Chemical foundations of distributed aspects

Nicolas Tabareau, Éric Tanter

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 chemical reactions. 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.

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
Inria, Nantes, 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 support the separation of 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 [30,20]. 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] The implementation of Aspect JoCaml, together with running examples, is available online at:
`https://tabareau.github.io/AJoCaml/`

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 [38, 11], 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 [49].

To address these limitations, distributed AOP has been the focus of several practical developments: JAC [44], DJcutter [39], QuO's ASL [19], ReflexD [57], AWED [4, 5], Lasagne [59], as well as a higher-order procedural language with distribution and aspects [55]. 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. Most of 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 defining the formal semantics of distributed aspects is almost impossible. While the formal foundations of aspects have been laid out in the sequential setting [62,15], 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 [22]. The join calculus is a functional name-passing calculus based on the chemical abstract machine and implemented in several mainstream languages like OCaml [24], C# [6] and Scala [26]. Chemical execution engines are also being developed for Cloud computing [45,40]. 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 makes 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 channels are synchronized and processed. The crosscutting phenomena manifest 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 syntax and semantics of the aspect join calculus are described in Section 3. In order to address the implementation of the aspect join calculus, Section 4 describes a translation from the aspect join calculus to the core join calculus; the correctness of this translation is proven by a bisimilarity argument. Section 5 discusses several design options. Then, Section 6 describes Aspect JoCaml, a prototype implementation of the aspect join calculus on top of JoCaml based on the translation described in Section 4. Finally, Section 7 discusses related work and Section 8 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 [23] and an explicit notion of location to account explicitly for distribution [21].

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

[2] `http://spring.io`

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

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

[5] This article builds upon a previous conference publication [53]. 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, including more expressive quantification, per-reaction weaving, and decentralized weaving. The translation from the aspect join calculus to the standard join calculus has been updated accordingly, as well as the proof of correctness of the translation. Finally, an implementation based on JoCaml is presented 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 3.4, once the basics of aspects in the calculus are established.

buffer presented in [23]. The basic operation of the join calculus is asynchronous message passing over channels. In the objective join calculus, channels are associated with an object and called *labels*. Accordingly, the definition of an object describes how messages received on some labels can trigger processes. For instance, the term

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

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:

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

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 in 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 [23].

2.2 Syntax

The grammar of the distributed objective join calculus is given in Figure 1; it has syntactic categories for processes P, definitions D, patterns M, and named definitions $\mathcal{D}$. 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.* $v \in \mathcal{O} \cup \mathcal{H}$.

To introduce all the syntactic constructs of the distributed objective join calculus, it is helpful to start by considering that a program is described as a configuration $\mathcal{C}$, called a *chemical solution*, which is a set of machines running in parallel:

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

$P ::=$		**Processes**
	0	null process
	$x.M$	message sending
	$\mathsf{obj}\ x = D\ \mathsf{in}\ P$	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
	$\top$	void definition
$M ::=$		**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 [21] and the objective join calculus [23])

A machine $\mathcal{D} \Vdash^{\varphi} \mathcal{P}$ consists of a set of *named definitions* $\mathcal{D}$ and of a multiset of *processes* $\mathcal{P}$ running in parallel at a given *location* φ.

Locations. A location φ (we also sometimes use metavariable ψ to denote locations) is a unique sequence of host names H, *i.e.* $\varphi = H_1 \cdots H_n$. We assume that the rightmost host H_n defines the location φ uniquely.

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. Then, 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. For instance, a concrete representation of locations (using | as a separator between Hs) could be `1.2.3.4:56`$|vm_1|t_2$ to denote thread t_2 of virtual machine vm_1 running at IP address `1.2.3.4:56`.

Named definitions. Named definitions $\mathcal{D}$ are a disjunction of object definitions $x.D$, where x is an object name, and D is a disjunction of reaction rules. Object definitions in $\mathcal{D}$ represent "active" objects ready to react to message sends. 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. Note that in the join calculus, it is required that every pattern M guarding a reaction rule be linear, that is, labels and object names appear at

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

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

most once in M. Also, each object is associated with exactly one named definition.

Named definitions also include sub-location definitions $H[\mathcal{D}:P]$, hosting the named definitions $\mathcal{D}$ and process P at host H.

Processes. Processes include the null process 0, message sending $x.M$, and object definition $\mathsf{obj}\ x = D\ \mathsf{in}\ P$, which corresponds to the creation of a new object (not yet ready to react). An object definition 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 process P. Objects are taken modulo renaming of bound names (or α-conversion). $H[P]$ 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 any location of the form $\psi H'$ (which must be unique by construction) 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* [22]. Each rewrite rule applies to a configuration $\mathcal{C}$. 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. In chemical semantics, each rule is local in the sense that it mentions only definitions and messages involved in the reaction; but it can be applied to a wider chemical solution that contains those definitions and messages. By convention, the rest of the solution, which remains unchanged, is implicit.

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-renaming, we can consider that any definition of an object x appears only once in a configuration).

The reduction rule Red specifies 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 $P\sigma$. Substitution is standard, replacing free occurrences (as defined by fn) of the variable to substitute.

Distribution. Rule Message-Comm states that a message emitted in a given location φ on an object 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 [46], the routing of messages is not explicitly described by the calculus.

The handling of locations and migration is directly based on the join calculus mechanisms presented by Fournet and Gonthier [21]. 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. Sub-Loc rule is best understood in tandem with the Move rule, which gives the semantics of migration. Intuitively, the Move rule dispatches a pack of definitions and processes to a new location, and the Sub-Loc rule allows unpacking.

More precisely, in the Move rule 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

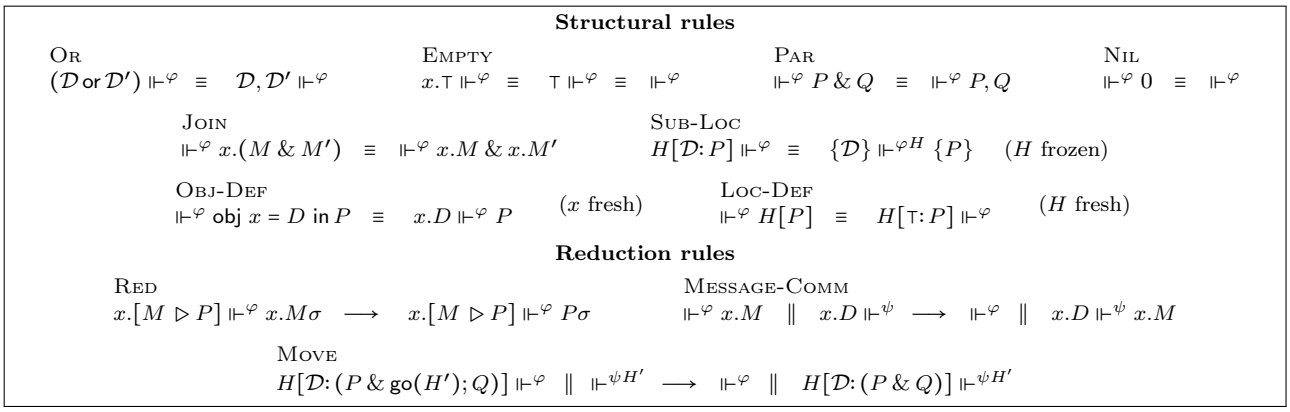


Fig. 3 Chemical semantics of the distributed objective join calculus (adapted from [21,23])

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 [21]. Similarly, the freshness condition of rule Loc-Def preserves the assumption that the rightmost host H_n uniquely defines the location φ.

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 $\mathcal{C} = (\mathcal{D}_i \Vdash^{\varphi_i} \mathcal{P}_i)_i$, 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 $H[\mathcal{D}\colon\mathcal{P}]$ 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 with 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:

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

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

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

At this point, two reactions can be performed, involving $c.state(d_0)$ and either $c.get(\text{“}\texttt{foo}\text{”}, r)$ or $c.put(\text{“}\texttt{foo}\text{”}, 5)$. 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:

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

2.5 Bootstrapping distributed communication

Since the join calculus is lexically scoped, programs 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 with a constant string—$\mathsf{NS}.register(\text{“}\texttt{x}\text{”}, x)$—and to look up a name based on a string —$\mathsf{NS}.lookup(\text{“}\texttt{x}\text{”}, r)$, where the value is sent on *r.reply*.

For instance, in the above example we (magically) assumed that *dict* was in scope. Recall that we wrote:

$$\mathsf{obj}\ c = \ldots \mathsf{in}\ dict.create(c)$$

The actual bootstrapping through the name server would occur as follows. First, the dictionary object should be created and registered:

$$\mathsf{obj}\ dict = \ldots \\ \mathsf{in}\ \mathsf{NS}.register(\text{“}\texttt{dict}\text{”}, dict)$$

Then, the client program can query the name server to obtain the dictionary, and then use it:

$$\mathsf{obj}\ client = reply(d) \triangleright \mathsf{obj}\ c = \ldots \mathsf{in}\ d.create(c) \\ \mathsf{in}\ \mathsf{NS}.lookup(\text{“}\texttt{dict}\text{”}, client)$$

Finally, note that in order to make the definition of processes more readable, we present some part of processes 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 the notions of lists, strings, integers, equality testing, conditionals (`if-then-else`), and a particular variable `lhost` that represents the current location on which a process is executing.

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 [37]. 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.

$D ::=$	$\ldots$	**Definitions**
	$D\ \mathsf{or}\langle Pc, Ad\rangle$	pointcut/advice pair
	$D\ \mathsf{or}\langle Pc, Ad\rangle^\circ$	activated aspect
$Pc ::=$		**Pointcuts**
	$\mathsf{contains}(x.M)$	reaction pattern binder
	$\mathsf{host}(h)$	location binder
	$\neg Pc$	negation
	$Pc \wedge Pc$	conjunction
	$\mathsf{causedBy}(Pc)$	causality
$Ad ::=$		**Advices**
	P	any process
	$Ad \,\&\, Ad$	parallel composition
	$\mathsf{proceed}$	proceed

Fig. 4 Syntax of aspects in the aspect join calculus

$J ::=$		**Join points**
	$\bullet$	empty join point
	$jp, \bar{J}$	join point with causality
$jp ::=$	$(\varphi, x.M)$	reaction join point

Fig. 5 Syntax of join points

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 [31] join points are method invocations, field accesses, etc. In functional aspect-oriented programming languages, join points are typically function applications [18,61].

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 jp* in the aspect join calculus is a pair $(\varphi, x.M)$, where φ is the location at which the reduction occurs, and $x.M$ is the matched synchronization pattern of the reduction.

In a pointcut definition, it is often of interest to know not only the current reaction join point but also the *causality tree* of reaction join points that gave rise to it.[7] Therefore, we introduce a general notion of *join*

[7] We further discuss dealing with causality in Section 5.1.

$$
\begin{array}{lcl}
\mathsf{fn}(\mathsf{contains}(x.M)) & = & \mathsf{fn}(x.M) \\
\mathsf{fn}(\mathsf{host}(h)) & = & h \\
\mathsf{fn}(Pc \wedge Pc') & = & \mathsf{fn}(Pc) \cup \mathsf{fn}(Pc') \\
\mathsf{fn}(\neg Pc) & = & \mathsf{fn}(Pc) \\
\mathsf{fn}(\mathsf{causedBy}(Pc)) & = & \mathsf{fn}(Pc) \\
\mathsf{fn}(\mathsf{proceed}) & = & \varnothing \\
\mathsf{fn}(\langle Pc, Ad \rangle) & = & \mathsf{fn}(Ad) \setminus \mathsf{fn}(Pc)
\end{array}
$$

Fig. 6 Definition of free names for aspects

point (with causality) J to denote a tree of reaction join points. The syntax of join points is given in Figure 5: a join point is either an empty tree, noted $\bullet$, or a join point with causality, noted $jp, \bar{J}$, where jp is a reaction join point and $\bar{J}$ is a list of join points.

We note $J' \prec J$ to indicate that J' is a subtree of J, *i.e.* J' is a *sub join point* of J. Formally, $J' \prec J$ is inductively defined as by the following two rules:

$$
\begin{array}{c}
\prec_{\mathtt{now}} : \forall i,\ J = J_i \Rightarrow J \prec (jp, [J_1, \ldots, J_n]) \\
\prec_{\mathtt{next}} : \forall i,\ J \prec J_i \Rightarrow J \prec (jp, [J_1, \ldots, J_n])
\end{array}
$$

For instance, consider messages $c.state(d)$ with history J_1 and $c.get(\text{“}\mathtt{foo}\text{”}, r)$ with history J_2. The join point of the corresponding reaction of both messages on host φ is:

$$((\varphi, c.state(d)\ \&\ c.get(\text{“}\mathtt{foo}\text{”}, r)), [J_1, J_2])$$

We give an example of reduction with causality in Section 3.3.

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, where the variables occurring in $\mathsf{contains}(x.M)$ are bound to the values involved in the reaction join point. In the same way, the pointcut $\mathsf{host}(h)$ binds h to the location of the reaction join point. A pointcut can also be constructed by negations and conjunctions of other pointcuts. Finally, the pointcut $\mathsf{causedBy}(Pc)$ says that Pc matches for a subtree of the current join point. The semantics of pointcuts is formally described in Section 3.3.

The free variables of a pointcut (as defined in Figure 6) are bound to the values of the matched join points. In this way, a pointcut acts as a binder of the free variables occurring in the corresponding advice, as standard in aspect-oriented languages. 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 unions of substitutions used in the definition of a semantics of pointcuts (Figure 7) are 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. Note that contrarily to the common practice in AOP, it is not possible to modify the process P by altering the substitution that is applied to it. This is because the notion of arguments of a reaction is not easy to set up in the join calculus as it should be induced by the substitution and not by the order in which they appear in the reaction join point. Nevertheless, it is still possible to skip using proceed and trigger another process instead. Free names of an advice are defined in Figure 6.

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 [25], where any object has the potential to behave as an aspect. Free names of an aspect are defined in Figure 6; the only interesting case is the last one, which specifies that the free variables of a pointcut act as binders in the advice.

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:

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

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 $(h \neq H')$ in the advice 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 Customized reactions

With a single notion of reaction, we are forced to consider a single weaving semantics that applies uniformly 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 [32,50], and several proposal have been made to restrict the freedom enjoyed by aspects (*e.g.* [10,41,42,51,52]). 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.

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

The full aspect join calculus therefore includes three possible weaving semantics, which can be specified per-reaction: opaque ($\blacktriangleright$), observable ($\circledcirc$), and asynchronously advisable ($\triangleright$). The default semantics is asynchronous advisable.

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 as follows:

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

Reactions on both *put* and *get* are declared *observable*, in order to ensure that aspects cannot prevent them from occurring normally. In particular, the replication aspect is not allowed to call proceed, which is anyway implicitly called in parallel with the advice:

$$\begin{array}{lcl}(\varphi, x'.M'), \bar{J} \Vdash \mathsf{contains}(x.M) & = & \begin{cases}\tau \text{ minimal substitution s.t.} \\ \quad x\tau = x' \text{ and } M\tau \subseteq M' \\ \bot \text{ otherwise}\end{cases}\\ (\varphi, x'.M'), \bar{J} \Vdash \mathsf{host}(h) & = & \{h \mapsto \varphi\}\\ J \Vdash Pc \wedge Pc' & = & J \Vdash Pc \cup J \Vdash Pc'\\ J \Vdash \neg Pc & = & \begin{cases}\{\ \} \text{ when } J \Vdash Pc = \bot \\ \bot \text{ otherwise}\end{cases}\\ J \Vdash \mathsf{causedBy}(Pc) & = & \begin{cases}J' \Vdash Pc \text{ for some } J' \text{ s.t.} \\ \quad J' \prec J \text{ and } J' \Vdash Pc \neq \bot \\ \bot \text{ otherwise}\end{cases}\end{array}$$

Fig. 7 Semantics of pointcuts

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

Additionally, reactions on the internal *getDict* label of the cache object are now *opaque*, hence enforce strong encapsulation: no aspect can observe such reactions.

3.3 Semantics

Semantics of pointcuts. The matching relation, noted $jp \Vdash Pc$, returns either a substitution τ from free names of Pc to names or values of jp, or a special value $\bot$ meaning that the pointcut does not match. That is, we enriched the notion of boolean values to a richer structure (here substitutions), as commonly done in aspect-oriented programming languages in particular. We note $\{\ \}$ the empty substitution, and consider it as the canonical true value. We note $\cup$ the join operation on disjoint substitutions that returns $\bot$ as soon as one of the substitution is $\bot$. Note that conjunction pointcuts are defined only on substitutions that are disjoint, but because variables occur linearly in pointcuts, we have the guarantee that this is always the case. The matching relation is defined by induction on the structure of the pointcut in Figure 7.

In the rule for the $\mathsf{contains}(x.M)$ pointcut, the inclusion of patterns $M\tau \subseteq M'$ is defined as the inclusion of the induced multiset of messages. For instance, suppose that the cache replication aspect defined previously has been deployed and that the emitted join point is:

$$(\varphi, x.put(\text{“}\texttt{bar}\text{”}, 5)\ \&\ state(\mathsf{d})), \bar{J}$$

Then, the pointcut of the aspect:

$$\mathsf{contains}(*.put(k,v)) \wedge \mathsf{host}(h)$$

$$
\begin{array}{lcl}
\{0\}_J & = & 0 \\
\{x.M\}_J & = & x.\{M\}_J \\
\{\mathsf{obj}\ x = D\ \mathsf{in}\ P\}_J & = & \mathsf{obj}\ x = D\ \mathsf{in}\{P\}_J \\
\{\mathsf{go}(H); P\}_J & = & \mathsf{go}(H); \{P\}_J \\
\{H[P]\}_J & = & H[\{P\}_J] \\
\{P\ \&\ P'\}_J & = & \{P\}_J\ \&\ \{P'\}_J \\
\{l(\bar{v})\}_J & = & l_J(\bar{v}) \\
\{M\ \&\ M'\}_J & = & \{M\}_J\ \&\ \{M'\}_J
\end{array}
$$

Fig. 8 Tagging of causal history.

matches, with partial bijection:

$$\tau = \{k \mapsto \text{“}\mathtt{bar}\text{”}, v \mapsto 5, h \mapsto \varphi\}$$

Note that the variable d is not mapped by τ because it is not captured by the pointcut.

The rule for the $\mathsf{host}(h)$ pointcut always returns the substitution that associates h with the location of the matched pattern. The semantics of the negation and conjunction is an extension of the traditional boolean semantics to truth values that are substitutions.

The rule for the $\mathsf{causedBy}(Pc)$ pointcut returns the substitution that matches Pc for any sub join point J' of J, that is any join point in the causal history of J. It returns $\perp$ when no join point matches Pc.

Remembering causality in processes. In order to conserve and propagate the causal history during the reduction, each message $l(\bar{v})$ is tagged with the join point J that causes it, noted $l_J(\bar{v})$. Given a pattern M that is matched during the reduction, we note $\{\!\{M\}\!\}_{\bar{J}}$ the pattern tagged with the causal history of each message present in the pattern (note that M and $\bar{J}$ have to be of the same size) as defined by:

- $\{\!\{l(\bar{v})\}\!\}_{[J]} = l_J(\bar{v})$
- $\{\!\{M\ \&\ M'\}\!\}_{\bar{J} +\!\!+ \bar{J}'} = \{\!\{M\}\!\}_{\bar{J}}\ \&\ \{\!\{M'\}\!\}_{\bar{J}'}$

where $\bar{J} +\!\!+ \bar{J}'$ denotes list concatenation, and $[J]$ is a singleton list containing J.

Figure 8 presents tagging for processes that are produced by a reduction. Here, the idea is to tag each message that has been produced by a reduction. Initially, all messages have an empty history, so we take l as syntactic sugar for $l_\bullet$.

For instance, consider the reduction from Section 2.4, augmented with causality tagging (but without considering aspects, and omitting location φ):

$$
\begin{array}{ll}
 & c.state_\bullet(d_0)\ \&\ c.get_\bullet(\text{“}\mathtt{foo}\text{”}, r)\ \&\ c.put_\bullet(\text{“}\mathtt{foo}\text{”}, 5) \\
\longrightarrow & dict.update_{J_1}(d_0, \text{“}\mathtt{foo}\text{”}, 5, c)\ \&\ c.get_\bullet(\text{“}\mathtt{foo}\text{”}, r) \\
\longrightarrow & c.getDict_{J_2}(d_1)\ \&\ c.get_\bullet(\text{“}\mathtt{foo}\text{”}, r) \\
\longrightarrow & c.state_{J_3}(d_1)\ \&\ c.get_\bullet(\text{“}\mathtt{foo}\text{”}, r) \\
\longrightarrow & c.state_{J_4}(d_1)\ \&\ dict.lookup_{J_4}(d_1, \text{“}\mathtt{foo}\text{”}, r) \\
\longrightarrow & c.state_{J_4}(d_1)\ \&\ r.reply_{J_5}(5)
\end{array}
$$

where

$$
\begin{array}{l}
J_1 = ((\varphi, c.state(d_0)\ \&\ c.put(\text{“}\mathtt{foo}\text{”}, 5)), [\,]) \\
J_2 = ((\varphi, dict.update(d_0, \text{“}\mathtt{foo}\text{”}, 5, c)), [J_1]) \\
J_3 = ((\varphi, c.getDict(d_1)), [J_2]) \\
J_4 = ((\varphi, c.state(d_1)\ \&\ c.get(\text{“}\mathtt{foo}\text{”}, r)), [J_3]) \\
J_5 = ((\varphi, dict.lookup(d_1, \text{“}\mathtt{foo}\text{”}, r)), [J_4])
\end{array}
$$

DEPLOY
$x.[\langle Pc, Ad\rangle] \Vdash^{\varphi} \multimap x.\langle Pc, Ad\rangle^{\circ} \Vdash^{\varphi}$

RED/NOASP
$x.[M \rhd P] \Vdash^{\varphi} x.\{\!\{M\sigma\}\!\}_{\bar{J}} \multimap x.[M \rhd P] \Vdash^{\varphi} \{P\sigma\}_{J'}$
when no pointcut of an activated aspect matches J'.

RED/ASP
$x.[M \rhd P] \Vdash^{\varphi} x.\{\!\{M\sigma\}\!\}_{\bar{J}} \ \|_{i\in I}\ x_i.\langle Pc_i, Ad_i\rangle^{\circ} \Vdash^{\psi_i} \multimap$
$x.[M \rhd P] \Vdash^{\varphi} \ \|_{i\in I}$
$x_i.\langle Pc_i, Ad_i\rangle^{\circ} \Vdash^{\psi_i} \{Ad_i[P\sigma/\,\mathsf{proceed}]\tau_i\}_{J'}$
where $J' \Vdash Pc_i = \tau_i$ for all $i \in I$
and no other activated aspect matches J'.

RED/OPAQUE
$x.[M \blacktriangleright P] \Vdash^{\varphi} x.\{\!\{M\sigma\}\!\}_{\bar{J}} \multimap x.[M \blacktriangleright P] \Vdash^{\varphi} \{P\sigma\}_{J'}$

RED/OBSERVABLE
$x.[M \circledcirc P] \Vdash^{\varphi} x.\{\!\{M\sigma\}\!\}_{\bar{J}} \ \|_{i\in I}\ x_i.\langle Pc_i, Ad_i\rangle^{\circ} \Vdash^{\psi_i} \multimap$
$x.[M \circledcirc P] \Vdash^{\varphi} \{P\sigma\}_{J'} \ \|_{i\in I}$
$x_i.\langle Pc_i, Ad_i\rangle^{\circ} \Vdash^{\psi_i} \{Ad_i\tau_i\}_{J'}$
where no Ad_i contains $\mathsf{proceed}$,
$J' \Vdash Pc_i = \tau_i$ for all $i \in I$
and no other activated aspect matches J'.

In every reduction rule, J' stands for $(\varphi, x.M\sigma), \bar{J}$.

Fig. 9 Semantics of aspect weaving

Semantics of aspect weaving. Figure 9 presents the semantics of aspects. All rules of Figure 3 are preserved, except for Rule RED because this is where weaving takes place. This rule is split into four rules, all of which depend on currently activated aspects as expressed by the following rule. We use the notation $\multimap$ to distinguish clearly between a reduction that occurs in the aspect join calculus and in the join calculus.

Rule DEPLOY corresponds to the asynchronous deployment of a pointcut/advice pair $x.\langle Pc, Adv\rangle$ by marking the pair as activated $x.\langle Pc, Adv\rangle^{\circ}$. Note that activated pairs are not directly user-definable. The presence of this rule is crucial in the semantics because it allows activating aspects one by one asynchronously. Another possible semantics would have been to deploy synchronously altogether pointcut/advice pairs of the same definition, but then it would have caused extra

synchronization in the translation to the core join calculus, and hence also in our implementation.

Rule Red/NoAsp is a direct reminiscence of Rule Red in case where no activated pointcut matches. Note that the new causal history is propagated to the produced process $P\sigma$.

Rule Red/Asp defines the modification of Rule Red in presence of aspects. If there is an aspect x_i with an activated pointcut/advice pair $x_i.\langle Pc, Ad\rangle^\circ$ such that Pc matches the join point with substitution τ, then the advice Ad is executed with the process P substituting the keyword proceed and where the variables bound by the pointcut are substituted according to τ. The side condition of Rule Red/Asp is that all Pc_is are the activated 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 $x_i = x_j$ and $\psi_i = \psi_j$. Note that all advices associated with a pointcut that matches are executed in parallel.

Rule Red/Opaque is computationally the same as Rule Red/NoAsp, since activated aspects are essentially ignored when an opaque reaction occurs.

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 cannot use proceed. This restriction could be guaranteed by a simple type system.

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

$$\begin{array}{ll} & x.put_{J_1}(\text{“bar”}, 5)\ \&\ state_{J_2}(\mathsf{d}) \\ \multimap & c.put_{J'}(\text{“bar”}, 5)\ \&\ dict.update_{J'}(\mathsf{d}, \text{“bar”}, 5, x) \\ & \text{where } J' = ((\varphi, x.put(\text{“bar”}, 5)), [J_1, J_2]) \end{array}$$

The original operation on *dict* to update d is performed, in addition to the replication on c.

3.4 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 [22]):[8]

```
def make_buffer(k) ▷
  def  put(n) | empty() ▷ some(n)
   ∧   get(r) | some(n) ▷ r(n) | empty()
in empty() | k(get, put)
```

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 [22]). 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 (*e.g.* 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 [23]. 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 [23] in our presentation of the aspect join calculus. Classes support extensible definitions, but do not contribute anything essential with respect to naming and quantification.

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 prove it correct via a bisimilarity argument. The translation is used in Section 6 to implement Aspect JoCaml on top of JoCaml [24], an implementation of the join calculus in OCaml.

4.1 General approach

The translation approach consists in considering that an aspect is a standard object that receives messages

[8] Syntactically, the main differences with our calculus are that conjunction is noted | instead of & and disjunction of rules is noted ∧ instead of or. Also, there are no objects with labels, only channels.

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 [27].

Aspect weaving. In order to determine whether an aspect applies or not, the translation must account for aspect weaving. Note that the description of the semantics of the aspect join calculus leaves open the question of the underlying aspect weaving infrastructure. The naive approach, described in [53], consists in relying on a central weaver that coordinates all distributed computations and triggers the weaving of all aspects. This centralized approach is however not realistic in a distributed setting.

Decentralized weaving. We adopt a decentralized weaving architecture, in which essentially each reaction is in charge of its own weaving, that is, determining which aspects apply to it and subsequently triggering their execution. In other words, with each reaction rule is associated a local, dedicated weaver. Recall that each reaction can have a specific weaving semantics (Section 3.2), hence there are correspondingly different kinds of weavers.

In order to support dynamic deployment of aspects, weavers consult a central registry that holds the list of currently-deployed aspects. Similarly, all aspect definitions register aspects with this registry. In Section 5.3, we discuss the possibility of distributing the aspect registry as well, introducing different policies of aspect deployment.

More specifically, the key interactions for aspect deployment and weaving are as follows:

- For each reaction rule $M \triangleright P$ in object x, there is a local weaver $W_{x.M}$, on the same host, that can receive *weave* messages. These *weave* messages are sent each time the reaction rule fires.
- The aspect registry R is executing at location H_w and is known by all other processes. It exposes the following definitions D_R:

$$D_R = get_{asp}(k) \,\&\, asp_{act}(\bar{a}) \triangleright k(\bar{a}) \,\&\, asp_{act}(\bar{a})$$
$$deploy(a) \,\&\, asp_{act}(\bar{a}) \triangleright asp_{act}(a, \bar{a})$$

- Aspects get activated by registering to the aspect registry through the label *deploy*.
- Upon each (advisable or observable) reaction that fires, local weavers get the current list of activated aspects from the aspect registry by passing a continuation to the label get_{asp}.

Note that to prove correctness of the translation, it is important that local weavers ask for the current list of aspects *before* weaving a reaction because it guarantees the consistency of the knowledge of the list of activated aspects between local weavers—indeed, the reference semantics of the distributed aspect join (Figure 9) assumes a globally consistent view of the list of activated aspects.

Processes

$$\begin{aligned}
[\![0]\!]_J &:= 0\\
[\![x.M]\!]_J &:= x.[\![M]\!]_J\\
[\![\mathsf{obj}\ x = D\ \mathsf{in}\ P]\!]_J &:= \mathsf{obj}\ \mathcal{W}(D_r, x)\ \mathsf{in}\\
&\quad\ \mathsf{obj}\ x = [\![D_r]\!]_x\ \mathsf{in}\ [\![D_a]\!] \,\&\, [\![P]\!]_J\\
[\![\mathsf{go}(H); P]\!]_J &:= \mathsf{go}(H); [\![P]\!]_J\\
[\![H[P]]\!]_J &:= H[[\![P]\!]_J]\\
[\![P \,\&\, P']\!]_J &:= [\![P]\!]_J \,\&\, [\![P']\!]_J\\
[\![P, P']\!]_J &:= [\![P]\!]_J, [\![P']\!]_J
\end{aligned}$$

Definitions

$$\begin{aligned}
[\![M \triangleright P]\!]_x &:= [\![M]\!]_{\bar{J}_M} \triangleright \mathsf{obj}\ ret = proceed(J) \triangleright [\![P]\!]_J\\
&\quad\ \mathsf{in}\ \mathsf{let}\ J_W = (\mathtt{lhost}, x.M) + \bar{J}_M\\
&\quad\ \ \mathsf{in}\ W_{x.M}.weave(ret.proceed, J_W)\\
[\![M \circledcirc P]\!]_x &:= [\![M \triangleright P]\!]_x\\
[\![M \blacktriangleright P]\!]_x &:= [\![M]\!]_{\bar{J}_M} \triangleright \mathsf{let}\ J = (\mathtt{lhost}, x.M) + \bar{J}_M\\
&\quad\ \mathsf{in}\ [\![P]\!]_J\\
[\![D\ \mathsf{or}\ D']\!]_x &:= [\![D]\!]_x\ \mathsf{or}\ [\![D']\!]_x
\end{aligned}$$

Messages

$$\begin{aligned}
[\![l_J(\bar{v})]\!]_\bullet &:= l(\bar{v}, J)\\
[\![M \,\&\, M']\!]_\bullet &:= [\![M]\!]_\bullet \,\&\, [\![M']\!]_\bullet\\
[\![l(\bar{v})]\!]_J &:= l(\bar{v}, J)\\
[\![l(\bar{v}) \,\&\, M']\!]_{J,\bar{J}} &:= [\![l(\bar{v})]\!]_J \,\&\, [\![M']\!]_{\bar{J}}
\end{aligned}$$

Pointcuts

$$\begin{aligned}
[\![\mathsf{contains}(x.M)]\!] &:= \lambda(\varphi, x'.M'), \bar{J}.\\
&\quad\ \mathsf{if}\ M\tau \subseteq M'\ \mathsf{then}\ [[x \mapsto x', \tau]]\ \mathsf{else}\ []\\
[\![\mathsf{host}(H)]\!] &:= \lambda(\varphi, x'.M'), \bar{J}.\ [[H \mapsto \varphi]]\\
[\![\neg Pc]\!] &:= \lambda J.\ \mathsf{if}\ [\![Pc]\!]\ J = []\ \mathsf{then}\ [[]]\ \mathsf{else}\ []\\
[\![Pc \wedge Pc']\!] &:= \lambda J.\ \mathsf{product}([\![Pc]\!]\ J, [\![Pc']\!]\ J)\\
[\![\mathsf{causedBy}(Pc)]\!] &:= \lambda jp, \bar{J}.\ \mathsf{concat}(\mathsf{map}\ [\![Pc]\!]_{rec}\ \bar{J})\\
[\![Pc]\!]_{rec} &:= \lambda J.\ \mathtt{match}\ J\ \mathtt{with}\\
&\quad\ \mid \bullet \quad\ \Rightarrow []\\
&\quad\ \mid jp, \bar{J} \Rightarrow [\![Pc]\!]\ (jp, \bar{J}) +\!\!+\\
&\quad\ \qquad\ \mathsf{concat}(\mathsf{map}\ [\![Pc]\!]_{rec}\ \bar{J})
\end{aligned}$$

Aspects

$$\begin{aligned}
[\![\mathsf{proceed}]\!]_J &:= proceed(J)\\
[\![\langle Pc, Ad\rangle]\!]_x &:= \mathsf{obj}\ adv = advice(proceed, J, \bar{v}_{Pc}) \triangleright\\
&\quad\ [\![Ad]\!]_J\\
&\quad\ \mathsf{in}\ R.deploy([\![Pc]\!], adv.advice)
\end{aligned}$$

where $\bar{v}_{Pc}$ is the list of variables occurring free in Pc.

Fig. 10 Translating the aspect join calculus to the join calculus (translation of named definitions is in Figure 12)

4.2 Translation

The general idea of the translation is that, given an aspect join calculus configuration:

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

we construct a distributed join calculus configuration without aspects by translating definitions, processes and aspects, as defined in Figure 10, and introducing the aspect registry R on host H_w, yielding the following configuration:

$$\begin{array}{c}[\![\mathcal{C}]\!] = [\![\mathcal{D}_1]\!] \Vdash^{\varphi_1} [\![\mathcal{P}_1]\!]_\bullet \quad \| \cdots \| \quad [\![\mathcal{D}_n]\!] \Vdash^{\varphi_n} [\![\mathcal{P}_n]\!]_\bullet \\ \| \quad R.D_R \Vdash^{H_w} R.asp_{act}([\![\bar{a}]\!]_{act})\end{array}$$

where $\bar{a}$ is the list of activated aspects in $\mathcal{D}_1, \ldots, \mathcal{D}_n$.

The translation of the list of activated aspects of $R.asp_{act}$ is given recursively by

$$[\![x.\langle Pc, Ad\rangle^\circ\,, \bar{a}]\!]_{act} = ([\![Pc]\!], x.advice)\,, [\![\bar{a}]\!]_{act}.$$

That is, the translation of a pointcut/advice pair is given by the pair of the translated pointcut and the label on which the advice can be called.

The rest of the translation is given as follows.

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. The translation of reaction rules is done in two steps. First, a local weaver is created for each reaction, using $\mathcal{W}(D_r, x)$, and then each reaction is replaced by a reaction that communicates with its weaver. We describe weavers in details later on in this section.

Definitions. The central point of the translation is to replace a standard reaction rule by a rule that reifies the reaction through an explicit join point, and then triggers a protocol with the weaver to decide whether or not some aspect intercepts the reaction rule and must be executed. Specifically, the translation of a reaction rule $M \triangleright P$ in object x, denoted $[\![M \triangleright P]\!]_x$, produces a call to the weaver $W_{x.M}.weave$, passing a locally-created single-use label *ret.proceed* to perform the original computation P and the current join point J_W, obtained by collecting join points of the matched pattern $\bar{J}_M$ and adding the current reaction join point $(\texttt{lhost}, x.M)$. Here, $\bar{J}_M$ is a list of (local) variables (of size equal to the number of messages in the pattern M) to be bound to the actual causal history (see the translation of messages explained below). It is important that the new label is locally-created and guarantees a single use because it ensures that different calls to proceed cannot be interleaved.

Observable reactions are translated similarly, since the difference in semantics is encapsulated in the weaver itself. Opaque reactions are not woven.

Messages. There are two ways to translate messages: (1) Case $[\![-]\!]_\bullet$ corresponds to messages that occur on the right hand side of a definition. This means that the messages are already tagged with the join point J that causes them. The tagged join point J is simply converted to an argument of the (untagged) message. (2) Case $[\![-]\!]_{\bar{J}}$ corresponds to messages that occur on the left hand side of a definition. In that case, the index $\bar{J}$ of the translation corresponds to the list of variables that are used to bind the causal history. There is the implicit hypothesis that the size of $\bar{J}$ is equal to the number of messages. Each variable J is added as an argument of the corresponding message.

Pointcuts. Pointcuts are recursively transformed into functions that operate on join points and return a list of substitutions (encoded as lists) that correspond to all the possible matches. When the list is empty, it means that the pointcut does not match. This is the usual folklore way of computing altogether the possible results of a non deterministic function in one step [60]. To preserve the non-deterministic nature of pointcut matching, only one substitution will be chosen randomly by the weaver.

In the definition of $[\![\mathsf{contains}(x.M)]\!]$, the substitution τ is seen as the list, and contains only free variables of M. The definition of $[\![Pc \wedge Pc']\!]$ uses the function $\mathsf{product}$ that takes two lists of lists and returns the list of lists of all possible concatenations, *i.e.*:

$$\mathsf{product}(ls, ls') = [\, l +\!\!+ l' \mid l \in ls,\ l' \in ls' \,].$$

The definition of $[\![\mathsf{causedBy}(Pc)]\!]$ is given by computing every possible match of Pc on every sub join point. concat and map are the usual operations on lists.

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 advice has only one label *advice* which expects the proceed label, the current join point and the list of variables that are free in the pointcut Pc.

The initialization sends the pointcut/advice pair to the weaving registry R 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.

Finally, the translation of $\mathsf{proceed}$ is obtained by adding the current join point J as argument to *proceed*.

$$
\begin{array}{l}
\mathcal{W}(M \triangleright P, x) := W_{x.M} = D_{\triangleright} \\
\mathcal{W}(M \circledcirc P, x) := W_{x.M} = D_{\circledcirc} \\
\mathcal{W}(M \blacktriangleright P, x) := W_{x.M} = 0 \\
\mathcal{W}(D \,\mathsf{or}\, D', x) := \mathcal{W}(D, x) \;\mathsf{in}\; \mathsf{obj}\; \mathcal{W}(D', x)
\end{array}
$$

$$
\begin{array}{l}
D_{\circledcirc} := \; weave(proceed, J) \triangleright \\
\quad \mathsf{obj}\; W_{init} = aspL(asps) \triangleright \\
\quad\quad \mathsf{let}\; ads = \mathsf{filter}\; (\lambda(\bar{\bar{v}}, \_).\bar{\bar{v}} \neq [\,]) \\
\quad\quad\quad\quad\quad\quad \mathsf{map}\; (\lambda(Pc, adv).([\![Pc]\!]\, J, adv))\; asps \\
\quad\quad \mathsf{in}\, proceed(J) \& \\
\quad\quad\quad \mathsf{iter}\; (\lambda(\bar{\bar{v}}, adv).\; adv(0, J, \mathsf{select}(\bar{\bar{v}})))\; ads \\
\quad \mathsf{in}\, R.get_{asp}(W_{init}.aspL)
\end{array}
$$

$$
\begin{array}{l}
D_{\triangleright} := \; weave(proceed, J) \triangleright \\
\quad \mathsf{obj}\; W_{init} = aspL(asps) \triangleright \\
\quad\quad \mathsf{let}\; ads = \mathsf{filter}\; (\lambda(\bar{\bar{v}}, \_).\bar{\bar{v}} \neq [\,]) \\
\quad\quad\quad\quad\quad\quad \mathsf{map}\; (\lambda(Pc, adv).([\![Pc]\!]\, J, adv))\; asps \\
\quad\quad \mathsf{in\; if}\; ads = [\,] \\
\quad\quad\quad \mathsf{then}\; proceed(J) \\
\quad\quad\quad \mathsf{else\; iter}\; (\lambda(\bar{\bar{v}}, adv).\; adv(proceed, J, \mathsf{select}(\bar{\bar{v}})))\; ads \\
\quad \mathsf{in}\, R.get_{asp}(W_{init}.aspL)
\end{array}
$$

Fig. 11 Per-reaction weaving

Per-reaction weavers. The per-reactions weavers are defined altogether at the beginning of the translation of an object using the inductive definition $\mathcal{W}(D, x)$, given in Figure 11. As explained above, for each reaction, a weaver is defined as an object with a *weave* method, used to trigger weaving at a join point.

The definition of the weaver depends on the kind of reaction. In the opaque case, the weaver is the null process. In both the observable ($D_{\circledcirc}$) and advisable ($D_{\triangleright}$) cases, when the weaver receives $weave(proceed, J)$, it creates a new object W_{init} that defines a fresh channel $aspL()$ whose aim is to get the current list of activated aspects $asps$ from the aspect registry by spawning $get_{asp}(W_{init}.aspL)$. When the list is received, the weaver filters the advices that match the current join point, ads (using the usual filter functions on lists). It then triggers all matching advices (using the iter function on lists) *by selecting randomly* a substitution from the list of substitutions $\bar{\bar{v}}$, using the non-deterministic selection function select.

For observable reactions, the weaver $D_{\circledcirc}$ just spawns in parallel a call to proceed with all the advices, corresponding to Rule Red/Obs. For advisable reactions, the weaver $D_{\triangleright}$ needs to distinguish between two cases. If no aspect applies, the weaver executes the original process by sending the message $proceed(J)$; this corresponds to Rule Red/NoAsp. Otherwise, the weaver only executes all advices in ads, without calling proceed (Rule Red/Asp).

$$
\begin{array}{lcl}
[\![x.[M \triangleright P]]\!] & := & x.[[\![M \triangleright P]\!]_x] \,\mathsf{or}\, W_{x.M}.D_{\triangleright} \Vdash^{\varphi} \\
[\![x.[M \circledcirc P]]\!] & := & x.[[\![M \circledcirc P]\!]_x] \,\mathsf{or}\, W_{x.M}.D_{\circledcirc} \Vdash^{\varphi} \\
[\![x.[M \blacktriangleright P]]\!] & := & x.[[\![M \blacktriangleright P]\!]_x] \Vdash^{\varphi} \\
[\![x.\langle Pc, Ad\rangle^{\circ}]\!] & := & x.[advice(proceed, J, \bar{v}_{Pc}) \triangleright [\![Ad]\!]_J] \Vdash^{\varphi} \\
[\![x.\langle Pc, Ad\rangle]\!] & := & x.[advice(proceed, J, \bar{v}_{Pc}) \triangleright [\![Ad]\!]_J] \Vdash^{\varphi} \\
 & & \quad R.deploy([\![Pc]\!], x.advice) \\
[\![\mathcal{D} \,\mathsf{or}\, \mathcal{D}']\!] & := & [\![\mathcal{D}]\!] \,\mathsf{or}\, [\![\mathcal{D}']\!] \Vdash^{\varphi} \\
[\![\mathcal{D}, \mathcal{D}']\!] & := & [\![\mathcal{D}]\!],\, [\![\mathcal{D}']\!] \Vdash^{\varphi} \\
[\![H[\mathcal{D}\colon P]]\!] & := & H[[\![\mathcal{D}]\!]\colon [\![P]\!]_{\bullet}] \Vdash^{\varphi} \\
[\![\top]\!] & := & \top \Vdash^{\varphi}
\end{array}
$$

where $\bar{v}_{Pc}$ is the list of variables occurring free in Pc.

Fig. 12 Translation of named definitions at location φ

Named Definitions. Each observable and advisable reactions introduce their weavers in addition to the named translation of the reaction itself (Figure 12). Opaque reactions do not introduce any weaver, since they cannot be advised.

The translation of an activated aspect is simply the translation of its advice, because the part dealing with its pointcut has been delegated to the weaver. When the pointcut/advice pair is not already activated, the translation must place the message $R.deploy([\![Pc]\!], x.advice)$ in the solution for future consumption by the aspect registry (hence emulating Rule Deploy from Figure 9).

For the other kinds of named definitions, the translation is just applied recursively.

4.3 Correctness of the 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. The first thing to check for the correctness of the translation is that it preserves structural rules.

Lemma 1 *Structural rules are preserved by the translation, that is if* $\mathcal{C} \equiv \mathcal{C}'$ *then* $[\![\mathcal{C}]\!] \equiv [\![\mathcal{C}']\!]$.

Proof By case analysis on the structural rule:

- Or:
  $[\![\mathcal{D} \,\mathsf{or}\, \mathcal{D}']\!] \Vdash^{\varphi} \equiv [\![\mathcal{D}]\!] \,\mathsf{or} [\![\mathcal{D}']\!] \Vdash^{\varphi} \equiv [\![\mathcal{D}]\!], [\![\mathcal{D}']\!] \Vdash^{\varphi} \equiv [\![\mathcal{D}, \mathcal{D}']\!] \Vdash^{\varphi}$
- Empty:
  $[\![\top]\!] \Vdash^{\varphi} \equiv \top \Vdash^{\varphi} \equiv \Vdash^{\varphi}$
- Par:
  $\Vdash^{\varphi} [\![P \,\&\, Q]\!]_{\bullet} \equiv \Vdash^{\varphi} [\![P]\!]_{\bullet} \,\&\, [\![Q]\!]_{\bullet} \equiv \Vdash^{\varphi} [\![P]\!]_{\bullet}, [\![Q]\!]_{\bullet} \equiv \Vdash^{\varphi} [\![P, Q]\!]_{\bullet}$
- Nil:
  $\Vdash^{\varphi} [\![0]\!]_{\bullet} \equiv \Vdash^{\varphi} 0 \equiv \Vdash^{\varphi}$
- Join:
  $\Vdash^{\varphi} [\![x.(M \,\&\, M')]\!]_{\bullet} \equiv \Vdash^{\varphi} x.([\![M]\!]_{\bullet} \,\&\, [\![M']\!]_{\bullet}) \equiv$
  $\Vdash^{\varphi} [\![x.M]\!]_{\bullet} \,\&\, [\![x.M']\!]_{\bullet} \equiv \Vdash^{\varphi} [\![x.M \,\&\, x.M']\!]_{\bullet}$
- Sub-Loc:
  $[\![H[\mathcal{D}\colon P]]\!] \Vdash^{\varphi} \equiv H[[\![\mathcal{D}]\!]\colon [\![P]\!]_{\bullet}] \Vdash^{\varphi} \equiv \{[\![\mathcal{D}]\!]\} \Vdash^{\varphi H} \{[\![P]\!]_{\bullet}\}$
- Obj-Def:
  To simplify, we consider the case of one definition and one aspect (the general case follows by induction).

$\Vdash^{\varphi} [\![\mathsf{obj}\ x = (M \rhd P\ \mathsf{or}\langle Pc, Ad\rangle)\ \mathsf{in}\ Q]\!]_\bullet \equiv$
$\Vdash^{\varphi}\ \mathsf{obj}\ \mathcal{W}(M \rhd P, x)\ \mathsf{in obj}\ x = [\![M \rhd P]\!]_x\ \mathsf{in}\ [\![\langle Pc, Ad\rangle]\!]\ \&$
$[\![Q]\!]_\bullet \equiv$
$W_{x.M}.D_\rhd\ \mathsf{or}\ x.[\![M \rhd P]\!]_x\ \mathsf{or}\ adv.advice(proceed, J, \bar{v}_{Pc}) \rhd [\![Ad]\!]_J$
$\Vdash^{\varphi} R.deploy([\![Pc]\!], adv.advice)\ \&\ [\![Q]\!]_\bullet \equiv$
$[\![x.M \rhd P]\!]\ \mathsf{or} [\![adv.\langle Pc, Ad\rangle]\!] \Vdash^{\varphi} [\![Q]\!]_\bullet \equiv$
$[\![x.(M \rhd P\ \mathsf{or}\langle Pc, Ad\rangle)]\!] \Vdash^{\varphi} [\![Q]\!]_\bullet$

– Loc-Def:
$\Vdash^{\varphi} [\![H[P]]\!]_\bullet \equiv \Vdash^{\varphi} H[[\![P]\!]_\bullet] \equiv H[\top : [\![P]\!]_\bullet] \Vdash^{\varphi} \equiv [\![H[\top : P]]\!] \Vdash^{\varphi}$

It is also necessary to check that the translation of a pointcut is a function that computes all the possible substitutions returned by the (non-deterministic) semantics of this pointcut.

Lemma 2 *For every pointcut Pc and join point J, we have $[\![Pc]\!]\ J = [\tau \mid J \Vdash Pc : \tau$ and $\tau \neq \bot]$. In particular, when a pointcut does not match $[\![Pc]\!]\ J$ returns the empty list.*

Proof By structural induction on Pc:

– $\mathsf{contains}(x.M)$: This case is just a unification problem. $[\![\mathsf{contains}(x.M)]\!]$ returns at most one substitution.
– $\mathsf{host}(h)$: $[\![\mathsf{host}(h)]\!]$ returns the singleton list containing the substitution that sends h to the current location of the join point.
– $Pc \wedge Pc'$: By induction, $[\![Pc]\!]$ and $[\![Pc']\!]$ compute the lists of all possible substitutions. The definition of $[\![Pc \wedge Pc']\!]\ J$ thus returns the list of all possible disjoint union (computed as list concatenation) of two substitutions picked up in $[\![Pc]\!]\ J$ and $[\![Pc']\!]\ J$.
– $\neg Pc$: By induction, $[\![Pc]\!]\ J$ is the empty list if and only if $J \Vdash Pc = \bot$.
– $\mathsf{causedBy}(Pc)$: By induction, $[\![Pc]\!]\ J'$ computes the list of all possible substitutions for any pointcut J'. Using this, it is easy to prove by induction on J that

$$[\![Pc]\!]_{rec}\ J = [\tau \mid J' \Vdash Pc : \tau \text{ such that } \tau \neq \bot \text{ and } (J' \prec J \text{ or } J' = J)].$$

It then follows that

$$[\![\mathsf{causedBy}(Pc)]\!]\ J = [\tau \mid J' \Vdash Pc : \tau \text{ such that } \tau \neq \bot \text{ and } J' \prec J].$$

As usual in concurrent programming languages, the correctness of the translation algorithm is given by a proof of bisimilarity. Namely, we prove that the original configuration with aspects (in the aspect join calculus) is bisimilar to the translated configuration without aspects (in the objective join calculus). 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:

$$\begin{array}{ccc} \mathcal{C}_0 & \xrightarrow{\ \mathcal{R}\ } & \mathcal{C}_1 \\ \downarrow & & \vdots \\ \mathcal{C}'_0 & \xrightarrow{\ \mathcal{R}\ } & \mathcal{C}'_1 \end{array} \qquad \vdots \text{ is } {}^*$$

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

With this definition, our notion of bisimulation 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*). However, to prevent the trivial translation that sends every process to the null process 0 to be a correct translation, we need to be able to observe at least one reduction. To this end, we consider a special object name `output`, with a message `result` on which to write the output and only one reaction involving $\texttt{output.result}(\bar{v}) \rhd 0$. This is essentially the channel on which one can observe the computational behavior of a configuration. In what follows, we assume that the definition of this object is always present in the configuration at location $H_{\texttt{output}}$. To express that the reaction on `output` is observable, we additionally require that a simulation $\mathcal{R}$ satisfies that any two configurations $(\mathcal{C}_0, \mathcal{C}_1) \in \mathcal{R}$ have the same number of messages `output.result` in the solution.

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, weavers, aspects and the aspect registry.
2. In the execution of $\mathcal{C}$, $\mathsf{proceed}$ 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 rules of the join calculus. Namely, we take Rule Red and Message-Comm restricted to the case where the pattern contains either of the dedicated labels: *weave*, *proceed*, *advice*, get_{asp} and *aspL* (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 asp_{act}. 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. However, the reduction is non-deterministic due to the presence of the non-deterministic function select in the definition of weavers. Therefore, it makes sense to talk about the normal forms of $\mathcal{C} \in \mathbb{T}$, whose set is noted $\widetilde{\mathcal{C}}$.

We note $\mathcal{C} \overset{\text{gc}}{\sim} \mathcal{C}'$ when $\mathcal{C}'$ is equal to $\mathcal{C}$ where every named definition $x.D$ for which x does not appear in the configuration (but in $x.D$ of course) is removed from the configuration. This condition deals with auxiliary definitions appearing during the translation that are used linearly and must then be garbage collected. This is mandatory to synchronize the current auxiliary definitions available in the solution.

Theorem 1 *The relation*

$$\mathcal{R} = \{(\mathcal{C}_0, \mathcal{C}_1) \mid \exists \mathcal{C}'_1 \in \widetilde{\mathcal{C}_1}.\ [\![\mathcal{C}_0]\!] \overset{\text{gc}}{\sim} \mathcal{C}'_1\}$$

is a bisimulation. In particular, any configuration is bisimilar to its translation.

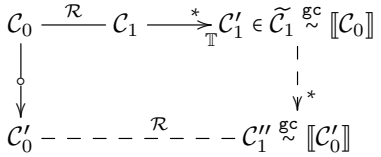
Proof The fact that any two configuration in $\mathcal{R}$ have the same number of messages `output.result` in the solution is direct as the translation preserves messages and does not introduce any message of this form.

(A) $\mathcal{R}$ is a simulation.

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.

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 $\mathcal{C}$.

More precisely, we show that for any reduction $\mathcal{C}_0 \longrightarrow \mathcal{C}'_0$, one can find a corresponding reduction chains from $[\![\mathcal{C}_0]\!]$ to $[\![\mathcal{C}'_0]\!]$:



Rule DEPLOY**.**

The activation of an aspect $x.\langle Pc, Ad\rangle$ is in one-to-one correspondence with the consumption of the message $R.deploy([\![Pc]\!], x.advice)$ by the aspect registry.

$$\begin{array}{ll}
[\![x.\langle Pc, Ad\rangle]\!] \Vdash^{\varphi} \ \| \ R.D_R \Vdash^{H_w} R.asp_{act}([\![\bar{a}]\!]_{act}) & \equiv \\
[\![x.\langle Pc, Ad\rangle^{\circ}]\!] \Vdash^{\varphi} R.deploy([\![Pc]\!], x.advice) \ \| & \\
R.D_R \Vdash^{H_w} R.asp_{act}([\![\bar{a}]\!]_{act}) & \longrightarrow \\
[\![x.\langle Pc, Ad\rangle^{\circ}]\!] \Vdash^{\varphi} \ \| & \\
R.D_R \Vdash^{H_w} R.deploy([\![Pc]\!], x.advice), R.asp_{act}([\![\bar{a}]\!]_{act}) & \longrightarrow \\
[\![x.\langle Pc, Ad\rangle^{\circ}]\!] \Vdash^{\varphi} \ \| & \\
R.D_R \Vdash^{H_w} R.asp_{act}(([\![Pc]\!], x.advice), [\![\bar{a}]\!]_{act}) &
\end{array}$$

Note at this point, that the fact that aspect activation is asynchronously described by Rule DEPLOY in the semantics is crucial in the proof.

Rule RED/NOASP**.**

Consider the reduction

$$x.[M \triangleright P] \Vdash^{\varphi} x.\{\!\{M\sigma\}\!\}_{\bar{J}} \ \multimap \ x.[EM \triangleright P] \Vdash^{\varphi} \{P\sigma\}_{J'}$$

This rule is simulated by the chain:

$$\begin{array}{rl}
[\![x.[M \triangleright P]]\!] \Vdash^{\varphi} & [\![x.\{\!\{M\sigma\}\!\}_{\bar{J}}]\!]_{\bullet} \\
R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \quad \equiv \\
x.[\![[M \triangleright P]]\!]_x, W_{x.M}.D_{\triangleright} \Vdash^{\varphi} & x.[\![\{\!\{M\sigma\}\!\}_{\bar{J}}]\!]_{\bullet} \\
R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \quad \longrightarrow \\
x.[\![[M \triangleright P]]\!]_x, W_{x.M}.D_{\triangleright}, \Vdash^{\varphi} & W_{x.M}.weave(ret.proceed, J') \\
\ ret.proceed(J) \triangleright [\![P\sigma]\!]_J & \\
R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \longrightarrow_{\mathbb{T}} \\
\multicolumn{2}{c}{\text{where } J' = (\varphi, x.M\sigma), \bar{J}} \\
x.[\![[M \triangleright P]]\!]_x, W_{x.M}.D_{\triangleright}, & \\
\ ret.proceed(J) \triangleright [\![P\sigma]\!]_J, \Vdash^{\varphi} & R.get_{asp}(W_{init}.aspL) \\
\ W_{init}.aspL(asps) \triangleright \ldots & \\
R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \longrightarrow_{\mathbb{T}} \\
x.[\![[M \triangleright P]]\!]_x, W_{x.M}.D_{\triangleright}, & \\
\ ret.proceed(J) \triangleright [\![P\sigma]\!]_J, \Vdash^{\varphi} & \\
\ W_{init}.aspL(asps) \triangleright \ldots & \\
R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}), \\
& R.get_{asp}(W_{init}.aspL) \longrightarrow_{\mathbb{T}} \\
x.[\![[M \triangleright P]]\!]_x, W_{x.M}.D_{\triangleright}, & \\
\ ret.proceed(J) \triangleright [\![P\sigma]\!]_J, \Vdash^{\varphi} & \\
\ W_{init}.aspL(asps) \triangleright \ldots & \\
R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}), \\
& W_{init}.aspL([\![\bar{a}]\!]_{\bullet}) \longrightarrow_{\mathbb{T}} \\
x.[\![[M \triangleright P]]\!]_x, W_{x.M}.D_{\triangleright}, & \\
\ ret.proceed(J) \triangleright [\![P\sigma]\!]_J, \Vdash^{\varphi} & W_{init}.aspL([\![\bar{a}]\!]_{\bullet}) \\
\ W_{init}.aspL(asps) \triangleright \ldots & \\
R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \longrightarrow_{\mathbb{T}}^{\text{gc}} \\
x.[\![[M \triangleright P]]\!]_x, W_{x.M}.D_{\triangleright}, \Vdash^{\varphi} & ret.proceed(J') \\
\ ret.proceed(J) \triangleright [\![P\sigma]\!]_J & \\
R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \longrightarrow_{\mathbb{T}}^{\text{gc}} \\
x.[\![[M \triangleright P]]\!]_x, W_{x.M}.D_{\triangleright} \Vdash^{\varphi} & [\![P\sigma]\!]_{J'} \\
R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \quad \equiv \\
[\![x.[M \triangleright P]]\!] \Vdash^{\varphi} & [\![\{P\sigma\}_{J'}]\!]_{\bullet} \\
R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act})
\end{array}$$

because by Lemma 2, $[\![Pc]\!]J' = [\,]$ for every pointcut Pc of activated aspects, so the result of filter in W_{init}

is empty. Here, the need for $\overset{\mathsf{gc}}{\sim}$ is for garbage collecting unused definitions such as $ret.proceed(J) \triangleright [\![P\sigma]\!]_J$.

Rule RED/ASP**.**

Consider the reduction

$$\begin{array}{l} x.[M \triangleright P] \Vdash^{\varphi} x.\{\!\{M\sigma\}\!\}_{\bar{J}} \quad \|_{i\in I} \quad x_i.\langle Pc_i, Ad_i\rangle^{\circ} \Vdash^{\psi_i} \quad \multimap \\ x.[M \triangleright P] \Vdash^{\varphi} \quad \|_{i\in I} \\ x_i.\langle Pc_i, Ad_i\rangle^{\circ} \Vdash^{\psi_i} \{Ad_i[P\sigma/\mathsf{proceed}]\tau_i\}_{J'} \end{array}$$

This rule is simulated by the chain:

$$\begin{array}{rl} [\![x.[M \triangleright P]]\!] \Vdash^{\varphi} & [\![x.\{\!\{M\sigma\}\!\}_{\bar{J}}]\!]_{\bullet} \\ R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \\ \|_{i\in I} \quad [\![x_i.\langle Pc_i, Ad_i\rangle^{\circ}]\!] \Vdash^{\psi_i} & \longrightarrow \end{array}$$

similar steps than for RED/NOASP:

$$\begin{array}{rl} x.[\![[M \triangleright P]]\!]_x, W_{x.M}.D_{\triangleright}, & \\ ret.proceed(J) \triangleright [\![P\sigma]\!]_J, \Vdash^{\varphi} & W_{init}.aspL([\![\bar{a}]\!]_{\bullet}) \\ W_{init}.aspL(asps) \triangleright \ldots & \\ R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \\ \|_{i\in I} \quad [\![x_i.\langle Pc_i, Ad_i\rangle^{\circ}]\!] \Vdash^{\psi_i} & \longrightarrow_{\mathbb{T}}^{\overset{\mathsf{gc}}{\sim}} \end{array}$$

$$\begin{array}{rl} x.[\![[M \triangleright P]]\!]_x, W_{x.M}.D_{\triangleright}, \Vdash^{\varphi} & \&_i x_i.advice(ret.proceed, J', \tau_i), \\ ret.proceed(J) \triangleright [\![P\sigma]\!]_J & \\ R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \\ \|_{i\in I} \quad [\![x_i.\langle Pc_i, Ad_i\rangle^{\circ}]\!] \Vdash^{\psi_i} & \longrightarrow_{\mathbb{T}}^{\overset{\mathsf{gc}}{\sim}} \end{array}$$

because by Lemma 2, $\tau_i \in [\![Pc_i]\!]J'$ for every x_i's involved, so there are executions of select that pick up the right substitutions. Then,

$$\begin{array}{rl} x.[\![[M \triangleright P]]\!]_x, W_{x.M}.D_{\triangleright}, \Vdash^{\varphi} & \\ ret.proceed(J) \triangleright [\![P\sigma]\!]_J & \\ R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \\ \|_{i\in I} \quad [\![x_i.\langle Pc_i, Ad_i\rangle^{\circ}]\!] \Vdash^{\psi_i} & x_i.[\![Ad_i]\!]_{J'}[ret.proceed/\mathsf{proceed}]_{\tau_i} \\ & \equiv \overset{\mathsf{gc}}{\sim} \end{array}$$

$$\begin{array}{rl} [\![x.[M \triangleright P]]\!] \Vdash^{\varphi} & \\ R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \\ \|_{i\in I} \quad [\![x_i.\langle Pc_i, Ad_i\rangle^{\circ}]\!] \Vdash^{\psi_i} & x_i.[\![\{Ad_i[P\sigma/\mathsf{proceed}]\tau_i\}_{J'}]\!]_{\bullet} \end{array}$$

Rule RED/OPAQUE**.**

Consider the reduction

$$x.[M \blacktriangleright P] \Vdash^{\varphi} x.\{\!\{M\sigma\}\!\}_{\bar{J}} \quad \multimap \quad x.[M \blacktriangleright P] \Vdash^{\varphi} \{P\sigma\}_{J'}$$

This rule is simulated by the chain:

$$\begin{array}{rl} [\![x.[M \blacktriangleright P]]\!] \Vdash^{\varphi} & [\![x.\{\!\{M\sigma\}\!\}_{\bar{J}}]\!]_{\bullet} \quad \equiv \\ x.[\![[M \blacktriangleright P]]\!]_x \Vdash^{\varphi} & x.[\![\{\!\{M\sigma\}\!\}_{\bar{J}}]\!]_{\bullet} \quad \longrightarrow \\ x.[\![[M \blacktriangleright P]]\!]_x \Vdash^{\varphi} & [\![P\sigma]\!]_{J'} \quad \equiv \\ [\![x.[M \blacktriangleright P]]\!] \Vdash^{\varphi} & [\![\{P\sigma\}_{J'}]\!]_{\bullet} \end{array}$$

Rule RED/OBSERVABLE**.**

Consider the reduction

$$\begin{array}{l} x.[M \circledcirc P] \Vdash^{\varphi} x.\{\!\{M\sigma\}\!\}_{\bar{J}} \quad \|_{i\in I} \quad x_i.\langle Pc_i, Ad_i\rangle^{\circ} \Vdash^{\psi_i} \quad \multimap \\ x.[M \circledcirc P] \Vdash^{\varphi} \{P\sigma\}_{J'} \quad \|_{i\in I} \\ x_i.\langle Pc_i, Ad_i\rangle^{\circ} \Vdash^{\psi_i} \{Ad_i\tau_i\}_{J'} \end{array}$$

This rule is simulated by the chain:

$$\begin{array}{rl} [\![x.[M \circledcirc P]]\!] \Vdash^{\varphi} & [\![x.\{\!\{M\sigma\}\!\}_{\bar{J}}]\!]_{\bullet} \\ R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \\ \|_{i\in I} \quad [\![x_i.\langle Pc_i, Ad_i\rangle^{\circ}]\!] \Vdash^{\psi_i} & \longrightarrow \end{array}$$

similar steps than for RED/NOASP:

$$\begin{array}{rl} x.[\![[M \circledcirc P]]\!]_x, W_{x.M}.D_{\circledcirc}, & \\ ret.proceed(J) \triangleright [\![P\sigma]\!]_J, \Vdash^{\varphi} & W_{init}.aspL([\![\bar{a}]\!]_{\bullet}) \\ W_{init}.aspL(asps) \triangleright \ldots & \\ R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \\ \|_{i\in I} \quad [\![x_i.\langle Pc_i, Ad_i\rangle^{\circ}]\!] \Vdash^{\psi_i} & \longrightarrow_{\mathbb{T}}^{\overset{\mathsf{gc}}{\sim}} \end{array}$$

$$\begin{array}{rl} x.[\![[M \circledcirc P]]\!]_x, W_{x.M}.D_{\circledcirc}, \Vdash^{\varphi} & ret.proceed(J'), \\ ret.proceed(J) \triangleright [\![P\sigma]\!]_J & \&_i x_i.advice(0, J', \tau_i) \\ R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \\ \|_{i\in I} \quad [\![x_i.\langle Pc_i, Ad_i\rangle^{\circ}]\!] \Vdash^{\psi_i} & \longrightarrow_{\mathbb{T}}^{\overset{\mathsf{gc}}{\sim}} \end{array}$$

because by Lemma 2, $\tau_i \in [\![Pc_i]\!]J'$ for every x_i's involved, so there are executions of select that pick up the right substitutions. Then, because no advice involved contains proceed (side condition of observable reactions):

$$\begin{array}{rl} x.[\![[M \circledcirc P]]\!]_x, W_{x.M}.D_{\triangleright}, \Vdash^{\varphi} & [\![P\sigma]\!]_{J'} \\ R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \\ \|_{i\in I} \quad [\![x_i.\langle Pc_i, Ad_i\rangle^{\circ}]\!] \Vdash^{\psi_i} & (x_i.[\![Ad_i]\!]_{J'})\tau_i \quad \equiv \overset{\mathsf{gc}}{\sim} \end{array}$$

$$\begin{array}{rl} [\![x.[M \circledcirc P]]\!] \Vdash^{\varphi} & [\![\{P\sigma\}_{J'}]\!]_{\bullet} \\ R.D_R \Vdash^{H_w} & R.asp_{act}([\![\bar{a}]\!]_{act}) \\ \|_{i\in I} \quad [\![x_i.\langle Pc_i, Ad_i\rangle^{\circ}]\!] \Vdash^{\psi_i} & x_i.[\![\{Ad_i\tau_i\}_{J'}]\!]_{\bullet} \end{array}$$

Rule MESSAGE-COMM**.**

Consider the reduction

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

This rule is simulated by the chain:

$$\begin{array}{rcl} \Vdash^{\varphi} [\![x.M]\!]_{\bullet} & \| & [\![x.D]\!] \Vdash^{\psi} \quad \equiv \\ \Vdash^{\varphi} x.[\![M]\!]_{\bullet} & \| & x.D' \,\mathsf{or}\, \mathcal{D} \Vdash^{\psi} \quad \longrightarrow \\ \Vdash^{\varphi} & \| & x.D' \,\mathsf{or}\, \mathcal{D} \Vdash^{\psi} x.[\![M]\!]_{\bullet} \quad \equiv \\ \Vdash^{\varphi} & \| & [\![x.D]\!] \Vdash^{\psi} [\![x.M]\!]_{\bullet} \end{array}$$

where the fact that there exists D' and $\mathcal{D}$ such that $[\![x.D]\!] \equiv x.D' \,\mathsf{or}\, \mathcal{D}$ can be proven by induction on D.

Rule MOVE**.**

Consider the reduction

$$\begin{array}{l} H[\mathcal{D}\colon (P \,\&\, \mathsf{go}(H'); Q)] \Vdash^{\varphi} \| \Vdash^{\psi H'} \multimap \\ \Vdash^{\varphi} \| H[\mathcal{D}\colon (P \,\&\, Q)] \Vdash^{\psi H'} \end{array}$$

This rule is simulated by the chain:

$$\begin{array}{rcl} [\![H[\mathcal{D}\colon (P \,\&\, \mathsf{go}(H'); Q)]]\!] \Vdash^{\varphi} & \| & \Vdash^{\psi H'} \quad \equiv \\ H[[\![\mathcal{D}]\!]\colon ([\![P]\!]_{\bullet} \,\&\, \mathsf{go}(H'); [\![Q]\!]_{\bullet})] \Vdash^{\varphi} & \| & \Vdash^{\psi H'} \quad \longrightarrow \\ \Vdash^{\varphi} & \| & H[[\![\mathcal{D}]\!]\colon ([\![P]\!]_{\bullet} \,\&\, [\![Q]\!]_{\bullet})] \Vdash^{\psi H'} \quad \equiv \\ \Vdash^{\varphi} & \| & [\![H[\mathcal{D}\colon (P \,\&\, Q)]]\!] \Vdash^{\psi H'} \end{array}$$

(B) $\mathcal{R}^{-1}$ is a simulation.

For the converse direction, we have to show that for $(\mathcal{C}_0, \mathcal{C}_1) \in \mathcal{R}$ and any reduction $\mathcal{C}_1 \longrightarrow \mathcal{C}_2$, one can fill the following diagram:

$$\begin{array}{ccccc} \mathcal{C}_0 & \overset{\mathcal{R}}{\text{———}} & \mathcal{C}_1 & \longrightarrow_{\mathbb{T}}^{*} & \mathcal{C}_1' \in \widetilde{\mathcal{C}_1}^{\overset{\mathsf{gc}}{\sim}} [\![\mathcal{C}_0]\!] \\ \vdots & & \downarrow & & \vdots \\ \mathcal{C}_0' & \overset{\mathcal{R}}{\text{– –}} & \mathcal{C}_2 & \longrightarrow_{\mathbb{T}}^{*} & \mathcal{C}_2' \in \widetilde{\mathcal{C}_2}^{\overset{\mathsf{gc}}{\sim}} [\![\mathcal{C}_0']\!] \end{array}$$

Again, we proceed by analysis of the kind of reduction.

Rule MOVE**.**

Suppose we have $H[\mathcal{D}{:}(P \& \mathsf{go}(H'); Q)] \Vdash^{\varphi} \in \mathcal{C}_1$ and $H[\mathcal{D}{:}(P \,\&\, Q)] \Vdash^{\psi H'} \in \mathcal{C}_2$.

As no reduction in $\longrightarrow_{\mathbb{T}}$ involves a migration and the rules are all left-linear, the migration rule is orthogonal to rules in $\longrightarrow_{\mathbb{T}}$. This means that we have $H[\mathcal{D}{:}(P' \,\&\, \mathsf{go}(H'); Q)] \Vdash^{\varphi} \in \mathcal{C}'_1$ for some P', so a similar migration can be done on $\mathcal{C}'_1$ ending up in $H[\mathcal{D}{:}(P' \& Q)] \Vdash^{\psi H'} \in \mathcal{C}'_2$ (by orthogonality, we have confluence).

Then, by a direct inversion lemma, we know that $H[\mathcal{D}{:}(P'' \,\&\, \mathsf{go}(H'); Q')] \Vdash^{\varphi} \in \mathcal{C}_0$ for some P'' and Q' such that $[\![P'']\!]_\bullet = P'$ and $[\![Q']\!]_\bullet = Q$. So we can apply the migration rule on $\mathcal{C}_0$. We set $\mathcal{C}'_0$ to be the resulting configuration. It is easy to check that $[\![\mathcal{C}'_0]\!] \overset{\mathsf{gc}}{\sim} \mathcal{C}'_2$.

Rule RED**.**

- If the reduction is $\longrightarrow_{\mathbb{T}}$, the normal form has not changed, so $\mathcal{C}'_1 = \mathcal{C}'_2 \overset{\mathsf{gc}}{\sim} [\![\mathcal{C}_0]\!]$ and we set $\mathcal{C}_0 = \mathcal{C}'_0$.
- 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 $\mathcal{C}'_2 \overset{\mathsf{gc}}{\sim} [\![\mathcal{C}'_0]\!]$.
- Otherwise, the reduction consumes a pattern $x.M_\sigma$ and produces a message of the form:

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

  By an inversion lemma, we know that $x.M'$ in $\mathcal{C}_0$ for some pattern M' such that $[\![M']\!]_\bullet = M_\sigma$.
  Then, by analyzing the reduction from $\mathcal{C}_2$ to $\mathcal{C}'_2$, we can recognize the simulation of one of the four possible reductions in the original configuration (as described above). Thus, it suffices to pick the right one and define $\mathcal{C}'_0$ as the result of this rule on $\mathcal{C}_0$ (by setting the substitution computed by the pointcuts to be the one choosing by the executions of select). We conclude this case by noting that in the proof that $\mathcal{R}$ is a simulation, the four reductions were handled similarly, starting with the emission of the message $w.weave(k, jp)$ and computing the normal form according to $\longrightarrow_{\mathbb{T}}$. So we have $[\![\mathcal{C}'_0]\!] \overset{\mathsf{gc}}{\sim} \mathcal{C}'_2$.

Rule MESSAGE-COMM**.**

- If the reduction is $\longrightarrow_{\mathbb{T}}$, the normal form has not changed, so $\mathcal{C}'_1 = \mathcal{C}'_2 \overset{\mathsf{gc}}{\sim} [\![\mathcal{C}_0]\!]$ and we set $\mathcal{C}_0 = \mathcal{C}'_0$.
- Otherwise, the reduction migrates a pattern $x.M$ that, by an inversion lemma corresponds to a pattern $x.M'$ in $\mathcal{C}_0$, with $[\![M']\!]_\bullet = M$. So the same communication can occur in $\mathcal{C}_0$.

To conclude the proof, we need to show that any configuration is bisimilar to its translation. This follows from the fact that $[\![\mathcal{C}_0]\!]$ is a normal form for $\longrightarrow_{\mathbb{T}}$ without message $proceed(J)$, so that $\mathcal{C}_0 \;\mathcal{R}\; [\![\mathcal{C}_0]\!]$. □

5 Discussion

We now elaborate on different design considerations, namely causality, synchronous weaving, and distributed aspect registries.

5.1 Dealing with Causality

In the aspect join calculus, we have introduced join points with causality, allowing pointcuts to discriminate join points based on the reactions that contributed to their occurrence. This choice is motivated by expressiveness, but it does make the calculus more complex and its implementation more challenging.

The motivation to deal with causality is inspired by prior work on aspect languages in different settings. Indeed, it is common in aspect languages to use control-flow related pointcuts in order to be able to discriminate join points based on call stack (*e.g.* `cflow` and `cflowbelow` in AspectJ [31]). Some proposals have even gone further, proposing history-based pointcuts that are not restricted to the call stack [16,17,34,43]. Also, all distributed aspect languages and systems support distributed control flow, although in a synchronous setting [4,39,44,57,59]. Leger *et al.* propose a library for distributed causality-based pointcuts in an asynchronous setting based on vector clocks [33].

Causality, be it synchronous or not, is important in practice for different reasons. A first basic motivation is that, more often than not, one needs to avoid advising aspects themselves. For instance, 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. These infinite loops can be avoided with control-flow pointcuts, or similar flow-based approaches [8,56].

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 $\neg\,\mathsf{causedBy}(*.rput)$.

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

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.

Causality in aspect languages is also very much useful for many typical applications of aspect-oriented programming. One salient example is security enforcement, such as access control [48], which can be handled by aspects. In particular, stack-based access control, as provided in Java, requires inspecting the call stack to determine whether a resource can be accessed or not; aspectizing these mechanisms requires pointcuts to access the call context [58]. Another related security mechanism that requires causality is integrity checking [7]: enforcing that certain actions are performed (or not) depending on whether the data or code comes from trusted parties.

In models of synchronous aspect languages, join points in context are represented as a linked data structure where a join point has a reference to its parent in the call stack [18]. The causality trees we have introduced in the aspect join calculus are a direct generalization of this model to the chemical setting. Of course, efficient implementations of aspect languages do not implement control-flow pointcuts or trace-based matching by relying on such costly structures [3,37,27]: instead, following the principles of partial evaluation [29], they statically evaluate which join points might potentially affect causality-related decisions, and introduce as little state-based indicators and bookkeeping operations as possible. Related techniques have been explored outside of the AOP community as well. For instance, Clements and Felleisen show that stack-based security mechanisms can be efficiently implemented using continuation marks [12].

Accordingly, we do not expect practical, scalable implementations of the aspect join calculus to implement causality trees as such. Causality trees are a fine *semantic* device, not a realistic implementation technique. It remains to be studied how the existing optimization techniques mentioned above for stack-based and trace-based mechanisms could be extended and adapted to the general setting of the causality tree. Simple techniques might be quite effective. For instance, assuming aspect definitions are available ahead of time, if an aspect uses the pointcut `causedBy("untrusted_label")` in order to discriminate doubtful computation, the implementation could simply *taint* all such join points as they are produced, making the implementation of `causedBy` a simple tag check.

5.2 Synchronous aspects

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. Doing so requires careful thinking about the synchronization of advices. The semantics we have presented corresponds to *asynchronous* reactions, in which all advices that match are triggered asynchronously. We could devise a weaving semantics that rather reflects the one of AspectJ by chaining implicitly advices and invoking them in a *synchronous* manner. First, this presents the issue of choosing the order in which advices are chained, which is not clear in an asynchronous setting. Second, the synchronous semantics can be encoded by an explicit chaining of advices and thus is not a primitive operations. For those two reasons, we have decided not to integrate a synchronous reaction in the semantics.

Also, 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 in 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. In case such sequentiality is needed, it has to be manually encoded using the typical explicit continuation-passing style. Fournet and Gonthier show how a wide variety of synchronization primitives can be easily encoded in the join calculus [21].

5.3 Distributed aspect deployment

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) [39], 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) [57], among others.

The formal model of the aspect join calculus that we have presented in this paper considers one central aspect registry. Extending this model to several registries is quite direct, does not affect the main results of this paper regarding the translation approach, and is straightforward to implement (in fact, our implementation of Aspect JoCaml supports multiple registries).

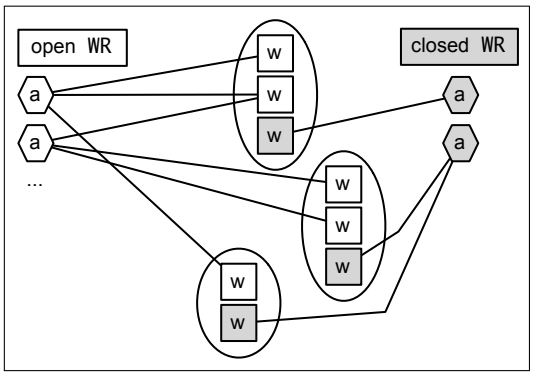


Fig. 13 Objects registering their public weavers to an open aspect registry (white), and the weavers of their sensitive reactions to a closed registry with only two trusted aspects (grey).

We describe here such an extension, and discuss the possibility to define fine-grained policies that go beyond prior work.

As a first step, we should extend the syntax of reactions with an exponent $M \triangleright^{reg} P$ to express that the reaction is registered in the aspect registry reg. In particular this means that only aspects deployed in registry reg are able to advise this reaction. (The case we have formally developed, where all aspects see all computations, corresponds to one global aspect registry to which all weavers and aspects are registered.)

An additional refinement of our model is to allow aspect registries to have different *policies*. For instance, the registry we have formalized is *open*, as it accepts dynamic aspect registration requests. One could allow some registries to be *closed*: a closed aspect registry is initiated with a fixed number of aspects deployed, and subsequently ignores any new aspect registration request. A closed registry would be defined with the following reactions:

$$D_R^{closed} = get_{asp}(k) \,\&\, asp_{act}(\tilde{a}) \triangleright k(\tilde{a}) \,\&\, asp_{act}(\tilde{a}) \\ deploy(a) \,\&\, asp_{act}(\tilde{a}) \triangleright asp_{act}(\tilde{a})$$

Figure 13 illustrates some of the topological flexibility offered by weaving registries and their policies: objects can register their reaction weavers in different registries with specific policies.

For instance, suppose a closed aspect registry reg_c with only a cache replication aspect and an open aspect registry reg_o. The following definition of the cache example guarantees that only cache replication can have access to the cache history, while any aspect registered with reg_o can observe accesses to the dictionary:

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

Note that closed repositories would allow a more efficient (and still correct) implementation of weaving, whereby the communication between weavers and the aspect registry can be limited to one initial request. Since the list of aspects is known to be immutable, there is no need to request it again upon each firing reaction.

An aspect registry policy may further specify that only weavers of a specific kind are accepted, such as observable reactions (Section 3.2). The design space of distributed deployment policies is wide and its exhaustive exploration is left open for future work.

6 Aspect JoCaml

Aspect JoCaml is a prototype implementation of the aspect join calculus on top of JoCaml, an extension of OCaml with join calculus primitives [24]. The implementation is directly based on the translation described in Section 4.

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.

6.1 Overview of Aspect JoCaml

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) =
    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

```
(* 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

```
(* 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 x.["put" (k,arg)] &&&
       Not(CausedBy(Contains _.["rput" (_,_)]))
   advice: repl#rput(k,arg) & proceed()
end
```

Fig. 15 Cache replication aspect in Aspect JoCaml

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 3.2. 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 an aspect registry, just as the extension discussed in Section 5.3. The parameter `ip` is passed at object creation time, making it possible to choose a different aspect registry for each created continuation object.

The definition of the cache class is given in Figure 14 and can be directly inferred from the definition of Section 3.2. 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 15. Labels in `Contains` pointcut are handled as strings and boolean pointcut combinators are defined by infix operators `&&&` and `|||`.

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

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

Then, a cache replication aspect can be registered to this aspect 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;
```

6.2 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 15). 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" [36]. 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 [54].

Static/dynamic pointcuts. Recall that the aspect 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 aspect registry does not need to pass the aspect to the weaver for weaving.

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.

Depth of causality tree. As discussed in Section 5.1, an optimized, scalable management of the causality tree is a challenging research challenge. The current implementation of Aspect JoCaml is naive in this regard: it is a direct implementation of the calculus, and as such keeps track of every causal match. This means that the causality tree may grow unboundedly. A simple general optimization to implement is to put a bound on the maximum depth of the tree, although this would change the semantics of pointcut matching. For instance, keeping only a bounded version of the causality tree may imply that a `causedBy("untrusted_label")` pointcut does not match, whereas the compromised label occurs is in fact in the *complete* causality tree, thus introducing a security flaw. In this sense, partial evaluation techniques, which sacrifice dynamism but preserve the matching semantics, might be preferable.

Non-determinism. The family of join calculi are non-deterministic. Similarly, the aspect join calculus is non-deterministic. In the translation, weavers rely on a random selection of a matching substitution (Figure 11). Aspect JoCaml is a direct implementation of the translation, and as such, uses the non-deterministic `select` function. This non-deterministic behavior in the translated code is important to prove the bisimulation with the direct semantics, which is also non-deterministic. Of course, a given implementation could make an arbitrary choice, which would be more efficient than an actual random selection. But programmers should not rely on determinism. In essence, this is similar to how the evaluation order of procedure arguments is left unspecified in the standards of some languages (such as C, C++, and Scheme): technically, an implementation can choose any arbitrary (even random) order for evaluating function arguments. This forces programmers to not rely on a specific evaluation order in their programs, even though most sequential implementations will (arbitrarily) adopt a left-to-right order. The same happens with JoCaml as an implementation of the objective join calculus in OCaml. It is supposed to be non-deterministic in the order of matching reactions, but it might very well be implemented using a FIFO strategy internally.

7 Related work

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

7.1 Formal semantics of aspects

There is an extensive body of work on the semantics of aspect-oriented programming languages (*e.g.* [62,13, 18,28,15]). 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 [28].

7.2 Distributed aspect languages and systems

We now compare specific features of practical distributed aspect languages and systems—in particular JAC [44], DJcutter [39], ReflexD [57], and AWED [4]—and relate them to the aspect join calculus.

Quantification. Remote pointcuts were first introduced in DJcutter and JAC, 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 5.2). 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 [10, 41, 51, 54].

Advanced quantification. DJcutter, AWED, and ReflexD support reasoning about distributed control flow, in order to be able to discriminate when a join point is in the (distributed) flow of a given method call. AWED also supports 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 support reasoning about causality; additional mechanisms are needed, for instance as developed in WeCa [33]. In Section 3.3, we describe how join points can capture their causality links, which can then be used for pointcut matching. While the synchronous communication pattern can be recognized in order to support a similar notion of distributed control flow, the causality tree model is much more general. An interesting venue for future work is to develop a temporal logic for pointcuts that can be used to reason precisely about causality. Temporal logic has been used in some aspect-oriented systems to perform semantic interface conformance checks [9]. 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 [14].

Aspect deployment. DJcutter adopts a centralized architecture with an aspect host where all aspects reside and advices are executed. JAC supports distributed aspect deployment onto 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 5.3 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* [55]. 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 programmers 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.

8 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. The aspect join calculus exposes causality trees for join points, supports customized weaving semantics and decentralized aspect weaving. In particular, customized weaving supports strong encapsulation (some reactions can be totally hidden from aspects) and non-interference guarantees (some reactions can be restricted to observer aspects [47]).

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 [21]. 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 [26]. Another interesting perspective is to study the application of aspects in chemical engines for Cloud computing.

Acknowledgments

We thank the anonymous reviewers for their valuable and detailed suggestions on how to improve this article.

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 2006)*, 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. Kenneth J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, USA, April 1977.
8. Eric Bodden, Florian Forster, and Friedrich Steimann. Avoiding infinite recursion with stratified aspects. In *Proceedings of Net.ObjectDays 2006*, Lecture Notes in Informatics, pages 49–54. GI-Edition, 2006.
9. 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.
10. Eric Bodden, Éric Tanter, and Milton Inostroza. Join point interfaces for safe and flexible decoupling of aspects. *ACM Transactions on Software Engineering and Methodology*, 23(1):7:1–7:41, February 2014.
11. 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.
12. John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1029–1052, November 2004.

13. 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.
14. 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.
15. 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.
16. Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005.
17. Rémi Douence and Luc Teboul. A pointcut language for control-flow. In Gabor Karsai and Eelco Visser, editors, *Proceedings of the 3rd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2004)*, volume 3286 of *Lecture Notes in Computer Science*, pages 95–114, Vancouver, Canada, October 2004. Springer-Verlag.
18. 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.
19. Gary Duzan, Joseph Loyall, Richard Schantz, Richard Shapiro, and John Zinky. Building adaptive distributed applications with middleware and aspects. In Lieberherr [35], pages 66–73.
20. Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), October 2001.
21. 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.
22. 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.
23. 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.
24. 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.
25. 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.
26. 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.
27. Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In Lieberherr [35], pages 26–35.
28. Radha Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming*, 63:267–296, 2006.
29. Neil D. Jones, Charles K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* International Series in Computer Science. Prentice Hall, 1993.
30. 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.
31. 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.
32. 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.
33. Paul Leger, Éric Tanter, and Rémi Douence. Modular and flexible causality control on the web. *Science of Computer Programming*, 78(9):1538–1558, September 2013.
34. Paul Leger, Éric Tanter, and Hiroaki Fukuda. An expressive stateful aspect language. *Science of Computer Programming*, 102(1):108–141, May 2015.
35. Karl Lieberherr, editor. *Proceedings of the 3rd ACM International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK, March 2004. ACM Press.
36. Louis Mandel and Luc Maranget. The JoCaml language release 4.00. Inria, August 2012.
37. 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.
38. 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.
39. Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote pointcut – a language construct for distributed AOP. In Lieberherr [35], pages 7–15.
40. Marko Obrovac and Cédric Tedeschi. Experimental evaluation of a hierarchical chemical computing platform. *International Journal of Networking and Computing*, 3(1):37–54, 2013.
41. Bruno C. d. S. Oliveira, Tom Schrijvers, and William R. Cook. EffectiveAdvice: discplined advice with explicit effects. In AOSD 2010 [1], pages 109–120.
42. Bruno C. D. S. Oliveira, Tom Schrijvers, and William R. Cook. MRI: Modular reasoning about interference in incremental programming. *Journal of Functional Programming*, 22:797–852, November 2012.
43. Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Andrew P. Black, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of *LNCS*, pages 214–240. Springer-Verlag, 2005.
44. 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.
45. Jean-Louis Pazat, Thierry Priol, and Cédric Tedeschi. Towards a chemistry-inspired middleware to program the internet of services. *ERCIM News*, 85(34), 2011.

46. James Riely and Matthew Hennessy. A typed language for distributed mobile processes. In *Proceedings of POPL'98*, pages 378–390. ACM Press, 1998.
47. Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM Symposium on Foundations of Software Engineering (FSE 12)*, pages 147–158. ACM Press, 2004.
48. Pierangela Samarati and Sabrina De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *FOSAD '00: Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design*, pages 137–196, London, UK, 2001. Springer-Verlag.
49. 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).
50. 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).
51. 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.
52. 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.
53. Nicolas Tabareau. A theory of distributed aspects. In AOSD 2010 [1], pages 133–144.
54. 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.
55. 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.
56. Éric Tanter, Ismael Figueroa, and Nicolas Tabareau. Execution levels for aspect-oriented programming: Design, semantics, implementations and applications. *Science of Computer Programming*, 80(1):311–342, February 2014.
57. 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.
58. Rodolfo Toledo, Angel Núñez, Éric Tanter, and Jacques Noyé. Aspectizing Java access control. *IEEE Transactions on Software Engineering*, 38(1):101–117, Jan./Feb. 2012.
59. Eddy Truyen and Wouter Joosen. Run-time and atomic weaving of distributed aspects. *Transactions on Aspect-Oriented Software Development II*, 4242:147–181, 2006.
60. Philip Wadler. How to replace failure by a list of successes. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '85)*, volume 201 of *Lecture Notes in Computer Science*, pages 113–128, Nancy, France, September 1985. Springer-Verlag.
61. 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.
62. 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.