is the application of a reaction rule, here specified by Rule Red. Therefore, a *reaction join point* in the aspect join calculus is a pair $(\varphi, x.M\sigma)$, where φ is the location at which the reduction occurs, and $x.M\sigma$ is the instantiated synchronization pattern of the reduction.

In Section 8 we will extend the join point model with join points that denote the migration of locations, as specified by Rule Move.

To make the following definition already compatible with an extension of the join point model, we introduce the general notion of *situated join point* (φ, jp). In this section, jp will always be an instantiated synchronization pattern.

Pointcuts. The aspect join calculus includes two basic pointcut designators, *i.e.* functions that produce pointcuts: contains for reaction rules selection, and host for host selection. The pointcut $\mathsf{contains}(x.M)$ selects any reaction rule that contains the pattern $x.M$ as left hand part. The pointcut $\mathsf{host}(H)$ matches join points that occur on a sub-location of H. Finally, a pointcut can be constructed by negations, conjunctions and disjunctions of other pointcuts.

The free variables of a pointcut (as defined in Figure 5) are bound to the values of the matched join points. In this way a pointcut acts as a binder on the free variables occurring in the corresponding advice. Consider for instance the pointcut $\mathsf{contains}(x.M)$. If x is free, the pointcut will match any reaction whose pattern includes M, irrespective of the involved object, and that object will be bound to the identifier x in the advice body. If x is not a free name, the pointcut will match any reaction *on the object denoted by* x, whose pattern includes M. Note that similarly to synchronization patterns in the join calculus, we require the variables occurring in a pointcut to be linear. This ensures that the union $\sigma \cup \tau$ of two substitutions is always well defined.

In the following, when the variable to be matched is not interesting (in the sense that it is not used in the advice), we use the $*$ notation. For instance, the pointcut $\mathsf{contains}(*.put(k, v))$ matches all reactions containing $put(k, v)$ on any object, without binding the name of the object.

Advices. An advice body Ad is a process to be executed when the associated pointcut matches a join point. This process may contain the special keyword proceed. During the reduction, proceed is substituted by the resulting process P of the matched reaction. To allow to change the values bound to variables in the reaction pattern and used in the resulting process P, the term proceed may be extended with a substitution update by $\mathsf{proceed}\{(l_i{:}\bar{v_i} \mapsto \bar{v_i}')_i\}$. This construction means that, in the substitution σ given by the instantiated reaction pattern, the variables $\bar{v_i}$ associated to label l_i are rebound to $\bar{v_i}'$. Note that this allows to change the values only for a specific message l_i within the reaction, leaving the others implicitly unchanged. Free names of an advice are defined in Figure 5.

$$
\begin{array}{lcl}
(\varphi, jp) \Vdash_{\mathcal{C}} \mathsf{contains}(x.M) & \equiv & \begin{cases} \tau \text{ when } jp = x\tau.M'\sigma \\ \qquad \text{and } M\tau \subseteq M'\sigma \\ \bot \text{ otherwise} \end{cases} \\
(\varphi, jp) \Vdash_{\mathcal{C}} \mathsf{host}(H) & \equiv & \begin{cases} \{\ \} \text{ when } \varphi = \psi.H.\psi' \text{and } H \vdash \mathcal{C} \\ \tau \text{ when } \varphi = \psi.H' \text{ and } H\tau = H' \\ \qquad \text{and } H \nvdash \mathcal{C} \\ \bot \text{ otherwise} \end{cases} \\
(\varphi, jp) \Vdash_{\mathcal{C}} Pc \wedge Pc' & \equiv & \begin{cases} (\varphi, jp) \Vdash_{\mathcal{C}} Pc \cup (\varphi, jp) \Vdash_{\mathcal{C}} Pc' \\ \bot \text{ otherwise} \end{cases} \\
(\varphi, jp) \Vdash_{\mathcal{C}} Pc \vee Pc & \equiv & \begin{cases} (\varphi, jp) \Vdash_{\mathcal{C}} Pc \cup (\varphi, jp) \Vdash_{\mathcal{C}} Pc' \\ (\varphi, jp) \Vdash_{\mathcal{C}} Pc \text{ if } (\varphi, jp) \Vdash_{\mathcal{C}} Pc' = \bot \\ (\varphi, jp) \Vdash_{\mathcal{C}} Pc' \text{ if } (\varphi, jp) \Vdash_{\mathcal{C}} Pc = \bot \\ \bot \text{ otherwise} \end{cases} \\
(\varphi, jp) \Vdash_{\mathcal{C}} \neg Pc & \equiv & \begin{cases} \{\ \} \text{ when } (\varphi, jp) \Vdash_{\mathcal{C}} Pc = \bot \\ \bot \text{ otherwise} \end{cases}
\end{array}
$$

Fig. 6 Semantics of pointcuts

Aspects. To introduce aspects in the calculus, we extend the syntax of definitions D with pointcut/advice pairs (Figure 4). This means that an object can have both reaction rules and possibly many pointcut/advice pairs. This modeling follows symmetric approaches to pointcut and advice, like CaesarJ [2] and EScala [20], where any object has the potential to behave as an aspect. Free names of an aspect are defined in Figure 5.

The following example defines an object *replicate* that, when sent a *deploy* message with a given cache replicate object c and a host H', defines a fresh sublocation φH, migrates it to host H', and creates a new replication aspect:

$$
\begin{array}{l}
\Vdash^{\varphi} \mathsf{obj}\ replicate = \\
\quad deploy(c, H') \ \triangleright\ H[\mathsf{go}(H'); \mathsf{obj}\ rep = \\
\qquad\qquad\qquad \langle \mathsf{contains}(*.put(k, v)) \wedge \neg\, \mathsf{host}(H'), \\
\qquad\qquad\qquad\ c.put(k, v)\ \&\ \mathsf{proceed}\{\}\rangle] \\
\quad \mathsf{in}\ NS.register(\text{“}\texttt{replicate}\text{”}, replicate)
\end{array}
$$

The advice body replicates on c every *put* message received by a cache object and makes an explicit use of the keyword proceed in order to make sure that the intercepted reaction does occur. The condition $\neg\, \mathsf{host}(H')$ in the pointcut is used to avoid replication to apply to reactions that happen on a sub-location of the location where the aspect is deployed. Indeed, the aspect must not replicate local modifications of the cache.

3.2 Semantics

Semantics of pointcuts. The matching relation, noted $(\varphi, jp) \Vdash_{\mathcal{C}} Pc$, returns either a substitution τ from free names of Pc not bound in $\mathcal{C}$ to names of φ and jp, or a special value $\bot$ meaning that the pointcut does not match. We note $\{\ \}$ the empty substitution. When defined, we note $\cup$ the join operation on substitutions. From the condition that variables occur linearly in pointcuts, we have the guarantee that the join operation is always well-defined. $(\varphi, jp) \Vdash_{\mathcal{C}} Pc$ is defined by induction on the structure of the pointcut in Figure 6, where the inclusion on patterns $M' \subseteq M$ is defined as the inclusion of the induced set of messages.

For instance, suppose that the cache replication aspect defined above has been deployed and that the emitted join point is:

$$x.(put(k, v)\ \&\ state(d))_{\sigma} \quad \text{with } \sigma = \{k \mapsto \text{“}\texttt{bar}\text{”}, v \mapsto 5, d \mapsto \mathsf{d}\}$$

Then, the pointcut of the aspect matches, with partial bijection $\tau = \{k' \mapsto \text{“}\texttt{bar}\text{”}, v' \mapsto 5\}$ (where k' and v' are the free variables occurring in the pointcut). Note that the free variable d is not mapped by τ because it is not captured by the pointcut.

The semantics of the $\mathsf{host}(H)$ pointcut is slightly more complicated: when H is free in the configuration, the pointcut binds H to the inner-most locality, and when H is bound in the configuration, the pointcut matches for any sub-location of H.

Semantics of aspects. Figure 7 presents the semantics of aspects. All rules of Figure 3 are preserved, except for Rule Red, which is split in two rules.

Rule Deploy corresponds to the asynchronous deployment of a pointcut/advice pair $\langle Pc, Adv\rangle$ by marking the pair as deployed $\langle Pc, Adv\rangle_a$. Note that deployed pairs are not directly user-definable.

Rule Red/NoAsp is a direct reminiscence of Rule Red in case where no activated pointcut matches.

Rule Red/Asp defines the modification of Rule Red in presence of aspects. If there is an aspect a with a deployed pointcut/advice pair $\langle Pc, Ad\rangle$ such that Pc matches the join point with substitution τ, the advice Ad is executed with the process P substituting

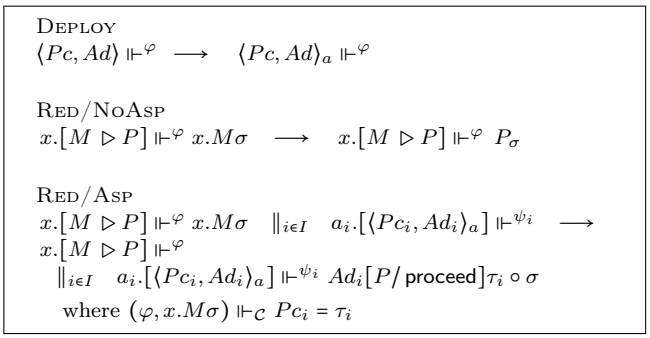

Fig. 7 Semantics of aspect weaving

the keyword proceed and where the variables bound by the pointcut are substituted according to τ.[7] The side condition of Rule RED/ASP is that all Pc_is are the deployed pointcuts that match the current join point $(\varphi, x.M\sigma)$. In particular, when two pointcut/advice pairs of the same object definition match, we can have $a_i = a_j$ and $\psi_i = \psi_j$.

Coming back to the example above, the synchronization pattern reacts to become:

$$\begin{array}{ll} & x.put(\text{“bar”}, 5)\ \&\ state(\mathsf{d}) \\ \longrightarrow & c.put(\text{“bar”}, 5)\ \&\ dict.update(\mathsf{d}, \text{“bar”}, 5, x) \end{array}$$

The original operation on *dict* to update d is performed, in addition to the replication on *c*. Note that all advices associated to a pointcut that matches are executed in parallel. We will discuss alternative weaving semantics in Section 6.

Finally, note that the semantics of aspect weaving relies on the currently-deployed aspects. As we have seen, deployment is asynchronous, which means that to be sure that an aspect is in operation at a given point a time, explicit synchronization has to be setup. This design is in line with the asynchronous chemical semantics of the join calculus. For instance, the same non-determinism occurs in the definition of an object, in which the initialization process is not guaranteed to be completed before the object process starts executing.

3.3 Why objects?

When designing the aspect join calculus, we considered defining it on top of the standard join calculus with explicit distribution, but without objects. However, it turns out that doing so would make the definition of aspects really awkward and hardly useful. Consider the standard join calculus definition of a buffer producer (adapted from [17]):

$$\begin{array}{l} \mathsf{def}\ make\_buffer(k) \rhd \\ \quad \mathsf{def}\ \ put(n) \mid empty() \rhd some(n) \\ \quad\ \wedge\ \ get(r) \mid some(n) \rhd r(n) \mid empty() \\ \mathsf{in}\ empty() \mid k(get, put) \end{array}$$

make_buffer takes as argument a response channel *k* on which the two newly-created channels *get* and *put* are passed (hence representing the new buffer). Crucially, the channel names *get* and *put* are local and not meaningful *per se*; when the definitions are processed, they are actually renamed to fresh names (rule STR-DEF in [17]). Therefore, there is no way for an aspect to refer to “a reaction that includes a message on the *get* channel”. Doing so would require modifying *make_buffer* to explicitly pass the newly-created channels also to the aspect, each time it is executed. An aspect would then have to match on all reactions and check if the involved channels include one of the ones it has been sent. In addition, the explicit modification of *make_buffer* defeats the main purpose of aspects, which is separation of concerns. A *make_buffer* that explicitly communicates its created channels to a replication aspect is not a general-purpose entity that can be reused in different contexts (*i.e.* without replication).

The objective join calculus, on the other hand, includes both object names and *labels*. Conversely to object names, labels have no local scope and are not subject to renaming [18]. They constitute a “shared knowledge base” in the system, which aspects can exploit to make useful quantification. This is similar to how method names are used in the pointcuts of object-based aspect-oriented languages.

The argumentation above also explains why we have chosen not to include classes as in [18] in our presentation of the aspect join calculus. Classes are interesting to support extensible definitions, but do not bring anything new with respect to naming and quantification (unless they are globally named, as in Java).

4 From the aspect join calculus to the join calculus

In this section, we present a translation of the aspect join calculus into the core join calculus. This allows us to specify an implementation of the weaving algorithm and to provide a bisimilarity proof that this translation has the same behavior as the original configuration with aspects. This translation is used in Section 9 to implement Aspect JoCaml on top of JoCaml [19], an implementation of the join calculus in OCaml.

[7] We use a structural rule that transfers the substitution update on processes to the usual notion of substitution update on substitutions: $(P\{l{:}\,\bar{v} \mapsto \bar{v}'\})\sigma \equiv P(\sigma\{\bar{v}_l \mapsto \bar{v}_l{}'\})$ where $\bar{v}_l$ correspond to the variables associated to the label l in the substitution σ.

Processes

$$
\begin{array}{rcl}
[\![0]\!] & \equiv & 0 \\
[\![x.M]\!] & \equiv & x.M \\
[\![\mathsf{obj}\, x = D \,\mathsf{init}\, P \,\mathsf{in}\, Q]\!] & \equiv & \mathsf{obj}\, x = [\![D_r]\!]_x \,\mathsf{init}\, [\![D_a]\!] \,\&\, [\![P]\!] \,\mathsf{in} [\![Q]\!] \\
[\![\mathsf{go}(H); P]\!] & \equiv & \mathsf{go}(H); [\![P]\!] \\
[\![H[P]]\!] & \equiv & H[[\![P]\!]] \\
[\![P \,\&\, P']\!] & \equiv & [\![P]\!] \,\&\, [\![P']\!]
\end{array}
$$

Definitions

$$
\begin{array}{rcl}
[\![M \rhd P]\!]_x & \equiv & M \rhd \mathsf{obj}\, ret_M = proceed(jp) \rhd \\
 & & \quad \mathsf{let}\ \bar{v} = \mathsf{fromJp}(M, jp)\ \mathsf{in}\ [\![P]\!] \\
 & & \mathsf{in}\ W.weave(ret_M, (\mathtt{lhost}, x.M)) \\
[\![D \,\mathsf{or}\, D']\!]_x & \equiv & [\![D]\!]_x \,\mathsf{or} [\![D']\!]_x
\end{array}
$$

Aspects

$$
\begin{array}{rcl}
[\![\langle Pc, Ad \rangle]\!]_x & \equiv & \mathsf{obj}\, adv = advice(k, jp) \rhd \\
 & & \quad \mathsf{let}\ \bar{v} = \mathsf{fromJp}(M, jp)\ \mathsf{in}\ [\![Ad]\!] \\
 & & \mathsf{init}\ W.deploy([\![Pc]\!], adv)
\end{array}
$$

Pointcuts

$$
\begin{array}{rcl}
[\![\mathsf{contains}(x.M)]\!] & \equiv & \lambda(\varphi, x'.M').\ x = x' \wedge M \subseteq M' \\
[\![\mathsf{host}(H)]\!] & \equiv & \lambda(\varphi, x.M').\ occurs(H, \varphi) \\
[\![\neg Pc]\!] & \equiv & \lambda jp.\ \neg [\![Pc]\!] jp \\
[\![Pc \wedge Pc']\!] & \equiv & \lambda jp.\ [\![Pc]\!] jp \wedge [\![Pc']\!] jp \\
[\![Pc \vee Pc']\!] & \equiv & \lambda jp.\ [\![Pc]\!] jp \vee [\![Pc']\!] jp
\end{array}
$$

Advices

$$
\begin{array}{rcl}
[\![\mathsf{proceed}\{\sigma\}]\!] & \equiv & k.\, \mathsf{proceed}\{\bar{v}'\} \\
 & & \quad where\ \sigma = (l_i{:}\bar{v_i} \mapsto \bar{v_i}')_i
\end{array}
$$

Fig. 8 Translating the aspect join calculus to the join calculus

Given an aspect join calculus configuration:

$$\mathcal{D}_1 \Vdash^{\varphi_1} \mathcal{P}_1 \quad \| \quad \cdots \quad \| \quad \mathcal{D}_n \Vdash^{\varphi_n} \mathcal{P}_n$$

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

$$[\![\mathcal{D}_1]\!] \Vdash^{\varphi_1} [\![\mathcal{P}_1]\!] \quad \| \cdots \| \quad [\![\mathcal{D}_n]\!] \Vdash^{\varphi_n} [\![\mathcal{P}_n]\!] \quad \| \quad \Vdash^{H_W} W.$$

The idea is to introduce an explicit join point in every reaction rule. This join point triggers a protocol with the weaver to decide whether or not advice intercepts the reaction rule and must be executed.

Note that in order to make the definition of processes more readable, we present some part of the translation in a functional programming style that can either be encoded in the join calculus, or can already be present in the language (*e.g.* in JoCaml). In particular, we will use a notion of list, equality testing, and a particular variable `lhost` that represents the current location on which a process is executing.

4.1 Translation

The general idea of the translation is that an aspect is seen as a standard object that receives messages from the weaver to execute a particular method that represents its advice. This is the usual way to compile aspects to a target object-oriented language without aspects [22]. The main difficulty is to encode the join point model and aspect weaving in the join calculus. For simplicity, we omit context exposure of pointcuts, *i.e.* we assume that the substitution defined by a match is always empty.

The translation is given in Figure 8.

Processes. The rules for processes recursively propagate the translation in sub-processes and definitions. The translation of objects requires to distinguish between reaction rules (D_r) and pointcut/advice pairs (D_a) in the original definition D, because each pointcut/advice is translated as a normal object.

Definitions. A reaction rule $M \rhd P$ is replaced by a call to the weaver, passing a locally-created object ret_M that defines a label *proceed* to perform the original computation P. The management of join points requires a function fromJp that converts a join point $(\varphi, x.M\sigma)$ into the tuple of arguments $\bar{v}$ present in the pattern M. These variables are rebound to potentially new values as specified in the join point passed as argument to *proceed*, taking into account modifications made by aspects.

Aspects. A pointcut/advice pair $\langle Pc, Ad \rangle$ is translated as an object that holds the advice and is registered with the weaver. The initialization sends the pointcut/advice pair to the weaver W by using the dedicated label *deploy*. Note that the pointcut is sent to the weaver but is not checked explicitly in the aspect. Indeed, it is the responsibility of the weaver to decide whether the advice must be executed or not. This is because the weaver must have the global knowledge of which pointcuts match, to perform Rule Red/NoAsp.

When going the left-hand side of a configuration, the translation of a pointcut/advice pair is obtained by applying the structural rule Obj-Def to the object translation. The translation of a deployed pointcut/advice pair is obtained by removing the message on *W.deploy*.

Pointcuts are recursively transformed into functions that operate on join points. The notation $M \subseteq M'$ means that labels occurring in M also occur in M', and $occurs(H, \varphi)$ means that H occurs in φ.

Finally, the translation of proceed obtains the tuple of arguments $\bar{v}'$ after application of the substitution update $(l_i{:}\bar{v_i} \mapsto \bar{v_i}')_i$ to $\bar{v}$.

Weaver. In this section, we rely on a central weaver that handles all reactions. A more realistic decentralized version will be described in Section 7.

A weaver is an object with three labels: $aspL$, which maintains the list of all currently-deployed aspects; *deploy*, which is used to deploy new aspects; and $weave$, which is used to start the weaving of a join point.

$$
\begin{array}{l}
\mathsf{obj}\, W = deploy(pc, ad)\ \&\ aspL(asps) \triangleright aspL((pc, adv){:}asps) \\
\qquad\quad weave(k, jp)\ \ \&\ aspL(asps) \triangleright aspL(asps)\ \& \\
\qquad\quad \mathsf{let}\ ads = \mathsf{filter}\ (\lambda(pc, \_).pc\ jp)\ asps\ \mathsf{in} \\
\qquad\quad \mathsf{if}\ \ ads = [\,]\ \mathsf{then}\ k.proceed(jp) \\
\qquad\qquad\qquad\quad \mathsf{else}\ \mathsf{iter}\ (\lambda(\_, ad).\ ad.advice(k, jp)))\ ads \\
\mathsf{init}\ aspL(asps_a)
\end{array}
$$

Initially, the list of deployed aspects is set to $asps_a$, the list of already-deployed aspects of the configuration. When the weaver receives a $deploy(pc, ad)$, it updates its list of aspects $aspL$ accordingly. When the message $weave(k, jp)$ is captured, the weaver tests the emptiness of the list ads of advices whose pointcut matches jp (defined using the usual filter functions on lists). If no aspects applies, the weaver executes the original process by sending the message $k.proceed(jp)$ to resume the original computation; this corresponds to Rule Red/NoAsp. Otherwise, the weaver executes all advices present in ads asynchronously using the iter function list; this corresponds to Rule Red/Asp.

4.2 Bisimulation between an aspect join calculus process and its translation

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 correctness 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 configurations can perform 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 (in 0, 1 or more steps) to $\mathcal{C}'_1$. We illustrate this with the following diagram:

$$
\begin{array}{ccc}
\mathcal{C}_0 & \xrightarrow{\ \mathcal{R}\ } & \mathcal{C}_1 \\
\big\downarrow & & \big\downarrow{\scriptstyle *} \\
\mathcal{C}'_0 & \xrightarrow{\ \mathcal{R}\ } & \mathcal{C}'_1
\end{array}
$$

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

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

1. During the evolution of $[\![\mathcal{C}]\!]$, auxiliary messages that have no correspondents in $\mathcal{C}$ are sent for communication between processes, weaver and aspects.
2. In the execution of $\mathcal{C}$, proceed is substituted by the process P to be executed, whereas in $[\![\mathcal{C}]\!]$, P is executed through a communication with the object where the reaction has been intercepted.

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 come 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 where the pattern contains either of the dedicated labels: $weave$, $proceed$, $advice$ (the label *deploy* is treated differently as it corresponds to the application of Rule Deploy). In $\mathbb{T}$, those labels only interact alone, or one-by-one with the constant label $aspL$. So the order in which reaction rules are selected has no influence on the synchronized pattern; in other words, 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 $\mathcal{C} \overset{\mathtt{proc}}{\sim} \mathcal{C}'$ when $\mathcal{C}'$ is equal to $\mathcal{C}$ where every message $proceed(jp)$ is substituted by the process $P(\bar{v})$.

Theorem 1 *The relation* $\mathcal{R} = \{(\mathcal{C}_0, \mathcal{C}_1) \mid [\![\mathcal{C}_0]\!] \overset{\mathtt{proc}}{\sim} \widetilde{\mathcal{C}_1}\}$ *is a bisimulation. In particular, any configuration is bisimilar to its translation.*

Proof See Appendix A. The crux of the proof lies in the confluence of $\longrightarrow_{\mathbb{T}}$ which means that once the message $weave(k, jp)$ is sent 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 $\mathcal{C}$.

5 Advanced quantification: causality

We now start exploring different extensions and refinements of the aspect join calculus. In this section, we focus on quantification.

If a cache replication aspect is deployed on each host of interest, then aspects will indefinitely replicate the cache replicated by aspects on other hosts. In AWED [4], this livelock is prevented by excluding join points produced within the body of the aspect, using the `within` pointcut designator. Similarly, it is common in aspect languages to use control-flow related pointcuts, in order to be able to discriminate join points caused by others. Most distributed aspect languages and systems support distributed control flow (although in a synchronous setting). This indicates that a notion of causality is required in order to express this kind of pointcuts. We show how to support causality in the aspect join calculus by extending the information available to pointcuts to include the whole *causality flow* of join points.

Before going into details, let us illustrate with the cache replication example. To be able to identify aspect-specific activity, we declare an aspect object, with a specific label *rput* whose goal is to make the activity of the aspect visible. Then the new definition of the cache replication aspect below also excludes the activity *caused by* a cache replication aspect using the pointcut $\neg\,\mathsf{producedBy}(*.rput)$.

$$
\begin{array}{l}
\Vdash^{\varphi} \mathsf{obj}\ replicate = \\
\quad deploy(c, H') \ \triangleright\ H[\mathsf{go}(H'); \\
\qquad\qquad\qquad\qquad \mathsf{obj}\ rep = \\
\qquad\qquad\qquad\qquad\quad rput(k,v) \ \triangleright\ c.put(k,v) \\
\qquad\qquad\qquad\qquad\ \mathsf{or}\ \langle \mathsf{contains}(*.put(k,v)) \wedge \neg\,\mathsf{host}(H') \wedge \\
\qquad\qquad\qquad\qquad\quad \neg\,\mathsf{producedBy}(\mathsf{contains}(*.rput(*,*))), \\
\qquad\qquad\qquad\qquad\quad rep.rput(k,v)\ \&\ \mathsf{proceed}\{\}\rangle] \\
\quad \mathsf{in}\ NS.register(\text{“}\texttt{replicate}\text{”}, replicate)
\end{array}
$$

This ensures that a cache replication aspect never matches a *put* join point that has been produced by the rule $rput(k,v) \triangleright c.put(k,v)$, thereby ignoring aspect-related computation.

5.1 A temporal logic for pointcuts

In [43], a flat notion of past reactions is introduced, useful for instance to know if the current join point is caused by an untrusted message. However, a flat list of past reactions is limited. We use here a tree-like structure that allows a more precise analysis of causality.

The local causality tree is just defined as a tree of join points whose root is the current join point to be matched, and whose child nodes corresponds to join points that gave rise to their parent join point. Note that we look at the tree “backward” (as it is the case for instance for genealogical trees) in the sense that subnodes in the tree correspond to previous reactions (*i.e.* parents in a genealogical tree).

In what follows, we extend the pointcut language with the primitives of PCTL, computation tree logic with past operators [27]. Note that *future operators* of PCTL will correspond here to looking at *past reactions* and vice versa. This allows pointcut to make precise reasoning on join points occurring in the causality tree. We extend the syntax of pointcuts with temporal operators as follows:

$$
\begin{array}{rcl}
Pc & ::= & \ldots \mid \mathbf{E}[Pc\,\mathbf{U}\,Pc] \mid \mathbf{A}[Pc\,\mathbf{U}\,Pc] \mid \mathbf{E}\,\mathbf{X}\,Pc \\
& & \mid Pc\,\mathbf{S}\,Pc \quad \mid Pc\,\mathbf{X}^{-1}\,Pc \quad \mid \mathbb{T}
\end{array}
$$

Compared to standard temporal logic, the situation here is richer as a pointcut is not simply true when it matches, it produces a substitution τ that can be used in the rest of the pointcut. The linearity condition on variables occurring in a pointcut is now relaxed as a linearity condition between two temporal operators.

A detailed definition of the temporal operators can be found in [27], we only propose here an intuitive description. $\mathbf{E}[Pc_1\,\mathbf{U}\,Pc_2]$ means that there exists a path in the causality tree such that Pc_1 matches with substitution τ until $Pc_2\tau$ holds. $\mathbf{A}[Pc_1\,\mathbf{U}\,Pc_2]$ means that for all paths in the causality tree, Pc_1 matches with substitution τ until $Pc_2\tau$ matches. $\mathbf{E}\,\mathbf{X}\,Pc$ means that Pc matches one of the direct ancestor join points. There are two operators for the past. $Pc_1\,\mathbf{S}\,Pc_2$ means that in the (unique) path to the root, $Pc_1\tau$ matches since Pc_2 matched with substitution τ. $Pc_1\,\mathbf{X}^{-1}\,Pc_2$ means that Pc_1 matches with substitution τ and $Pc_2\tau$ matches the successor join point. Finally, $\mathbb{T}$ is the pointcut that matches every join point with the empty substitution.

5.2 Causality pointcuts

Implementing the whole logic of pointcuts can be very time and space consuming as for instance it requires to keep track of the whole causality tree for every join point. For efficiency reasons, we have isolated three temporal formulas that are of particular interests in the setting of aspects: direct causality, expressed with the pointcut designator $\mathsf{producedBy}$, transitive causality, expressed with $\mathsf{causedBy}$, and a designator that recognizes synchronous patterns, $\mathsf{inThread}$. Note that in Aspect JoCaml (Section 9), we only implement these designators, in order to allow for a more efficient management of join points.

The $\mathsf{producedBy}(Pc)$ pointcut matches whenever Pc matches a direct ancestor in the causality tree. This is directly expressed in our extended pointcut logic:

$$\mathsf{producedBy}(Pc) \ \equiv\ \mathbf{E}\,\mathbf{X}\,Pc$$

Quantifying over the direct causality is often too restrictive, however. Consider a secure cache replication

policy where the activity that (indirectly) comes from an untrusted host H_u is not to be replicated. Then, when need a pointcut $\mathsf{causedBy}(Pc)$ which matches on the first join point in the causality tree for which Pc matches. Using this pointcut, we can alter the cache replication pointcut with $\neg\,\mathsf{causedBy}(\mathsf{host}(H_u))$ to avoid replication of activity coming from the untrusted host H_u. Again, $\mathsf{causedBy}(Pc)$ can be easily expressed in our pointcut logic:

$$\mathsf{causedBy}(Pc) \;\equiv\; \mathbf{E}[\neg Pc\,\mathbf{U}\,Pc]$$

We finally consider a notion of causality that lets us support synchronous communication patterns. Recall that synchronous communication in the join calculus is obtained by passing around a response channel that serves as a way to communicate a result back to the sender of a message.

We first define a general temporal connector, called $\mathsf{inThread}(Pc_1, Pc_2)$, which expresses an unfinished thread of execution; such a thread is specified by a starting event (a join point matched by Pc_1) and a closing event (a join point matched by Pc_2). In other words, inThread matches execution paths in which Pc_1 has matched, and since then, Pc_2 has not matched yet. This is defined in the temporal logic of pointcuts as:

$$\mathsf{inThread}(Pc_1, Pc_2) \equiv \mathbf{E}[\neg Pc_1\,\mathbf{U}(Pc_1\,\mathbf{X}^{-1}\,\neg(\mathbb{T}\,\mathbf{S}(Pc_2)))]$$

We can use inThread to express control flow in the synchronous sense by relating Pc_1 and Pc_2 through their shared response channel. For instance, suppose that we only want to activate cache replication while a critical computation is performed. This computation is triggered by the message $\mathit{critical}(k)$, which asks to return the result on label $k.\mathit{reply}$. We can express this bounded flow of execution with the following pointcut:

$$\mathsf{inThread}(\mathsf{contains}(*.\mathit{critical}(k)),\\ \mathsf{contains}(k.\mathit{reply}(*)))$$

Note that the pointcut $\mathsf{causedBy}(*.\mathit{critical}(k))$ would not be applicable in this case: once a parent join point exists in the causality tree, causedBy never ceases to match. In contrast, inThread makes it possible to capture the notion of bounded sub-computations.

5.3 Translating causality pointcuts

To extend the translation of Section 4 to causality pointcuts, we promote the definition of messages with a fresh first argument that will be used to pass the causality flow—in a similar way than a state is threaded in a monadic computation. That is, a message sending $x.l(\bar{v})$ is translated into $x.l(\mathit{cflow}, \bar{v})$. And the translation of reaction rule is modified as (we only present the case of a pattern with 2 messages for simplicity)

$$\begin{array}{l}[\![l_1(\bar{v_1})\,\&\,l_2(\bar{v_2})\,\triangleright\,P]\!]_x \;\equiv\\ \quad l_1(\mathit{cflow}_1, \bar{v_1})\,\&\,l_2(\mathit{cflow}_2, \bar{v_2})\,\triangleright\\ \qquad \mathsf{obj}\,\mathit{ret}_M = \mathit{proceed}(\mathit{jp})\,\triangleright\\ \qquad\qquad \mathsf{let}\;\bar{v} = \mathsf{fromJp}\;\;M\;\;\mathit{jp}\;\;\mathsf{in}\;[\![P]\!]\\ \qquad \mathsf{in}\;\;\mathsf{let}\;\;\mathit{jp} = (\texttt{lhost}, x.M)\;\mathsf{in}\\ \qquad\quad W.\mathit{weave}(\mathit{Node}(\mathit{jp}, [\mathit{cflow}_1, \mathit{cflow}_2]), \mathit{ret}_M, \mathit{jp})\end{array}$$

where Node is a node constructor for a tree structure.

Then, in the translation of causality pointcuts is done by pattern matching on the causality flow. For instance, the $\mathsf{producedBy}(Pc)$ pointcut is translated into

$$\begin{array}{l}[\![\mathsf{producedBy}(Pc)]\!] \equiv\; \lambda\;\mathit{cflow}.\;\;\mathsf{match}\;\;\mathit{cflow}\;\;\mathsf{with}\\ \qquad\qquad \mid \mathit{Node}(\mathit{jp}, [\mathit{cflow}_i]) \Rightarrow \bigvee_i [\![Pc]\!]\;\mathit{cflow}_i\\ \qquad\qquad \mid \mathit{EmptyFlow} \qquad\quad \Rightarrow \mathit{False}\end{array}$$

where $\mathit{EmptyFlow}$ is the leaf of the causality flow.

The proof of bisimulation is almost unchanged, simply extended to causality pointcuts.

6 Customized weaving

Until now, we have considered a single weaving semantics that applies to all reactions. In practice, however, exposing each and every join point to aspects can be a source of encapsulation breach as well as a threat to modular reasoning. This issue has raised considerable debate in the AOP community [26,40], and several proposal have been made to restrict the freedom enjoyed by aspects (*e.g.* [8,35,41,42]). We now present three variants of weaving semantics.

First of all, it is important for programmers to be able to declare certain reactions as *opaque*, in the sense that they are internal and cannot be woven. This is similar to declaring a method `final` in Java in order to prevent further overriding.

A particularity of aspects compared to traditional event handling is the possibility to advise around join points and therefore have the power to proceed the original computation, either once, several times, or not at all; and possibly with modified arguments. Doing so requires careful thinking about the synchronization of advices. The semantics presented so far correspond to an *asynchronous* reaction, in which all matching advices are triggered asynchronously. We can devise a weaving semantics that rather matches the one of AspectJ by chaining advices and invoking them in a *synchronous* manner.

Finally, for the many cases in which the semantics of asynchronous event handling is sufficient, it is desirable to be able to specify that aspects can only *observe* a

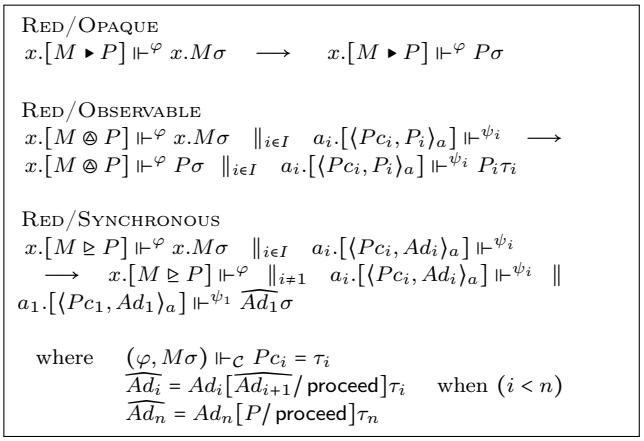

Fig. 9 Per-Reaction Weaving Semantics

given reaction, meaning that advices are not given the ability to do proceed at all, and are all executed in parallel. This gives programmers the guarantee that the original reaction happens unmodified, and that aspects can only "add" to the resulting computation.

The full aspect join calculus therefore includes four possible weaving semantics, which can be specified per-reaction: opaque ($\blacktriangleright$), observable ($\circledcirc$), and two kinds of advisable reactions, asynchronous ($\triangleright$) and synchronous ($\trianglerighteq$). The default semantics is asynchronous advisable.

Figure 9 presents the detailed semantics of the three new reactions. Rule RED/OPAQUE is self explanatory. Rule RED/OBSERVABLE proceeds the original reaction in parallel with the application of all deployed pointcut/advice pairs that match the join point. Note that in this rule, an advice has to be a simple process, and hence can not do proceed (this can be guaranteed by a simple type system). Rule RED/SYNCHRONOUS presents the synchronous semantics when n aspects apply. The first advice that has been deployed, here Ad_1, is executed by replacing proceed with the second advice, and so on up to advice n where proceed is replaced by the original continuation process P. Note that as usual in synchronous AOP, when an advice does not proceed, the rest of the advice chain is not triggered, which is not the case for the asynchronous semantics.

It would be possible to devise more fine-grained variants (for instance, being able to proceed only once, or not being able to change the arguments to proceed with). The four weaving semantics we have presented illustrate how this design space can be explored. We leave a more detailed coverage of this space to future work.

Per-reaction weaving in practice. To illustrate the benefits of different weaving semantics, we refine the definition of a cache object to ensure strong properties with respect to aspect interference:

- reactions on both *put* and *get* are declared *observable*, in order to ensure that aspects cannot prevent them from occurring normally;
- reactions on the internal *getDict* channel are *opaque* to enforce strong encapsulation.

The updated definition is as follows:

```
obj c = put(k,v) & state(d) ◎ dict.update(d,k,v,c)
      or get(k,r) & state(d) ◎ dict.lookup(d,k,r) & c.state(d)
      or getDict(d) ▸ c.state(d)
init dict.create(c)
```

Finally, we also refine the definition of the replication aspect in order to hide reactions on the private *rput* channel. This ensures that replication cannot be tampered with.

```
⊩^φ obj replicate =
      deploy(c,H') ▷ H[go(H'); &
                       obj rep =
                           rput(k,v) ▸ c.put(k,v)
                       or ⟨contains(*.put(k,v)) ∧ ¬ host(H') ∧
                           ¬ producedBy(contains(*.rput(*,*))),
                           rep.rput(k,v) & proceed{}⟩]
    in NS.register("replicate", replicate)
```

7 Decentralized weaving

The description of the semantics of the aspect join calculus so far leaves open the question of the underlying aspect weaving infrastructure. The naive approach consists in relying on a central weaver that coordinates all distributed computation and triggers the weaving of all aspects. This centralized approach has been used in the translation of Section 4 and is described in [43]. A centralized omniscient weaver is however not realistic in a distributed setting.

The first step towards a flexible decentralized weaving architecture is to distribute weavers across several hosts. Having several weavers raises the question of their granularity. Because each reaction can have a specific weaving semantics (Section 6), we consider one weaver per reaction, in charge of performing aspect weaving for a given reaction. Weavers can then cooperate to ensure that all aspects see all computation, although this arguably only makes sense in a local network, and is hardly realistic on a large scale.

In order to mediate between aspects and weavers, we introduce the notion of a *weaving registry*, in direct analogy with, for instance, RMI registries in Java. A weaving registry is in charge of bootstrapping the

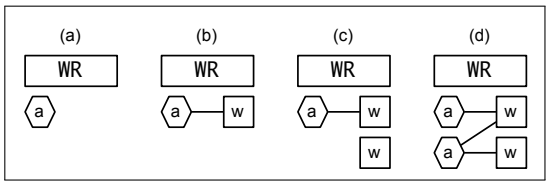

Fig. 10 Weaving registry: (a) an aspect is registered. (b) a weaver is registered, and is connected to the first aspect. (c) a second weaver is registered, not interesting for the existing aspect. (d) a second aspect is registered, interested in both weavers.

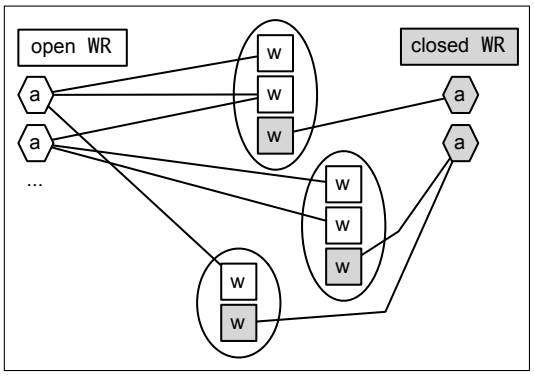

Fig. 11 Objects registering their public weavers to an open weaving registry (white), and the weavers of their sensitive reactions to a closed registry with only two trusted aspects (grey).

binding between weavers and aspects. Both aspects and weavers register to a given registry (Figure 10a). The weaving registry can then connect aspects to weavers (Figure 10b). Once an aspect and a weaver are connected, they communicate directly without the mediation of the weaving registry. As we will see in Section 9, it is possible to partially evaluate the pointcut of an aspect in order to connect it only to the weavers that produce join points that it can possibly match, thereby optimizing network communication. For instance, in Figure 10c, the second weaver is not connected to the first aspect; only the second aspect is (Figure 10d).[8]

As it turns out, distributed aspect deployment is a complex task, for which several different policies can be conceived. This is reflected in the different designs and choices of specific distributed AOP systems, such as DJcutter (one central aspect server) [33], AWED (aspects are either deployed on all hosts, or only on their local host of definition) [4], and ReflexD (distributed aspect repositories to which base programs are connected at start-up time) [46], among others.

[8] Note that we depict aspects and weavers below a weaving registry to reflect the fact that the registry knows them; it does not mean that the aspects and weavers are physically located inside the registry.

Weaving registries are a simple and flexible abstraction that captures the need for precisely specifying the distributed deployment of aspects. This mechanism is very flexible topologically (*e.g.* the case where all aspects see all computation correspond to one global weaving registry to which all weavers and aspects are registered), and allows for fine-grained policies that go beyond existing work. For instance, some weaving registry may be initiated with a number of aspects deployed, and subsequently reject any new aspect registration requests. Other registries may accept dynamic aspect registration requests, with the restriction that these aspects will not be able to react to the computation of previously-registered rules. A weaving registry policy may further specify that only weavers of a specific kind are accepted, such as observable reactions (Section 6). The design space of policies is here again wide and its exhaustive exploration is left open.

Figure 11 illustrates the topological flexibility offered by weaving registries and their policies. Objects can register their reaction weavers in different registries with specific policies. We extend the syntax of reactions with an exponent $M \triangleright^{reg} P$ to express that the reaction is registered in the weaving registry reg.

For instance, suppose a closed weaving registry reg_c with only a cache replication aspect and an open weaving registry reg_o. The following updated definition guarantees that only cache replication can have access to the cache history.

$$\begin{array}{l}\mathsf{obj}\ c = \ put(k,v)\ \&\ state(d) \circledcirc^{reg_c} dict.update(d,k,v,c)\\ \quad\ \mathsf{or}\ \ get(k,r)\ \&\ state(d) \circledcirc^{reg_o} dict.lookup(d,k,r)\ \&\ c.state(d)\\ \quad\ \mathsf{or}\ \ getDict(d) \blacktriangleright^{reg_o} c.state(d)\\ \mathsf{init}\ dict.create(c)\end{array}$$

8 Reifying migration

Locations in the distributed join calculus represent fully-encapsulated "agents", which can be explicitly migrated. As a matter of fact, dealing with locality, for instance to perform load balancing, is a good example of a crosscutting concern, which would be advantageously handled by aspects. In this section, we extend the join point model of the aspect join calculus by considering join points that reflect the migration of locations. This allows the definition of aspects that react to migration requests, possibly adjusting the target host of the migration.

A *migration join point* is a pair (φ, H), where φ is the location at which the migration is about to occur, and H is the target host of the migration. The pointcut $\mathsf{migrateTo}(H)$ matches join points that represent mi-

$$(\varphi, jp) \Vdash_{\mathcal{C}} \mathsf{migrateTo}(H) \equiv \begin{cases} \tau \text{ when } jp = H\tau \\ \bot \text{ otherwise} \end{cases}$$

Fig. 12 Semantics of migration pointcut.

$$\begin{array}{l}
\textsc{Move/NoAsp} \\
\Vdash^{\varphi H} \mathsf{go}(H'); P \longrightarrow \Vdash^{\varphi H} \mathsf{go}_2(H, H'); P \\
\\
\textsc{Move/Asp} \\
\Vdash^{\varphi H} \mathsf{go}(H'); P \quad \|_{i\in I} \quad a_i.[\langle Pc_i, Ad_i\rangle] \Vdash^{\psi_i} \longrightarrow \\
\Vdash^{\varphi H} \|_{i\in I} \\
a_i.[\langle Pc_i, Ad_i\rangle] \Vdash^{\psi_i} Ad_i[(\mathsf{go}_2(H, H'); P)/\mathsf{proceed}]\tau_i \\
\quad \text{where } (\varphi H, H') \Vdash_{\mathcal{C}} Pc_i = \tau_i \\
\\
\textsc{Move-Prim} \\
H[\mathcal{D}: (P \,\&\, \mathsf{go}_2(H, H'); Q)] \Vdash^{\varphi} \| \Vdash^{\psi H'} \longrightarrow \\
\Vdash^{\varphi} \| H[\mathcal{D}: (P \,\&\, Q)] \Vdash^{\psi H'} \\
\\
\textsc{Move-Comm} \\
\Vdash^{\varphi} \mathsf{go}_2(H, H'); P \| \Vdash^{\psi H} \longrightarrow \Vdash^{\varphi} \| \Vdash^{\psi H} \mathsf{go}_2(H, H'); P
\end{array}$$

Fig. 13 Semantics of migration with aspects.

grations to a sub-location of H (Figure 12). As before, if H is a free variable, then the pointcut matches all migration join points, and binds H for use in the advice body. $\mathsf{migrateTo}(\cdot)$ can be combined with $\mathsf{host}(\cdot)$ to use or discriminate based on the location that is being migrated.

To make it possible for aspects to intervene upon migration, and to possibly change the target location of the migration, we replace the migration semantics described by Rule Move of Figure 3 by a two-step process. The source-level go operation reduces to a new internal primitive go_2, whose semantics is the same as the original migration (Rule Move-Prim). The reduction of go depends on whether or not aspects apply. In the case there is no aspect that applies to the migration request, go is simply turned into a go_2 (Rule Move/NoAsp); consequently, Rule Move-Prim applies and carries out the actual migration.

Rule Move/Asp specifies the semantics when some aspects match the migration join point. The syntax of proceed is extended with a new substitution update $\mathsf{proceed}\{\mathsf{re\text{-}go}: H\}$ that changes the target host of the migration:

$$(\mathsf{go}_2(H, H'); P)\{\mathsf{re\text{-}go}: H''\} \equiv \mathsf{go}_2(H, H''); P$$

Finally, note that go_2 explicitly carries along the name of the source host to be migrated. This is necessary because a new migration request can be emitted by an advice on a different host than the host subject to migration. Rule Move-Comm is in charge of transmitting the new go_2 request back to the appropriate source host—similarly to how Rule Message-Comm in Figure 3 transmits messages to the solution where they can react.

Illustration. Consider that a known host H_1 becomes unavailable. The following aspect re-routes all migration requests to host H_1 to another known host H_2:

$$\langle \mathsf{migrateTo}(H_1), \mathsf{proceed}\{\mathsf{re\text{-}go}: H_2\}\rangle$$

The following aspect implements another recovery mechanism: it cancels migration requests to H_1 by actually re-routing them to their source host (free variable H in the pointcut):

$$\langle \mathsf{host}(H) \wedge \mathsf{migrateTo}(H_1), \mathsf{proceed}\{\mathsf{re\text{-}go}: H\}\rangle$$

Note that this aspect is different from one that just does not proceed migrations to H_1 (*i.e.*$\langle \mathsf{migrateTo}(H_1), 0\rangle$), because the continuation process is not skipped; rather, it is performed on the source host.

9 Aspect JoCaml

Aspect JoCaml is an implementation of the aspect join calculus on top of JoCaml, an extension of OCaml with join calculus primitives [19]. The implementation is based on a decentralized version of the translation described in Section 4. The decentralization relies on weaving registries as a bootstrap mechanism (Section 7).

While slightly different in the syntax, Aspect JoCaml supports all the functionalities of the aspect join calculus, except for migration, which is not supported in JoCaml. Using the facilities provided by OCaml, we have also introduced new concepts not formalized in the aspect join calculus, such as classes for both objects and aspects, and the distinction between private and public labels.

This section presents a quick overview of the language through the implementation and deployment of the cache replication example. We then discuss salient points in the implementation.

9.1 Overview of the language

Aspect JoCaml uses directly the class system of OCaml, providing a new `dist_object` keyword to define distributed objects with methods and reactions on public or private labels. For instance, a continuation class that defines a label `k` that expects an integer and prints it to the screen can be defined as:

```
class continuation ip =
  dist_object(self)
  reaction react_k at ip: 'opaque k(n) =
```

```
(* cache class *)
class cache ip dict =
  dist_object(self)
  reaction
     r_get at ip: 'observable
     state(d) & get(k,r) =
              dict#lookup(d,k,r) & state(d)
  or
     r_put at ip: 'observable
     state(d) & put(k,v) =
              dict#update(d,k,v,getDict)
  or
     r_getDict at ip: 'opaque
     getDict(d) = state(d)
  private label state
  public  label get, put, getDict
  initializer spawn dict#create(self#getDict)
end
```

Fig. 14 Cache class in Aspect JoCaml

```
    print_int(n); print_string(" is read\n");0
  public label k
end
```

The label `k` is declared as `public`, meaning that it is visible in a reaction join point. Conversely, a `private` label is not visible, and hence can be neither quantified over nor accessed by aspects. Private labels hence provide another level of encapsulation by hiding patterns, in addition to the possibility to hide reactions discussed in Section 6. The different per-reaction weaving semantics are specified by a quoted keyword, *e.g.* `'observable`.

A reaction definition is parametrized by an IP address using `at`. This IP address is meant to be the address of a weaving registry. The parameter `ip` is passed at object creation time, making it possible to choose a different weaving registry for each created continuation object.

The definition of the cache class is given in Figure 9.1 and can be directly inferred from the definition of Section 6. We omit the code for the dictionary class, which directly uses hash tables provided by the `Hashtbl` OCaml module. A message that creates a dictionary is initially emitted using `spawn` in the `initializer` process.

Aspects are defined as classes with a pointcut and an advice. The instantiation mechanism is identical to that of objects, using the `new` keyword. The cache replication aspect is defined in Figure 9.1. Labels in `Contains` pointcut are handled as strings and boolean pointcut combinators are defined by infix operators `&&&` and `|||`. The only difference with the aspect join calculus in the definition of pointcut/advice is the binding of arguments of the join point. Here, the function `jp_to_arg` must be used with as first argument the label `"l"` of interest and as second argument the join point. This function returns the tuple of arguments passed to `l`, from which each argument can be recovered by explicit pattern matching.

```
(* cache replication aspect *)
class replication ip cache =
  dist_object(self)
  reaction
    react_rput at ip: 'opaque
    rput(k,v) = cache#put(k,v)
  public label rput
end

aspect my_asp ip repl =
   pc: (Contains ["put"] &&&
        Not (ProducedBy (Contains ["rput"])))
   advice: let (k,v) = jp_to_arg "put" jp in
           repl#rput(k,v)
end
```

Fig. 15 Cache replication aspect in Aspect JoCaml

Deployment. Before creating any process, at least one weaving registry must be created and registered to the name server. For instance, the following code creates a permanent weaving registry at IP `12345`:

```
(* create a permanent weaving registry*)
let () =
    let _ = new weaving_registry 12345 in
    while true do Thread.delay 1.0 done
```

Then, a cache replication aspect can be registered to this weaving registry:

```
(* register a cache replication aspect *)
let () =
  let ip   = 12345 in
  let dict = new Dict.dict ip in
  let buf  = new cache ip dict in
  let repl = new replication ip buf in
  let _    = my_asp ip repl in
  while true do () done
```

Finally, the execution of the cache process defined below is replicated on the machine where the aspect has been deployed:

```
(* a cache process loop *)
let () =
 let ip   = 12345 in
 let dict = new Dict.dict ip in
 let z    = new cache ip dict in
 let k    = new continuation ip in
 for arg = 1 to 10 do
   spawn z#put("key",arg);
   spawn z#get("key",k#k)
 done;
```

9.2 Implementation

We now briefly discuss some elements of the Aspect JoCaml implementation.

Architecture. An Aspect JoCaml file is translated into a JoCaml file and then compiled using the JoCaml compiler. To simplify the parser, there are { ... } separators for plain JoCaml code (for clarity, those separators have been omitted in Figure 9.1). While these separators clutter the code, they have the advantage that new features of JoCaml or OCaml can be directly back ported to Aspect JoCaml.

A more advanced solution would be to use Camlp5, the preprocessor-pretty-printer of OCaml, to produce a type-safe translation. Unfortunately, compatibility issues between Camlp5 and JoCaml forbids this solution at the moment.

Typing issues. As the code produced is compiled using JoCaml, everything needs to be typed. Sometimes, this requires type annotations in class definitions when dealing with parametric polymorphism.

However, as mentioned in the JoCaml manual: "communications through the name server are untyped. This weakness involves a good programming discipline."[30] On the one hand, this limitation of distributed programming in OCaml simplifies the task of creating a list of aspects of different types. On the other hand, to avoid type errors at runtime, an anti-unification mechanism has to be developed to guarantee type safe application of aspects [44].

Static/dynamic pointcuts. The weaving registry is responsible for bootstrapping the communication between weavers and aspects. This is performed by adding aspects to the list of current aspects connected to the weaver. But part of communications between weavers and aspects can be avoided. Indeed, it is sometimes possible to statically decide whether a pointcut can match a join point coming from a given weaver. If the pointcut can never match, the weaving registry does not need to register the aspect to the weaver.

To that end, our implementation differentiates between the static and dynamic parts of a join point. The static part is used at registration time, whereas the dynamic part is used during runtime weaving.

Bounded depth of causality tree. Optimized, scalable management of the causality tree discussed in Section 5 has not been investigated yet. The current implementation is naive, keeping track of every causal match. This means that the causality tree may grow unboundedly. Therefore, a bounded version of the causality tree has to be implemented. But this raises the issue of deciding the size of the tree, because dealing with a bounded tree changes the semantics of pointcut matching. It remains to be studied how existing optimization techniques for control flow pointcuts [31] and trace-based matching [3] can be adapted to the general setting of the causality tree.

10 Related work

We first discuss work related to the formal semantics of aspects, and then relate to existing distributed aspect languages and systems.

10.1 Formal semantics of aspects

There is an extensive body of work on the semantics of aspect-oriented programming languages (*e.g.* [49,10, 13,23,12]). These languages adopt either the lambda calculus or some minimal imperative calculus at their core. To the best of our knowledge, this work is the first to propose a chemical semantics of aspects. In addition, none of the formal accounts of AOP considers distributed aspects. Among practical distributed aspect systems, only AWED exposes a formal syntax; the semantics of the language is however only described informally [4].

The approach of starting from a direct semantics with aspects, and then defining a translation to a core without aspects and proving the correctness of the transformation is also used by Jagadeesan et al., in the context of an AspectJ-like language [23].

10.2 Distributed aspect languages and systems

We now compare specific features of practical distributed aspect languages and systems—in particular JAC [36], DJcutter [33], ReflexD [46], and AWED [4]—and relate them to the aspect join calculus.

Quantification. Remote pointcuts were first introduced in DJcutter and JAC, making it possible to specify on which hosts joint points should be detected. Remote pointcuts are also supported in AWED, ReflexD, and in the aspect join calculus, in a very similar fashion. Remote pointcuts can be seen as a necessary feature for distributed AOP (as opposed to using standard AOP in a distributed setting).

Hosts. Remote pointcuts bring about the necessity to refer to execution hosts. In DJcutter and AWED, hosts are represented as strings, while in ReflexD they are reified as objects that give access to the system properties of the hosts. The host model in ReflexD is therefore general and expressive, since host properties constitute

an extensible set of metadata that can be used in the pointcuts to denote hosts of interest. In the aspect join calculus, we have not developed locations beyond the fact that they are first class values. A peculiarity is that locations are organized hierarchically, and can possibly represent finer-grained entities than in existing systems (for instance, a locality can represent an actor within a virtual machine within a machine). A practical implementation should consider the advantages of a rich host metadata model as in ReflexD. AWED and ReflexD support dynamically-defined groups of hosts, as a means to deal with the distributed architecture in a more abstract manner than at the host level.

Weaving semantics. Most distributed AOP languages and systems adopt a synchronous aspect weaving semantics. This is most probably due to the fact that the implementation is done over Java/RMI, in which synchronous remote calls is the standard. Notably, AWED supports the ability to specify that some advices should be run asynchronously. The aspect join calculus is the dual: the default is asynchronous communication, but we can also express synchronous weaving (Section 6). In addition, we have developed the ability to customize the weaving semantics on a per-reaction basis. An interesting consequence of this granularity is that we are able to express opaque and observable reactions. Both kinds of reactions support stronger encapsulation and guarantees in presence of aspects, and therefore fit in the line of work on modular reasoning about aspects [8, 35, 41, 44].

Advanced quantification. DJcutter, AWED, and ReflexD support reasoning about distributed control flow, in order to be able to discriminate when a join point in the (distributed) flow of a given method call. AWED also support state-machine-like pointcuts, called stateful aspects, which are able to match sequences of events that are possibly unrelated in terms of control flow. However, stateful aspects per se do not make it possible to reason about causality; additional support is needed, for instance as developed in WeCa [28]. In Section 5, we describe a temporal logic for pointcuts that can be used to reason about causality. While the synchronous communication pattern can be recognized in order to support a similar notion of distributed control flow, the model is much more general. Temporal logic has been used in some aspect-oriented systems to perform semantic interface conformance checks [7]. Causality in widely-asynchronous (distributed) contexts is a topic of major interest. It would be interesting to study how our approach relates to the notion of causality in the π-calculus proposed by Curti et al. in the context of modeling biochemical systems [11].

Aspect deployment. DJcutter adopts a centralized architecture with an aspect host where all aspects reside and advices are executed. JAC allows distributed aspect deployment to various containers with a consistency protocol between hosts, to ensure a global view of the aspect environment. Both AWED and ReflexD adopt a decentralized architecture, in which it is possible to execute advices in different hosts: multiple parallel advice execution in specific hosts is possible, and programmers can control where aspects are deployed. ReflexD is more flexible than AWED in the localization of advices and in deployment, by supporting stand-alone aspect repositories to which a Reflex host can connect. The weaving registries mechanism we have described in Section 7 subsumes these mechanisms, and also adds support for controlling the openness of the distributed architecture.

JAC, AWED and Reflex support dynamic undeployment of aspects. While we have not introduced undeployment in this paper, it is trivial to add it to the core calculus. More interesting, in previous work we explore structured deployment through *scoping strategies* [45]. Scoping strategies make it possible to specify the computation that is exposed to a given aspect in a very precise manner. The model of scoping strategies relies on per-value and per-control-flow propagation of aspects; it would be not trivial, but interesting, to study how these strategies can be adapted to a chemical setting.

Parameter passing. In Java, remote parameter passing is by-copy, unless for remote objects that are passed by-reference. ReflexD allows to customize the remote parameter passing strategy for each parameter passed to a remote advice. The join calculus has a by-reference strategy, where names act as references. It would be possible to add a by-copy mechanism in the aspect join calculus, by adding a rule to clone named definitions.

Migration. To the best of our knowledge, there has not been any proposal related to aspects with code mobility where aspects can take control of migration and perform re-routing. This re-routing is however interesting when considering load balancing and fault tolerance, both of which are rather crosscutting concerns. We have shown how we can extend the join point model of the calculus to expose migrations as join points (Section 8. Distributed scoping strategies make it possible to specify that an aspect should be migrated along with a given object [45]. This can be achieved in the aspect join calculus by attaching 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 the object.

11 Conclusions

This article describes a formal foundation for distributed aspect-oriented programming based on a chemical calculus. More precisely, we extend an objective and distributed version of the join calculus with means to address crosscutting through pointcuts and advices. The semantics of the aspect join calculus is given both directly and by translation to the standard join calculus. The latter translation is proven correct by a bisimilarity argument, and is the basis for implementing the Aspect JoCaml language on top of JoCaml. We explore different dimensions of extensions to the core calculus: advanced quantification through causality, customized weaving semantics with stronger encapsulation and non-interference guarantees, decentralized aspect weaving, and the possibility to advise migration of localities. This work shows that the main features of previous distributed AOP systems can be expressed by the few relatively simple constructs of the calculus, and that the calculus can even go beyond existing proposals. We believe the aspect join calculus can serve as a solid formal basis on top of which to explore and compare distributed aspect language features. A particular feature of interest, which we have not addressed so far, is dealing with failures. Fournet and Gonthier briefly describe an extension of the join calculus with partial failure and remote failure detection [16]. The aspect join calculus can also serve as a basis to implement concurrent and distributed aspects in other languages for which a variant of the join calculus has been developed, such as Cω [6] and Scala Joins [21]. Another perspective is to study the application of aspects in chemical engines for Cloud computing.

References

1. *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, Rennes and Saint Malo, France, March 2010. ACM Press.
2. Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag, February 2006.
3. Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer-Verlag, 2006.
4. Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the 5th ACM International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 51–62, Bonn, Germany, March 2006. ACM Press.
5. Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, and Bart Verheecke. Modularization of distributed web services using AWED. In *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA'06*, volume 4276 of *Lecture Notes in Computer Science*, pages 1449–1466. Springer-Verlag, October 2006.
6. Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C♯. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, September 2004.
7. Eric Bodden and Volker Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In Welf Löwe and Mario Südholt, editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, pages 147–162, Vienna, Austria, March 2006. Springer-Verlag.
8. Eric Bodden, Éric Tanter, and Milton Inostroza. Join point interfaces for safe and flexible decoupling of aspects. *ACM Transactions on Software Engineering and Methodology*, 2013. To appear.
9. Hsing-Yu Chen. COCA: Computation offload to clouds using AOP. In *Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 466–473, Ottawa, ON, USA, 2012.
10. Curtis Clifton and Gary T. Leavens. MiniMAO$_1$: An imperative core language for studying aspect-oriented reasoning. *Science of Computer Programming*, 63:312–374, 2006.
11. Michele Curti, Pierpaolo Degano, and Cosima Tatiana Baldari. Causal π-calculus for biochemical modelling. In *Computational Methods in Systems Biology*, volume 2602 of *Lecture Notes in Computer Science*, pages 21–34. Springer-Verlag, February 2003.
12. Bruno De Fraine, Erik Ernst, and Mario Südholt. Essential AOP: the A calculus. In Theo D'Hondt, editor, *Proceedings of the 24th European Conference on Object-oriented Programming (ECOOP 2010)*, number 6183 in Lecture Notes in Computer Science, pages 101–125, Maribor, Slovenia, June 2010. Springer-Verlag.
13. Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Science of Computer Programming*, 63(3):207–239, December 2006.
14. Gary Duzan, Joseph Loyall, Richard Schantz, Richard Shapiro, and John Zinky. Building adaptive distributed applications with middleware and aspects. In Lieberherr [29], pages 66–73.
15. Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), October 2001.
16. C. Fournet and G. Gonthier. The join calculus: a language for distributed mobile programming. In *Applied Semantics*, volume 2395 of *Lecture Notes in Computer Science*, pages 268–332. Springer-Verlag, 2002.
17. Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL'96*, pages 372–385. ACM, January 1996.
18. Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Inheritance in the join calculus. *Journal of Logic and Algebraic Programming*, 57(1):23–70, 2003.
19. Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. JoCaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 129–158. Springer-Verlag, 2003.

20. Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: modular event-driven object interactions in Scala. In *Proceedings of the 10th ACM International Conference on Aspect-Oriented Software Development (AOSD 2011)*, pages 227–240, Porto de Galinhas, Brazil, March 2011. ACM Press.
21. Philipp Haller and Tom Van Cutsem. Implementing joins using extensible pattern matching. In Doug Lea and Gianluigi Zavattaro, editors, *Proceedings of the 10th International Conference on Coordination Models and Languages (COORDINATION 2008)*, volume 5052 of *Lecture Notes in Computer Science*, pages 135–152, Oslo, Norway, June 2008. Springer-Verlag.
22. Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In Lieberherr [29], pages 26–35.
23. Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 63:267–296, 2006.
24. G. Kiczales, J. Irwin, J. Lamping, J. Loingtier, C.V. Lopes, C. Maeda, and A. Mendhekar. Aspect oriented programming. In *Special Issues in Object-Oriented Programming*. Max Muehlhaeuser (general editor) et al., 1996.
25. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Jorgen L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.
26. Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering (ICSE 2005)*, pages 49–58, St. Louis, MO, USA, 2005. ACM Press.
27. F. Laroussinie and P. Schnoebelen. A hierarchy of temporal logics with past. *Theoretical Computer Science*, 148(2):303–324, 1995.
28. Paul Leger, Éric Tanter, and Rémi Douence. Modular and flexible causality control on the web. *Science of Computer Programming*, December 2012. Available online.
29. Karl Lieberherr, editor. *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK, March 2004. ACM Press.
30. Louis Mandel and Luc Maranget. *The JoCaml language Release 4.00*. INRIA, august 2012.
31. Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. A compilation and optimization model for aspect-oriented programs. In G. Hedin, editor, *Proceedings of Compiler Construction (CC2003)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.
32. A. Mdhaffar, R. Ben Halima, E. Juhnke, and M. Jmaiel. AOP4CSM: An aspect-oriented programming approach for cloud service monitoring. In *Proceedings of the 11th IEEE International Conference on Computer and Information Technology (CIT)*, pages 363–370, 2011.
33. Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut – a language construct for distributed AOP. In Lieberherr [29], pages 7–15.
34. Marko Obrovac and Cédric Tedeschi. Experimental evaluation of a hierarchical chemical computing platform. *International Journal of Networking and Computing*, 3(1):37–54, 2013.
35. Bruno C. d. S. Oliveira, Tom Schrijvers, and William R. Cook. EffectiveAdvice: discplined advice with explicit effects. In AOSD 2010 [1], pages 109–120.
36. Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, and Laurent Martelli. JAC: an aspect-oriented distributed dynamic framework. *Software—Practice and Experience*, 34(12):1119–1148, 2004.
37. Jean-Louis Pazat, Thierry Priol, and Cédric Tedeschi. Towards a chemistry-inspired middleware to program the internet of services. *ERCIM News*, 85(34), 2011.
38. James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Proceedings of POPL'98*, pages 378–390. ACM Press, 1998.
39. Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2002)*, pages 174–190, Seattle, Washington, USA, November 2002. ACM Press. ACM SIGPLAN Notices, 37(11).
40. Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2006)*, pages 481–497, Portland, Oregon, USA, October 2006. ACM Press. ACM SIGPLAN Notices, 41(10).
41. Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology*, 20(1):Article 1, June 2010.
42. Kevin Sullivan, William G. Griswold, Hridesh Rajan, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, and Nishit Tewari. Modular aspect-oriented design with XPIs. *ACM Transactions on Software Engineering and Methodology*, 20(2), August 2010. Article 5.
43. Nicolas Tabareau. A theory of distributed aspects. In AOSD 2010 [1], pages 133–144.
44. Nicolas Tabareau, Ismael Figueroa, and Éric Tanter. A typed monadic embedding of aspects. In Jörg Kinzle, editor, *Proceedings of the 12th International Conference on Aspect-Oriented Software Development (AOSD 2013)*, pages 171–184, Fukuoka, Japan, March 2013. ACM Press.
45. Éric Tanter, Johan Fabry, Rémi Douence, Jacques Noyé, and Mario Südholt. Scoping strategies for distributed aspects. *Science of Computer Programming*, 75(12):1235–1261, December 2010.
46. Éric Tanter and Rodolfo Toledo. A versatile kernel for distributed AOP. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, volume 4025 of *Lecture Notes in Computer Science*, pages 316–331, Bologna, Italy, June 2006. Springer-Verlag.
47. Eddy Truyen and Wouter Joosen. Run-time and atomic weaving of distributed aspects. *Transactions on Aspect-Oriented Software Development II*, 4242:147–181, 2006.
48. David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proceedings of the 8th ACM SIGPLAN Conference on Functional Programming (ICFP 2003)*, pages 127–139, Uppsala, Sweden, September 2003. ACM Press.
49. Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems*, 26(5):890–910, September 2004.

A Proof of bisimulation between an aspect join calculus process and its translation

Recall Theorem 1 from Section 4.2.

Theorem 1 *The relation* $\mathcal{R} = \{(\mathcal{C}_0, \mathcal{C}_1) \mid \widetilde{\mathcal{C}_1} \overset{\text{proc}}{\sim} [\![\mathcal{C}_0]\!]\}$ *is a bisimulation. In particular, any 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 Deploy, Red/Asp or Red/NoAsp, one can find a corresponding reduction chains from $[\![\mathcal{C}_0]\!]$ to $[\![\mathcal{C}'_0]\!]$:

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

Consider the case of Rule Red/Asp (Rule Red/NoAsp is easier):

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

This rule is simulated by the chain:

$$\begin{array}{rcl} x.l(\bar{v}) & \longrightarrow & w.weave(k, jp) \\ & \longrightarrow & Ad_1.advice(k, jp) \,\&\, \cdots \,\&\, Ad_n.advice(k, jp) \\ & \longrightarrow & [\![Ad_1]\!]_{a_1} \,\&\, \cdots \,\&\, [\![Ad_n]\!]_{a_n} \end{array}$$

leading the normal form $\widetilde{\mathcal{C}'_1} \overset{\text{proc}}{\sim} [\![\mathcal{C}'_0]\!]$. Rule Deploy is simulated by the first reaction rule in the definition of the weaver.

The converse direction is more interesting as it says that any reduction in the translated configuration can be seen as a step in the simulated reduction of Rule Deploy, Red/Asp or Red/NoAsp 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:

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

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 $proceed(jp)$, then $\widetilde{\mathcal{C}'_1} \overset{\text{proc}}{\sim} \widetilde{\mathcal{C}_1} \overset{\text{proc}}{\sim} [\![\mathcal{C}_0]\!]$. If it introduces a message $deploy(pc, ad)$, then $\mathcal{C}_0 \equiv \mathcal{C}'_0$ and $\widetilde{\mathcal{C}'_1} \overset{\text{proc}}{\sim} [\![\mathcal{C}'_0]\!]$. If it consumes a message $deploy(pc, ad)$, then $\mathcal{C}'_0$ is obtained by applying Rule Deploy to the corresponding pointcut/advice pair and $\widetilde{\mathcal{C}'_1} \overset{\text{proc}}{\sim} [\![\mathcal{C}'_0]\!]$. Otherwise, the reduction consumes a pattern $x.M_\sigma$ and produces a message of the form:

$$w.weave(k, jp).$$

Then, if some aspects match the join point, $\mathcal{C}'_0$ is obtained by applying Rule Red/Asp to $x.M \rhd P \Vdash^{\varphi} x.M_\sigma$, and if no aspect match, $\mathcal{C}'_0$ is obtained by applying Rule Red/NoAsp to $x.M \rhd P \Vdash^{\varphi} x.M_\sigma$. The fact that the diagram above commutes is a direct consequence of the confluence of $\longrightarrow_{\mathbb{T}}$ and its non-interference with other reductions of the system.

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

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 messages sent on auxiliary labels (*weave*, *proceed*, *advice* or *deploy*). But we are interested in equivalent behavior of two closed configurations, not of two terms that can appear in any context, so a simple bisimulation is sufficient.