A chemical programming language for orchestrating services : Application to interoperability problems

Mayleen Lacouture

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Mayleen LACOUTURE

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'École nationale supérieure des mines de Nantes
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 31/10/2014
Thèse n° : 2014 EMNA 0011

A Chemical Programming Language for Orchestrating Services

Application to Interoperability Problems

JURY

Rapporteurs :	**M^me^ Isabelle BORNE**, Professeur des universités, Université de Bretagne-Sud
	M. Pascal POIZAT, Professeur des universités, Université Paris Ouest Nanterre la Défense
Examinateurs :	**M. Achour MOSTEFAOUI**, Professeur des universités, Université de Nantes
	M. David BROMBERG, Maître de conférences, LABRI – Université de Bordeaux Segalen
Directeur de thèse :	**M. Mario SÜDHOLT**, Professeur de l'Institut Mines-Télécom, Ecole des Mines de Nantes
Co-directeur de thèse :	**M. Hervé GRALL**, Chargé de Recherche, Ecole des Mines de Nantes

Contents

List of Tables

List of Figures

Acknowledgments

I would specially like to thank my advisor Hervé Grall for his expertise, understanding, and patience. I appreciate his participation in all the stages of my research project always providing his help and advice. I would like to thank all the members of the Ascola team for their support.

A very special thanks go to Mario Südholt who accepted to be my director of thesis, whose advice and support were important for the realisation of this thesis.

I must also thank Isabelle Borne and Pascal Poizat who participated as reviewers of my thesis, providing valuable remarks and suggestions. I would like to thank Achour Mostefaoui and David Bromberg for their participation as examiners.

Special thanks also to the secretary staff for all the moments in which their assistance helped me along the way.

Finally, I would like to thank my family for the understanding and support they provided, specially my husband, Vincent, whose love and encouragement helped the completion of this thesis.

Introduction

With the emergence of cloud computing and mobile applications, it is possible to find a web service for almost everything. A service is a computer program that provides a set of operations accessible from a network address. Client programs on the web interact with the service using `HTTP` messages. Thanks to the variety of available web services, developers can create complex applications by combining several independent services, whose arrangement and execution can be automated with the aid of orchestration languages.

Nevertheless, the diversity of technologies and the lack of standardization can hinder the collaboration between services. Imagine for instance that you are writing a mobile application that allows users to publish, manage and share their photos. For storing and managing the photos there are several online services, with `Flickr` being one of the most popular. By using the `Flickr` application programming interface, you implement an application that communicates with this service. But once the application gains in popularity, more and more users would like to be able to use other alternative services like `Picasa`. However, `Flickr` and `Picasa` differ not only in the way photos are organized, but also on the protocols and technologies they use to provide their services. By protocols we mean the way in which messages are organized to complete common tasks, like searching and editing: while `Flickr` directly provides a variety of more or less complex operations, `Picasa` relies on client libraries that perform the same tasks by combining the basic `HTTP` methods `GET`, `POST`, etc. From a technological point of view, although both services provide `REST` programming interfaces, `Flickr` also allows clients to use `SOAP` and `XML-RPC`.

Thus, we face interoperability problems due to the heterogeneity of the different services. In this respect, we have identified three types of problems, namely adaptation, integration and coordination, that can be described using the photo management example.

Adaptation: the client application that orchestrates `Flickr` services must be adapted to orchestrate the services supplied by `Picasa`

or other providers.

Integration: the client application has to orchestrate both `Picasa` and `Flickr` services by defining a common data model and interface.

Coordination: from the point of view of the languages used in the orchestration of web services, existing scripts written in different languages need to be coordinated for cooperating in the orchestration of the services provided.

Middleware infrastructures are usually proposed for solving interoperability problems in the form of bus architectures with a central component that translates messages. Nevertheless, a complete solution requires a universal representation of resources. Our approach, analogous to these works, consists of a pivot architecture that integrates different orchestration languages with heterogeneous service providers around a pivot language, thus allowing the implementation of typical programming patterns: the adapter pattern for solving adaptation problems, the facade pattern for solving integration problems, and the mediator pattern for solving coordination problems. The challenge remains to find the adequate orchestration language that can be used as a pivot language.

The thesis of this dissertation is that the chemical programming paradigm, which has already been studied as a solution for service oriented programming [BP09], can provide the foundations for an orchestration language in a pivot architecture. Concretely,

- we present a new orchestration language, called `Criojo`, which implements and extends an original calculus[1] based on a *chemical abstract machine* (cham) dedicated to service-oriented computing,
- we show how the orchestration language can be used to define a pivot architecture.

The consequence of adopting the approach we have developed would be an improvement in the interoperability of services and orchestration languages, thus easing the development of composite services. The high level of abstraction of `Criojo` could allow developers to write very concise programs since message exchanges are represented in a natural and

[1] The calculus, called Heta-calculus, has been designed by Thesis Adviser Hervé Grall.

intuitive way. These programs could be used not only as effective orchestrations, replacing orchestrations written in traditional languages, but also as prototypal orchestrations, giving a clear specification for concrete orchestrations written in traditional languages. Moreover, the formal foundations of `Criojo` provide a specification of the core of an orchestration language for a pivot architecture, which leads to many advantages, not only during the development phase of the language but also during the specification and the validation phases of orchestrations written in the language.

- The formal specification being clear and concise eases the language implementation, avoiding the pitfalls often encountered in standards, as in the orchestration language `BPEL` [HHH10];

- The formal specification provides the theoretical basis of useful tools for specifying, testing and verifying orchestrations.

Organization of the dissertation. We have organized this dissertation as follows.

- In Chapter 1, we give an overview and a state of the art of service-oriented computing. We begin with the fundamental concepts of distributed computing that lead later to service oriented computing, with the objective of finding the required characteristics that a language for the orchestration of web services must have. We finish the chapter with a specification, in the form of a list of requirements for an orchestration language, integrating aspects for service-oriented computing and data computing.

- In Chapter 2, we present the chemical calculus, called Heta-calculus. After the specification given in th previous chapter, we describe the development process, following a standard V-model. After a state of the art presenting previous works that led to the Heta-calculus, the chapter presents

  - the design decisions that have been made,
  - the calculus, with its syntax and its chemical semantics,
  - the validation against the requirements.

- In Chapter 3, we show how the theoretical concepts of the Heta-calculus lead to practical applications in the form of a programming language called `Criojo`, which we describe as a language for writing chemical reaction rules. The practical aspects of `Criojo` are treated from the point of view of the developer, in the form of a tutorial, and from the point of view of the implementer with a set of guidelines for possible implementations in different host languages.

- In Chapter 4, we continue with the practical aspects of the language, by showing the real applicability of the prototype in the context of web services. With the aid of `Criojo` as the language at the core of a pivot architecture, we solve interoperability problems in the case of photo management with web applications like `Picasa` and `Flickr`. We present an example for each of the problems listed above, and propose a solution based on a design pattern whose implementation is eased by the pivot architecture.

- The last chapter concludes with a discussion of our results, what we have learnt and the perspectives that arise from the results of our work.

Contributions. Finally, this dissertation makes the following contributions:

- a well-reasoned definition of a set of requirements for an orchestration language,

- the formulation of all the design decisions that have been made for the design of the chemical calculus and its validation against requirements, in addition to the presentation of the original calculus,

- a prototype implementation of an orchestration language called `Criojo`, based on the theoretical foundation given by the chemical calculus, described from the points of view of a programmer and an implementer respectively,

- especially, a set of helpful extensions that ease the programming of real world applications, like the ability to interface with external functions and resources,

- a method for the development of different solutions to interoperability problems, in the form of a pivot architecture using `Criojo` as a pivot language.

Chapter 1

Towards an Orchestration Language for Web Services

Orchestration is the automated arrangement and execution of different autonomous processes. Accordingly, an orchestration language for web services allows to program new web services by specifying the composition and coordination of existing services. In this chapter we study the state of the art on service-oriented computing with the objective of defining a solution space from which the requirements for an orchestration language can be drawn.

The first part of this chapter is dedicated to languages for service-oriented programming. Actually, we start with the basis of distributed computing, since distributed computing provides the foundations for service-oriented computing. Indeed, service-oriented computing can be considered as a "computing paradigm that utilizes services as fundamental elements to support rapid, low-cost development of distributed applications in heterogeneous environments. The promise of Service-Oriented Computing is a world of cooperating services that are being loosely coupled to flexibly create dynamic business processes and agile applications that may span organizations and computing platforms, and can adapt quickly and autonomously to changing mission requirements" [GP09, chap. 1]. With the advent of the Web, and the associated Web services, service-oriented computing now represents a new generation platform for distributed computing. It is the reason why we study some fundamental notions for distribution, like the different communication and synchronization models, key elements in the definition of the language.

The second part covers another important aspect of any programming language, which is the representation of data and the computation over these representations. Concretely, we focus on the representation of resources, which can be accessed and manipulated via services.

Finally, from the state of the art we draw a set of requirements for an orchestration language, integrating aspects for service-oriented computing

and data computing. These requirements form the specification for the development of the Heta-calculus and of `Criojo`, presented in the next chapters.

1.1 From Distributed Computing to Service-Oriented Computing

Distributed computing deals with computations that involve multiple *agents*, in other words, with distributed computations involving communication. These agents are in fact autonomous computation units which collaborate with each other to form a distributed system. According to the mechanism used to collaborate, there exist two models for distributed computing: one is based on *message-passing*, the other is based on *shared memory*. Distributed computing introduces new concerns that cannot be tackled with traditional sequential programming. Therefore distributed programming requires programming languages dedicated to distribution. On the logical level, languages for distributed programming are based on one of the two models of distributed computing, but distribution also occurs at other levels: hardware, systems and middleware over which programs will execute. Hence an important implementation issue is that the distribution model of the concrete and logical level may differ, and different combinations between models at the concrete and logical level are possible, as shown in Figure 1.1: a language based on a message-passing model can be implemented on a system with shared memory and vice-versa.

Before coming into the core of the section, it is important to note the difference between concurrency, parallelism, and distribution, three concepts that somehow overlap. Parallelism is the property of multiple activities executing simultaneously. Concurrency is a more general concept than parallelism, where activities may not execute at the same physical time, but execute at the same logical time: it is a form of virtual parallelism. Clearly, distribution implies both parallelism and concurrency: activities are distributed over agents executing in parallel, and at the same time each agent may execute in a concurrent way, when dealing with communications and local computations. For instance, consider an agent that sends a request to the `Picasa` server and continues to operate while waiting for a response. When the answer finally arrives, it has to deal with both the incoming message and current computations.

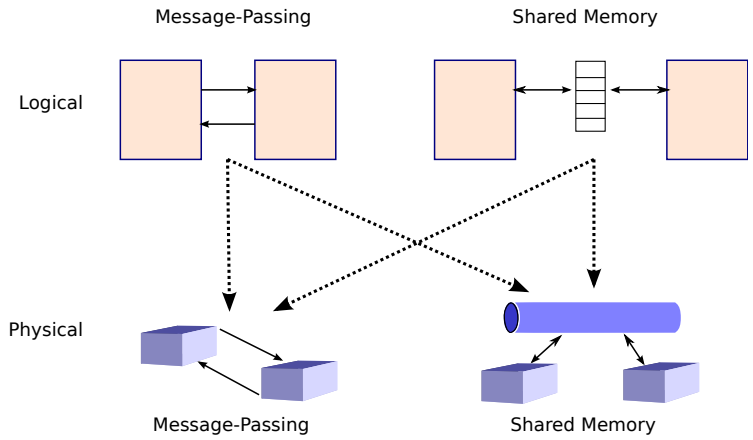

Figure 1.1: Languages based on message-passing can be implemented on a hardware system with shared memory, and vice-versa.

However, distributed programming is traditionally regarded as independent from concurrent and parallel programming since each of these forms addresses different kinds of problems. Distributed programming tackles communication, synchrony and fault tolerance problems, while concurrent programming addresses problems related to resource sharing, and parallel programming addresses problems related to performance. Nevertheless, in this thesis we adopt a different point of view since we consider that distributed programming encompasses concurrent and parallel programming.

Finally, distributed computing leads to service-oriented computing. Contrary to a class of distributed systems, which are built from highly coupled components to solve a specific computational problem, service-oriented systems are more loosely coupled and more open: they allow complex applications to be built by combining several independent components called services. Thus, service-oriented computing is a specific instance of distributed computing.

1.1.1 Fundamentals of Distributed Computing

A distributed system is a set of autonomous computational units, here called agents, that collaborate to give the aspect of a single coherent system [TS06, p.2]. To collaborate, it is necessary to establish a communication mechanism between agents. To act as a coherent system,

agents must be synchronized [LL90]. To be reliable, the system needs to continue functioning in the presence of faults. Thus, the design of distributed systems relies on three fundamental aspects: communication, synchronization and fault tolerance.

1.1.1.1 Communication Models

In the mainstream models of distributed computing, a system is represented as a set of autonomous nodes or agents, which execute in parallel and interact with each other by exchanging messages [LL90]. These are called *message-passing* models. Other models of distributed computing represent communication between agents via a shared memory space. These are called *shared-memory* models.

Shared-Memory Models. In the literature, shared-memory models relate mostly to parallel or concurrent computing. Nevertheless, shared-memory computing is sometimes treated as a particular case of highly coupled distributed computing. In a shared-memory model agents communicate by writing and reading registers in a shared memory space, which can be centralized or distributed. Access to the shared memory can be uniform (UMA), with all the processors having the same opportunity and access time, or non-uniform (NUMA). An important issue in shared-memory models is to ensure the mutual exclusion between critical sections, namely to prevent two agents from accessing the same shared register at the same time.

Message-Passing Models. In message-passing models, the system is represented as a communication graph, where arcs between nodes correspond to communication links between agents, called channels. Two nodes connected by a link can send messages directly to one another. Communication graphs can be directed or undirected, representing one-way or two-way communications, respectively. A typical implementation resorts to *firewalls* for controlling communication: a firewall may forbid communication over some channels. Moreover, some models assume that the topology of the system changes dynamically as new links are created between nodes thanks to discovery and redirection. We refer to this as *channel mobility*, a notion introduced by the π-calculus [MPW92a, MPW92c], a foundational process calculus described in the next chapters.

One good example of channel mobility is a discovery scenario in a service-oriented architecture, as presented in Figure 1.2. Whenever a client agent needs a service, it sends a look-up request to the broker managing a service registry and providing the run-time discovery of services. In this way, the location of a service can change without affecting the clients. However, channel mobility raises the problem of channel scopes with re-

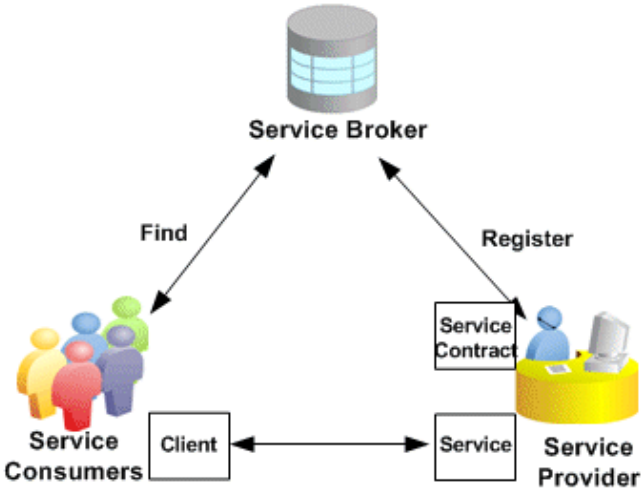

Figure 1.2: Discovery (Credit: Oracle)

spect to firewalls: if a channel traverses as data a firewall that forbids any communication over this channel, then either the firewall maintains the prohibition, or it drops it, allowing *scope extrusion* [MPW92a, Ex. 3].

Message-passing models abstract away from the details of communication, which can be decomposed into five steps.

Production: A source agent produces a message.

Emission: The same agent sends the message.

Routing: The network routes the message to its destination.

Reception: The target agent receives the message.

Consumption: The same agent consumes the message.

A latency between production and emission, during routing and between reception and consumption is possible and can be modeled as a relation

between agents clocks and communication delays. If the total delay is unbounded, communication is said *asynchronous*. If it is (logically) null, communication is said *synchronous*. As a result of the distinction between production and emission on the one hand, and reception and consumption on the other hand, communication is often modeled with buffers to hold input and output messages. Buffers can be finite or infinite, depending on their capacity to hold messages. Infinite buffers is the most common assumption given the great capabilities of real systems. Synchronization is necessary in distributed systems, as in any concurrent system, as agents need to be coordinated to achieve a specific goal. Assume for instance a task A that must be executed only after a task B has executed (serialization). In a centralized context, synchronization depends on schedulers based on a centralized clock, which rule the execution of agents. However, in a distributed system this is impossible, since each process has its own clock, which may drift from other agents' clocks. Thus, distributed systems are by definition asynchronous. Nevertheless, it is possible to establish relations between the independent times of agents by means of communication. The more restrictive form of synchronization is synchronous communication, where sending and receiving events occur at the same theoretical time. Since this simultaneity is impossible due to physical conditions, synchronous communication is rather defined as a communication where the sender remains blocked waiting for its request to be accepted. It is therefore possible to simulate synchronous communication with asynchronous communication. The problems related to this translation and its inverse are discussed later in Section 1.1.2.4. The more liberal form of synchronization is asynchronous communication: it allows to assert that the emission event has happened before the reception event, and nothing else. There are intermediate forms of synchronization, allowing the preservation of the order between messages from production to consumption. Thus, the causal order is preserved when the buffers implement a first-in first-out (FIF0) discipline and the network implements a synchronous communication [MF95].

As an example of a message passing system, think of a client agent that communicates with a `Flickr` or `Picasa` server agent. To retrieve a list of photos from the server, the client sends a message containing a user identifier, some search criteria, and the location of the channel where it would like to receive the answer. Thanks to the channel information, the server knows where to send the response directly to the client. The

communication is asynchronous, since the delay between the request and the response is unbounded. Nevertheless, if the client agent has previously sent a message to assign tags to some of the photos, producing an effect over the search result, the preservation of the order between the two messages is necessary to maintain consistency.

Finally, shared memory models can be implemented with message-passing, by having one or more processes that control the access to shared variables. Then, variables can be accessed with request and response messages. Nevertheless, the fault tolerance of the system can be compromised since the halting of any process managing shared variables implies the halting of the complete system.

1.1.1.2 Fault Tolerance

Fault tolerance, or the capacity of functioning in the presence of faults, is an important aspect of distributed systems, since they are inherently error prone: faults can have many origins, either agents or the network. In a distributed system, the behavior of a process can be modeled as a transition system led by communication events. A *fault* is then represented as a transition firing an *error*, which may lead to a deviation with respect to the specification of the system [GÖ99], in other words, to a *failure*. Let us illustrate these terms with an example. Consider the function

```
int f(){
  x:=1;
  y:=0;
  return x + (x/y);
}
```

There is a *fault* in the program since variable `y` is initialized with zero while being a divisor. This fault produces an execution *error* if the function is called, leading eventually to a *failure* which is the interruption of the program. According to Gärtner, a model for fault tolerance is based on some class of faults and offers some level of tolerance with respect to the faults in the class. Traditional classes consider Byzantine, omission and crash faults respectively [LL90]. Byzantine faults are arbitrary faults: they can model any fault, which occurs due to the misbehavior of a process or of the network. An omission fault occurs when a process fails to communicate due to problems in the communication network. A crash fault occurs when a process stops working, reaching an invalid state from which it cannot recover. Mechanisms for fault tolerance aim at avoiding

failures, in other words at satisfying all the properties required by the specification of the system, despite faults. It is possible to classify these mechanisms by resorting to a current decomposition of these specification properties: indeed they can be subsumed under the conjunction of a safety property and a liveness property [AS85], where the safety property asserts that nothing bad ever happens and the liveness property asserts that something good will eventually happen [Lam77]. For instance, a safety property can be defined as a conjunction of local invariants, but also as a global invariant. A typical example is *consistency*, a property asserting that for all the clients of a resource, their own view of the resource and its original value are consistent: if a client modifies a resource, any subsequent request sees the modification. A liveness property is often defined as a termination property or an availability property. For example, in a client-server interaction, it can assert that any request terminates with a response, in other words, that the service is always available.

The decomposition into safety and liveness directly leads to four types of fault tolerance, according to whether the safety and liveness properties are preserved or not respectively. The resulting combinations are shown in Table 1.1.

	Property	
Fault tolerance type	Safety	Liveness
Masking	✓	✓
Fail-safe	✓	×
Robust	×	✓
None	×	×

Table 1.1: Fault tolerance type according to whether safety and liveness are preserved.

The strongest form of fault tolerance is *masking* fault tolerance, which transparently preserves the safety and liveness of the system in the presence of a fault. On the other hand, the weakest form of fault tolerance is to do nothing, which does not guarantee any property. The remaining two intermediate forms guarantee either only safety or only liveness. These are called *fail-safe* and *robust* fault tolerance, respectively. The preservation of both safety and liveness may require a trade-off: thus, serializability

(or linearizability), a strong notion of consistency, and availability are incompatible for a service [GL12]. Either a weaker notion of consistency must be adopted or availability must be sacrificed.

The protection mechanisms for fault tolerance are mainly based on redundancy, used for detection and correction [G̈99]. Redundancy essentially avoids the loss of resources to be fatal: it corresponds not only to replications [CBPS10] but also to superfluous additions, like logging or different kinds of checks. Detection and correction allow errors to be circumvented. Precisely, error detection enables an erroneous state to be identified; error correction enables to recover from an error, either by bringing the system to a correct state in a backward or forward way, or by compensating the error. For example, in a transaction management system, a protection mechanism is to keep a log of the modifications sustained by the system during a transaction. When an error is detected before the transaction is committed, the log is used to correct the error by reverting the modifications. In this way the system returns to a correct state.

1.1.2 Distributed Programming

Distributed programming differs in many aspects from sequential programming. For this reason, specialized languages have been designed to handle communication and synchronization between computation units that execute in parallel, and to handle detection and recovery of errors, which are more prone to happen in distributed applications. Thus, the aspects of distributed computing that we discussed before translate into essential requirements that languages for distributed programming must satisfy:

- mechanisms for communication and synchronization,
- mechanisms for parallel execution, and
- a support for error detection and recovery [Bal90].

One additional question that is treated in the more general context of concurrent programming is the sharing of resources when multiple clients try to access and modify the same resource. This translates into an additional requirement related to communication and synchronization: a concurrency control mechanism.

From the implementation point of view, programming languages must deal with the configuration of the execution level, which not always matches the choices made for communication, parallelism and fault tolerance at the language level. We begin by presenting the different alternatives for fulfilling the different requirements and conclude by discussing the issues related to the implementation at the execution level.

1.1.2.1 Communication and Synchronization

In the previous section we presented the two main communication models for distributed systems: message-passing and shared-memory. There are several variations of the two models implemented by existing languages for distributed programming, as shown in Table 1.2.

Communication	Message Passing	Synchronous	Rendez-Vous
			RPC
		Asynchronous	
	Shared Memory	Variables	
		Tuples	
		Logical variables	

Table 1.2: Variations of message passing and shared memory communication models.

Message-passing. Message passing can be synchronous or asynchronous. In synchronous message-passing, the sender and the receiver synchronize at some point during their execution, corresponding to a *rendez-vous* or a *Remote Procedure Call* (RPC). In the rendez-vous model, first introduced by the `Ada` programming language, the sender and the receiver synchronize at specific interaction points, where they can exchange information synchronously. Afterwards, both participants continue their execution in parallel. RPC is a procedure call between distributed agents, where the sender invokes a procedure and rests blocked during the execution of the procedure. Unlike rendez-vous, the receiver does wait for the invocation to occur. An example of RPC is `Java`'s Remote Method Invocation [WRW96].

In asynchronous message passing, the sender does not wait for its message to be accepted. The communication can be point-to-point or broadcast. A point-to-point communication is a direct exchange between sender and receiver. Hewitt's actor model [HBS73] is an example of asynchronous point-to-point communication: actors are logical entities that execute in parallel and communicate by directly sending messages to one another via mailboxes. Messages are stored in mailboxes in the same order they arrive, waiting to be retrieved by the actor. The actors model is implemented by languages like `Erlang`, and more recently `Scala`. Contrary to point-to-point communication is broadcast communication, where a sender publishes a message for several known (multicast) or unknown recipients. The idea of broadcast programming was introduced by Gehani in his Broadcasting Sequential Processes (BSP) [Geh84], where agents communicate by broadcasting in order to implement a particular program. Broadcasting is often used in algorithms for game programming such as the alpha-beta algorithm, where a set of *slave* agents is in charge of calculating all possible moves.

Shared memory. The alternative to message-passing is communication via variables in a shared memory space. The advantage of this type of communication is that modifications have immediate effect, contrary to message-passing, where there is a delay between sending and receiving of a message. Moreover, messages can be broadcasted as multiple agents can access the same variable. Nevertheless, languages based on shared memory must provide mechanisms to avoid race conditions. Efficient for avoiding race conditions, mutual exclusion is often implemented with the help of locks. To enter into a critical section, the lock protecting a resource must be free: it then becomes acquired. At the end of the critical section, the lock is released. Beyond critical sections, transactions allow sequences of actions to become an atomic, indivisible, action. There are two classical approaches for implementing transactions. The pessimistic approach prevents conflicts from happening. There are two predominant solutions, the first one based on locks and a two-phase protocol [BHG87, chap. 3], the second one based on timestamps [BHG87, chap. 3]: in both cases, at each access, a control is done, possibly resulting in blocking or aborting. The optimistic approach [Her90] is more liberal. During the execution, initial read accesses are effectively executed whereas write accesses are virtually executed over a copy. At the commit time, there is

a validation phase, followed if it is successful by a write phase performing the real write accesses. These approaches are effectively applied in software transactional memory [LR06], that is memory equipped with a software transactional mechanism, as in database management systems.

Shared-memory architectures are typical of sequential programming languages providing a notion of threads, like `Java` or `Haskell`: a program decomposes into multiple threads that execute concurrently or even in parallel on multicore machines, and share a memory space. We now focus on shared memory in a distributed context.

One example is `Linda`'s tuple space (TS) [ACG86], where information is stored in the form of tuples that can be added and retrieved by process with the three following operations:

`out ("foo", 42)`	Adds the tuple to TS. The tuple can then be queried by any of its components.
`in ("foo", int x)`	Removes the tuple from TS. In this case the first parameter "`foo`" serves as key.
`read("foo", int x)`	Consults the tuple without removing it.

Conflicts are avoided as tuples cannot be modified in situ: a tuple can be read by multiple agents, but modified only by the process that removes it from the TS. Thus, no blocking is needed.

Another example of shared variables are logical shared variables, which are used in logic languages like Concurrent `Prolog` [Sha86]. In this model agents communicate via logical variables, which are assigned a value by unification during goal reduction. Binding is irreversible and once a value is assigned to a variable, it cannot be changed. For example, the following two agents communicate and synchronize with the variable `X`:

```
A(X?,Y), C(X).
```

Note that only process `C` is allowed to bind `X`, while `A` can only read the variable. Thus a variable in a shared memory is transformed into a communication channel.

1.1.2.2 Parallelism

The most common way to express parallelism is by representing each parallel task as a virtual process, executing on a sequential processor with its own state. At the opposite side, sequential tasks can be composed as a unique virtual process, sequentially executing. Generally speaking,

the execution of tasks can be abstracted as a graph of task dependencies: there is a link from task `T1` to task `T2` if the execution of `T1` must immediately precede the execution of `T2`. A task can be executed if all preceding tasks have been executed.

Parallel tasks can be expressed in the form of statements, objects, functions, logical clauses or processes.

- *Statements* can be grouped to execute in parallel. An example of this approach is `Occam` [BEZ92], a language based on `CSP`. In an `Occam` program each line or statement corresponds to a process. The keywords `SEQ` and `PAR` allow the programmer to specify which statements execute sequentially or in parallel:

```
SEQ                 | PAR
  x := x + 1        |   x := z + 1
  y := x * x        |   y := z * z
```

  More recently, the languages `Java` and `Scala` provide a framework for parallel collections, with the statement granularity[1].

- *Objects* are analogous to processes in that they have an internal state and data. Thus, objects can be allowed to execute in parallel, and to send and receive messages. This representation is exemplified by the actors model implemented in an object-oriented language like `Scala`.

- *Functions* in pure functional languages can be executed in parallel, provided they do not have data dependency, thanks to referential transparency. Nevertheless, this approach on its own may be inefficient, since forking on small calculations produces an unnecessary overhead. Thus, constructs for explicit parallelism and explicit sequentiality may be necessary. This is the case of Parallel `Haskell` [JS08]:

```
(1)  par f g
(2)  seq f g
```

[1] Cf. `Java` 7 and the `Scala` documentation for parallel collections.

In the first expression `f` is sparked, i.e. queued to execute as a new thread, while `g` is evaluated immediately. Alternatively, the second expression forces `f` to be evaluated before `g`.

- *Logical clauses* allow the expression of parallelism due to their declarative nature. Take for instance the following program in Concurrent `Prolog`:

```
A :- B, C, D.
A :- E, F.
```

  The procedural interpretation of this expression is: to prove goal `A` it is necessary to prove subgoals `B,C` *and* `D` *or* subgoals `E` *and* F. In Concurrent `Prolog`, these clauses are also interpreted in terms of processes: each goal is a process, and each conjunction of goals forms a network of processes that communicate via shared variables. Parallelism is achieved by reducing several processes in parallel (*AND-parallelism*), or by trying in parallel each clause, in order to reduce a process (*OR-parallelism*).

- *Processes* are coarse-grained parallel tasks. They can be heavy, as in operating systems, or light like threads in many programming languages like `Java` or `Haskell`.

Note that it is possible for a programming language to combine different methods for expressing parallelism. In the case of Parallel `Haskell`, in addition to the `par` and `seq` functions, the language provides a threading control function called `forkIO` that runs an expression as a new thread. Another example is Smalltalk [GR83], an object oriented language based on message-passing: in this language parallel tasks correspond to either processes or objects.

Parallelism can be *explicit* or *implicit*. In a programming language, parallelism is said explicit if there is a construct that is directly interpreted as a parallelization in the graph of task dependencies; it is said semi-explicit if it may be interpreted as a parallelization; it is said implicit otherwise, which means that parallelism only appears at the execution level. The languages cited above generally use a sequential model as the default model. Then explicit or semi-explicit parallelism is added. For instance, `Ada`, `Scala` or Concurrent `Prolog` explicitly declare parallelism. For Parallel `Haskell`, parallelism can also be semi-explicit, as it is the

runtime that actually handles parallelism: it decides, for instance, if the expressions in a `par` function are actually worth being executed in parallel or not. Thus, the common model for parallelism is an evolution of the sequential model, which describes the default behaviors, while the parallel behaviors result from extensions, with explicit parallelism expressed using dedicated primitives.

Explicit parallelism often results in a state-space explosion, which makes programs hard to understand and to debug. Indeed, the observed behavior results from the interleaving of fine-grained atomic actions, in a non-deterministic way. To get a correct behavior, programmers need to restrict the state-space, by building critical sections from the atomic actions, while avoiding deadlocks. Implicit parallelism then appears as an alternative: compilers guess and translate implicit parallelism into explicit one at the machine level, without the involvement of programmers. However, the degree of parallelism obtained is often too weak to get significant improvements. Indeed, given a sequential algorithm, the opportunities for parallelization may be limited. A better but challenging solution is to define an equivalent parallel algorithm, if possible, since some problems do not have known efficient parallel algorithms.

Table 1.3 sums up the different forms for expressing parallelism, explicitly and implicitly. It shows the possibilities for parallelism at different levels in a programing language.

Parallelism	Implicit	Statements
		Objects
	Explicit	Functions
		Logical clauses
		Processes

Table 1.3: Parallelism in Programming Languages.

1.1.2.3 Fault Tolerance

Languages for distributed programming provide fault tolerance following two ways. They differ by the responsibility assigned to the developer.

Detection and Notification. The language run-time system detects and notifies errors but it is the programmer who handles the er-

rors with constructs provided by the language. Constructs include atomic transactions and error handling routines that are executed automatically in case of errors. Atomic transactions is the most common construct for fault tolerance. Composed with multiple operations, they have the all or nothing property: either all the operations of the transaction are executed or nothing is done. Atomic transactions are supported by languages like Argus [Lis88], in the form of *actions* and nested *sub-actions*. Sub-actions are committed, eventually performing updates, only when the parent action is committed. An example of error handling routines is SR (Synchronizing Resources) [AO93], which allows the programmer to define exception handlers that manage errors detected by the run-time.

Transparent Fault Tolerance. It is the run-time system that handle faults. To counter crash failures, some languages like Fault Tolerant Concurrent C [CGR88] allow programmers to decide which processes are to be replicated. For instance, the following line indicates that the process `master` is replicated twice:

```
create master(parameters) copies(3)
```

1.1.2.4 Implementation Pitfalls

The implementors of orchestration languages must deal with the restrictions imposed by the execution level, specially when the model for communication, synchrony and topology does not match the choices made for the language. A translation from one model to the other is possible in most of the cases. However, it is important to consider the possible effects that any translation may have on fault tolerance and performance.

Communication model. In the best scenario, the communication model at the logical level coincides with the model at the execution level. However this is not always the case. As we saw in section 1.1.1.1, a shared memory is implemented over message-passing with one of the agents taking care of the shared variables, with the drawback of compromising fault tolerance. Another form of implementation is distributed shared-memory, where the memory is distributed over the agents. Nevertheless, one of the issues of a distributed shared-memory is keeping the consistency of

the memory. Thus a consistency model is necessary to define how variables are updated and read. Implementing message-passing over shared-memory is also possible. An example are Message Oriented Middlewares (MOM). A MOM is a messaging infrastructure that handles the communication between different agents, guaranteeing the reliability of exchanges. To ensure that messages are effectively transmitted, communication with a MOM is made in two phases: send and forget, and store and forward. In the first phase the sender transmits the message to the MOM and continues with its operation without waiting for an answer. In the second phase the MOM repeatedly tries to forward the message to the receiver until it succeeds. The middleware is analogous to a shared memory space where agents deposit and retrieve messages.

Synchronization. It is possible to implement asynchronous communication over a synchronous execution level and vice-versa by specifying suitable communication protocols [CBMT96]. The flexibility of the asynchronous model eases its implementation over a synchronous execution level: buffers can be used to store messages and allow the sender to continue its execution, assuming that the buffers are big enough to avoid overflowing. Synchronous communication over asynchronous systems can be implemented by using acknowledgment messages which are sent back to the sender, which rests blocked waiting for the notification. HTTP is an interesting example to illustrate both translations. The HTTP protocol is based on a request-respond client-server model, thus on a synchronous communication. Nevertheless, Internet is built on top of an asynchronous communication protocol, the Internet Protocol (IP), whose only task is to deliver packets of data from one host to the another. Therefore HTTP relies on a transport protocol as TCP, which ensures the synchronization between the client and the server. TCP makes sure that every packet sent will arrive and that their order is preserved by using acknowledgment messages between each packet, a counter to keep track of sent packets, and timers to detect timeouts. At the same time it is possible to implement asynchronous communication over HTTP, thanks to long live connections and WebSockets, which provide a two-way communication between client and server over a single TCP connection.

Topology. One final problem, which is not often treated in literature, relates to the topology of the execution level. As we saw in section 1.1.1.1,

the topology of the system can be described by a communication graph, which can evolve as new channels between nodes are dynamically created. Imagine for instance the topology of Figure 1.3, where there are channels between A and B, and B and C; but not between A and C. The topology evolves as A sends its location to B, which then forwards it to C. Afterwards C can respond directly to A, creating a new channel between them. Nevertheless, in network configurations where some nodes

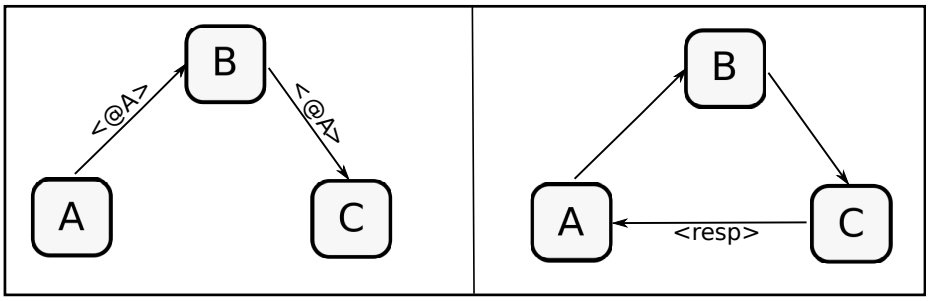

Figure 1.3: Dynamic topology

stand behind firewalls or NATs (Network Address Translators), there is no way of implementing a dynamic topology without some kind of NAT traversal solution. Approaches include specialized protocols such as the Simple Transversal of UDP Through NAT(STUN) [RWHM03], and more recently the Interactive Connectivity Establishment (ICE) [Ros10].

To summarize, implementing one model over another is never straightforward. Fault tolerance can be compromised when a shared memory is implemented over a message passing model, and vice-versa. The communication protocol becomes more complex when synchronous communication is implemented over an asynchronous execution level. The reverse direction requires buffers which add an overhead for the agents. Finally, implementing a dynamic topology in a configuration with a restricted visibility for some nodes implies an overhead on the communication as additional steps are needed in the routing of messages.

1.1.3 Service-Oriented Computing

Distributed computing takes another dimension with the emergence of Internet. Therefore it is important to make the difference between traditional distributed systems and network-based systems. In a restricted sense conforming to the tradition, distributed systems aim at emulating

the behavior of a centralized system by hiding the fact that multiple collaborating computation units work in parallel from different locations. Network-based systems, on the other hand, are implemented in a network environment, and users may be aware of this [Fie00]. Network-based systems lead to Service Oriented Computing, which we discuss in this section.

1.1.3.1 Basic Concepts

With the introduction of service-oriented computing, distributed systems evolved from monolithic structures to loose coupled organization of automated *agents* that communicate with each other by exchanging messages [HS05]. Each agent can assume the role of a *server*, providing some service to other agents or of a *client* consuming services from other agents. When it plays both roles, it becomes an *orchestrator*, consuming services and providing a new (composed) service. The *services* provided by agents are software components, available at some location in the network, that manipulate information, represented by *resources*, in response to requests. The underlying software components are considered as *black boxes*: their implementation may evolve without any functional effect over the service.

To date, there are two popular – and often antagonistic – models for service-oriented computing [PZL08]. One is based on the `WS*` standards, the other based on the `REST` architectural style, already instantiated with the `HTTP` protocol. We refer to the `WS*` model as the process-oriented model and to the `Restful` model as the resource-oriented model, respectively.

First, interoperability and integration issues have led to the development of `WS*`-services technology, mainly based on `XML` technology. Messages are exchanged between service consumers and providers in the form of `XML` documents. The operations offered by a service, along with the expected structure of messages, are defined using the *Web Service Description Language* (`WSDL`), which allows services to be independent of the underlying communication protocol. Another optional but widely used component is the *Simple Object Access Protocol* (`SOAP`), which is used to encapsulate the `XML` message, in order to separate it from other infrastructure information, like routing. However, services are not to be confused with distributed objects since there is no notion of objects, object references or factories in `WS*` [Vog03]. Upon services, orchestrators are defined with orchestration languages, like the *Business Process*

Execution Language for Web Services (BPEL), which is a standard. To establish a relationship between messages that are shared between separate processes, the BPEL specification includes the notion of *correlation sets*, which can be extended to reflect different collaboration scenarios. As processes are central in this model (WS*), we say that this model is process-oriented.

More recently, the REST paradigm has emerged as an alternative, offering light and easy to implement web services. Restful web services manipulate resources via four basic operations: create, read, update and delete, known as CRUD operations. Although REST is independent from the underlying protocol, it is usually associated with HTTP. Restful Web services return to the original design principles of the World Wide Web, and the REpresentational State Transfer (REST) architectural style formalized by Fielding [Fie00]. The REST architectural style lies on four principles.

(i) Resources can be identified with logical names. Restful web services represent these identifiers as URIs (Uniform Resource Identifiers), defined in a standard dedicated to a language for universal naming.

(ii) Resources are manipulated with a uniform interface composed of actions or methods that have a universal semantic interpretation, that is have the same meaning for all resources. In the case of Restful Web services, these actions essentially correspond to HTTP's methods PUT, GET, POST and DELETE.

(iii) Messages are self-descriptive, containing information about the purpose of the message and control data, like cachability[2]. Since resources can be represented with multiple formats (Html, XML, pdf, jpeg, etc.), messages can contain information about the expected/actual representation of the resource.

(iv) Interactions are stateless. Thus messages are self-containing: no context information is stored in the server. Therefore each message from the client contains the information required to understand the request. It is thus the responsibility of the client to keep the relationship between messages.

[2]Since clients can cache responses, messages can explicitly indicate if they can be cached or not.

Since resources are central in this model (`Restful`), we say that the model is resource-oriented.

It is possible to design a uniform model for both service models [All14, ADG+12]: it is a message-passing model. Precisely, distributed agents acting as orchestrators provide services while requiring other services that are consumed. In accordance with the black-box principle, an agent is an abstraction that hides all the implementation details: it is composed with an interface and an abstract state that evolves during its execution. The interface is composed of provided or required services. Each service is a set of channels to receive incoming messages or to send messages over the network. The execution is described by the parallel composition of agents: while updating their internal state, agents asynchronously exchange messages, without sharing memory or without synchronizing the sending and the receiving of messages in a rendez-vous. The difference between the process-oriented model (`WS*`) and the resource-oriented model (`Restful`) essentially corresponds to different decompositions of messages into a channel and a content (traditionally called a payload). In the `Restful` model, the channel describes the resource and the invoked operation, which belongs to the uniform interface, while the content describes the arguments of the operation. In the `WS*` model, the channel describes the whole service while the content describes not only the invoked operation but also its arguments. Thus the payload with `WS*` services is largely greater than with `Restful` services: it is one of the main reason why `WS*` services are qualified as heavyweight or as big. Another interesting point is that the two main requirements of the unified model, asynchronous communication and true concurrency, has led its authors [All14, ADG+12] to resort to a chemical model: the thesis directly extends the unified model as it can be considered as a concrete realization of the unified model.

Due to the existence of a unified model, all the properties described for distributed computing, and especially for message-passing models, also apply to service-oriented computing. Likewise, all the requirements for a language dedicated to distributed programming are still valid for service-oriented programming. However, there is a new requirement that we need to emphasize. Indeed, the loose-coupled nature of web services, contrary to tightly bounded systems where persistent connections are established between components, requires a mechanism for establishing relationships between the different messages exchanged in a collaboration.

Thus, *correlation*, defined as a common value shared by messages that are related, is a fundamental requirement of service-oriented computing regardless of the model.

Finally, despite of the possibility of a conceptual unified model, the diversity of models for service-oriented computing leads to new questions when it comes to the orchestration of services, questions about interoperability.

1.1.3.2 Interoperability in Service-Oriented Computing

Interoperability is the capacity of two or more systems to exchange information and to be able to use this information[3]. In service-oriented computing, interoperability is an important concern since collaborations between agents that differ in the service model they use, in their data interface or in their communication protocol are not uncommon. For instance, organizations like WS-I (Web Service Interoperability)[4] search to promote standards for the development of interoperable web services through the publication of guidelines, or profiles. However, despite these efforts, interoperability remains a challenge.

The challenge essentially consists in solving coordination problems and adaptation problems. Following the classical definition of Malone and Crowston [MC94], generally speaking, "Coordination is managing dependencies between activities", which gives in the area of programming, following Carriero and Gelernter[GC92]: "Coordination is the process of building programs by gluing together active pieces." Following the seminal paper of Yellin and Strom [YS97], adaptation aims at eliminating mismatches between software components that do not fit together.

In the following, we review research work over adaptation, in the context of service-oriented computing. The other question, coordination, is dealt with in Chapter 2. Traditionally, in the service-oriented computing field, coordination is split into two related notions, orchestration and choreography. An orchestration defines the behavior of an agent while a choreography specifies or describes from a global point of view the execution of the orchestrations involved in the collaboration of agents. As Chapter 2 presents the foundations of our orchestration language, this is the natural place to deal with coordination.

[3] According to the IEEE Standard Computer Dictionary.
[4] Cf. organization's web site.

The mismatches that adaptation aims at eliminating happen when the required and the provided services that have to be bound do not fit together. These mismatches can be classified into five general categories [BBG+06].

- Technique: for instance because of different communication protocols
- Signature: mismatch between the type of the channels
- Protocol: mismatch between the expected sequences of messages
- Concept: given ontologies describing concepts, mismatch between the concepts associated as meta-information to the services
- Quality: mismatch between quality attributes (dealing with some notion of quality of services) associated to services

Here, we focus on signature and protocol mismatches, following our programming perspective.

A natural solution to eliminate mismatches is to promote types not only for interfaces, as usual, but also for communication protocols. While interface types avoid signature mismatches, they do not guarantee the absence of protocol mismatches: components can interoperate incorrectly, since undesired deadlocks may occur. Protocol types ensures not only type safety but also deadlock freeness. Different formalisms have been proposed for extending interface types with protocol information: see for instance the review of Brogi et al. [BCP07] or more recently the state of the art report of the project Betty [Pro14]. Typically, they are based on finite automata or process calculi.

When a mismatch is detected, adaptation can be derived following two main approaches [CMP06]: the restrictive approach aims at ruling out the behaviors causing the mismatch, while the generative approach aims at defining an intermediate adaptor used to compose the mismatching behaviors.

Behavioral type systems, like session type systems [DCd10], use a restrictive approach. Yellin and Strom [YS97] initially promote the generative approach, more liberal. For instance, a solution is given by model-driven engineering techniques, as exemplified by the Starlink framework [BGR11, BGRB11]. This work in particular focuses on the problem of interoperability between protocols that have a similar functionality, for example

between SLP (Service Location Protocol) and SSDP (Simple Service Discovery Protocol). The objective is to allow the communication between components using different protocols, e.g. between a SLP client and a SSDP server. Interoperability is only possible if there exists a translation from the messages of one protocol into the other. Nevertheless, the translation in some cases is more complex than a one-to-one mapping. For example, to accomplish task A, protocol P1 requires a single message, while protocol P2 requires two messages. They are *interoperable* if a translation is possible. In order to ease the translation between two interoperable protocols, their behavior is represented by two automata representing the sequence of messages, where states are labeled according to different aspects of the communication protocol, such as message sequence, ports used, and synchrony. Then, the coordination of both automata is driven by a merge automaton, finally implemented by a middleware layer.

The main drawback of generative adaptation comes from its possible complexity: if n agents must fit with p agents, $n.p$ adaptors are required. The solution is to deploy an adaptive middleware, specially those based on messaging, like Message Oriented Middleware (MOM). The complexity can then reduce to $n + p$: it suffices to adapt each agent to the middleware. Thus, the middleware layer acts as an integration layer. Indeed, this kind of infrastructures offers several possibilities for solving interoperability problems in the form of integration patterns, where the *Message Bus* [HW03, p.137] is one of the most used. Also known as Enterprise Service Bus (ESB) [Erl09, p. 704], the message bus pattern relies on a communication component (bus) that carries messages between the connected agents, which can disconnect from the bus at any time without disrupting the functioning of the system. See Figure 1.4 for a typical architecture, where an enterprise service bus connects diverse applications and technologies through service interfaces. An enterprise service bus also requires an intermediary protocol comprising a common data model and a common command structure. Thus, messages transmitted by agents are translated into an intermediary protocol and then, translated back into the protocol used by each consumer agent. The main difficulty lies on finding the correct intermediary protocol.

Another middleware-oriented solution for interoperability problems between web services is the architecture proposed by Wang and Pazat [WP13], based on a chemical model. In this architecture the orchestration and choreography of web services is performed in an intermediate component

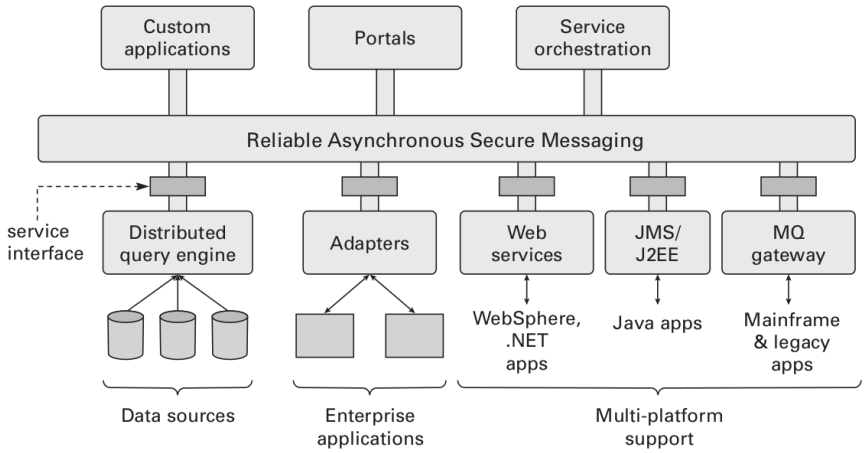

Figure 1.4: Enterprise Service Bus (Credit: MIT Press [GP09])

called the chemical middleware. Services are abstracted as chemical solutions, where floating molecules represent the meta-data of the service. Orchestrations and choreographies are written in a chemical programming language in the form of reaction rules that model the data flow between services. This work, which is closely related to our own approach, shows the interest raised by the chemical paradigm as a way of expressing orchestrations in service-oriented computing.

Finally, the preceding solutions can be considered as variations of a *pivot architecture*, a possible solution for interoperability problems, as we have shown [LGL10]. Concretely, we have proposed the implementation of some design patterns to solve interoperability problems, following a well-known trend [BBG+06]. The pivot architecture allows the implementation of such patterns by combining different orchestration languages with heterogeneous service providers around a pivot language[5].

It remains that a complete solution to the interoperability issue requires a universal representation of resources: this is the subject of the next section.

[5]The implementation of the pivot architecture with `Criojo` is further developed in Chapter 4.

1.2 Working With Data: How to Represent Resources

In the context of web services, data and operations over data can be abstracted through resources and their interfaces. In the `Restful` model, which is resource-oriented, this is clearly the case: data correspond to representations of resources while operations correspond to a fixed set of `CRUD` operations. In the `WS*` model, a service itself essentially corresponds to a specific interface for a resource (or a set of resources).

Generally speaking, anything that can be identified can be considered as a resource, including concrete objects such as documents, files, services, etc, as well as abstract concepts like the terms of an algebra[6]. Note that a resource may correspond to a temporal relationship between an identifier and a representation (value): the representation of the resource may change over time but not its semantic interpretation. Think for instance about a document in a versioning system: the identifier "latest revision" maps to a different version of the document each time the document is updated. Besides the binding between the identifier and the representation, a resource also provides an interface to manipulate the representation. As the types of the representations and the interfaces are many and varied, working with data in the context of web services requires a generic abstraction.

We now describe three prevailing forms for representing data, and the languages used to manipulate the corresponding representations.

Algebraic Model: Typically used in functional programming, it induces a recursive style for declaring types as *inductive data types* and for defining computations as recursive functions.

Relational Model: Used by most database systems, the relational model represents data types as relations and computations as relational queries.

Other Models: Resources have representations that do not only belong to the algebraic model nor to the relational model. There is no consensual term for this class of models, which is not precisely defined. In these models, data are often qualified as *semistructured data*.

[6] As defined by the URI specification.

For each selected form, after an introduction describing the way in which data are represented, we mention the main properties of the languages used to compute over these data and determine the impact of distribution over computations.

Why Objects are Out of Scope. In the preceding list, we deliberately omit objects as a possible form for representing data. At a first glance, it may seem strange as object-orientation is the prominent paradigm for programming. And indeed, in nowadays applications, a resource is generally implemented as an object (or a graph of objects). However, we are interested in *representing* resources. A representation of an object essentially corresponds to an observation: it is computed by calling a method, a pure observer if no side effects are required, which returns either a primitive data, directly observable, or an object, which is subsequently observed, leading to a recursive process. In object-oriented frameworks for web services, like `CXF`, this conversion between objects and representations is delegated to a specific component, called a data binding, like `JAXB` [McL02]. Representations are expressed as documents, written in `XML` or `Json`. From an abstract point of view, they can be considered as terms of an inductive data type. As we can consider that services can be used to make object-oriented applications interoperable, it is interesting to compare the service-oriented approach with two other standards for interoperability between distributed object-oriented applications: Common Object Request Broker Architecture (`CORBA`) [Vin97] and Remote Method Invocation (`RMI`) [WRW96]. With `RMI`, the execution environments are homogeneous, at both ends: this is the main difference with services. As a consequence, the representation is low level, since objects are transmitted in a serialized form, which in `Java` is a binary form. With services, the representation is required to be abstract, high-level. With `CORBA`, the execution environments can be heterogeneous, as with services. However, objects are passed as references, and not as values: in other words, there is no representation, but an indirection through a stub (a proxy). It means that `CORBA` implements channel mobility, where here a channel is an object reference. To get value passing instead of reference passing, it is needed to resort to a data binding to convert objects into structures before their transmission through a `CORBA` interface. With this usage, the `CORBA` technology becomes another alternative to the existing technologies for web services, like `Restful` and `WS*`.

1.2.1 Inductive Data Types

Inductive types are composite types used in functional languages like `Haskell` [HHPJW07]. To explain, let us define the type Option, declared as

```
data Option a = None | Some a
```

with two constructors `None` and `Some`, stating that an element of type `Option a` can be either `None` or `Some x`, where `x` has type `a`. Thus, type `Option a` is a sum type with two alternatives. Another classical example of an inductive type is a list, whose declaration is

```
data List a = Nil | Cons a (List a)
```

This declaration states that an element of type `List a` is either an empty list or the concatenation of a value with a list. Both alternatives are product types: the first is an empty product with zero field, the second has two fields, a value and a list. Thus, formally, an inductive data type is a sum type with one or more alternatives, where each alternative is a product type with zero or more fields. Inductive types not only allow to represent a wide range of data types, including recursive types, but also smoothly admit useful extensions, like parametric polymorphism and dependent types. The preceding examples in `Haskell` turns out to be parametrized with a type (denoted `a`). As for dependent types, which are types depending on a value, like arrays with a fixed size, they can be found for instance in the interactive theorem prover `Coq`[7], based on the calculus of inductive constructions, which actually includes inductive, polymorphic and dependent types.

Recursive Computations. To operate on inductive types, *pattern-matching* is applied to decompose a value into its alternative types and subsequent components, allowing *recursive functions* to be easily defined. Resuming our previous examples, we declare in the following example a function that removes all the empty elements from a list of type `Option`.

```
1 cleanList Nil = Nil
2 cleanList (Cons (Some a) tl) = Cons a (cleanList tl)
3 cleanList (Cons None tl) = cleanList tl
```

[7] Cf. Coq's web site.

As you can see, `cleanList` is a recursive function that uses pattern matching for operating over either an empty list (line 1), or a concatenation (lines 2 and 3). We apply again pattern matching for discriminating actual values (`Some a`) from empty ones (`None`), in order to eliminate the empty values.

Distribution and Concurrency with Inductive Data Types. In a distributed context, with a language based on inductive data types and recursive functions, it is relatively straightforward to distribute computations as shown, for instance, by the project Cloud `Haskell`, a successful extension of `Haskell` with a layer for message passing [EBJ11], implementing the actor model [HBS73]. Indeed, a functional programming language can be specialized to embed some native data binding, directly inside its type system: see the language `CDuce` [BCF03] for instance. An essential reason stands in the proximity of the data models: terms and inductive types used for computations inside agents versus documents and schemas used for communication through the network, both sides being bound by the data binding. In contrast, the data bindings for object-oriented programming languages suffer from an impedance mismatch [LM07a]. An essential reason stands in the gap between data models: observations and co-inductive types [Jac95] (instead of terms and inductive types) versus documents and schemas. Besides its main proximity, there is another one, between the functional model and the `Restful` model. Inductive data types naturally produce immutable and persistent data: immutable because their state does not change after creation, persistent because a new version of a data actually corresponds to a new data so that both versions, the old one and the new one, are available. Assume we want to implement a counter with a pure functional language (without side effects). The interface then contains a unique operation that given a natural number returns its successor. The corresponding service corresponds to a stateless service, therefore adhering to the `Restful` model. Of course, by no way, it does not mean that it is impossible to implement a stateful service with a pure functional language: for instance, with `Haskell`, it suffices to use a state monad, with the extra advantage that the statefulness of the service becomes apparent at the type level.

With a language based on inductive data types and recursive functions, concurrency and parallelism are less easy than distribution, but

are still simpler than with an imperative language using threads and locks. For instance, parallelism is only limited by fundamental data dependencies in local computations [JS08]. For instance, for local concurrent computations, the language `Haskell` uses a shared memory model. Immutable data can be copied or shared in a totally transparent way. An implicit parallelism is then induced by the data flow and the associated data dependencies. Let us take for example the expression

```
f (g(a), h(b)).
```

Since functions `g` and `h` do not have data dependencies, i.e. they operate over disjoint values, it is implied that they can be executed in parallel.

1.2.2 The Relational Model

Since its introduction by Codd in the seventies [Cod70], the relational model has been the data model that is the most used in databases to represent and manipulate information. The model is based on first-order predicate logic. The definition of a relation is based on the logical notion of a predicate: a relation is the interpretation of a predicate as a set of tuples. Thus, each tuple t in a relation R corresponds to an assertion $R(t)$, an atomic fact. Concretely, in the database relational model, relations are represented as tables with named columns where each row corresponds to a tuple. Since two rows cannot contain the same information (no duplicates), tables well correspond to the set-theoretical notion of relations. An essential property of this model lies in the possibility of normalizing any relation: the normalization process decomposes the relation into a set of relations with dependencies, in order to minimize redundancies. The main objective is "to free the collection of relations from undesirable insertion, update and deletion dependencies", as Codd said [Cod71]. Thus normalization eases data consistency to be checked and preserved.

Relational Algebra. The relational model relies on an associated relational algebra to compute over relations. The relational algebra contains a set of operators to retrieve and manage information. Retrieval operators derive from the logical and set-theoretical operations, including projection, Cartesian product, difference, union and intersection, and join. Relations can be modified by operations like insertion, updating and deletion. Based on this formal specification and possibly extending it, query

languages have been designed: they provide a useful syntax for working with databases. We present two paradigmatic languages, following two perspectives, the logical one and the engineering one.

`Datalog`. We start with the logical perspective. `Datalog` is a database query language based on logic programming [CGT89]. It is the most popular language used in deductive databases, which combine the relational model with logic programming. `Datalog` programs consist of finite sets of *ground facts* (facts without variables) and *rules*. *Facts* are assertions about the information stored in the database. A typical example of a fact is "A is an ancestor of B". Upon existing facts, *rules* express inferences that allows new facts to be deduced. For example, here is a rule using three variables (A, B, C): "If A is an ancestor of B and B is an ancestor of C, then A is an ancestor of C". It illustrates the expressive power of logical rules, which comes from recursion. More generally, rules in `Datalog` have the form

$$L_0 : -L_1, ..., L_n$$

where for any i, L_i is an atomic fact. The single fact L_0 is called the Head whereas the sequence $L_1, ..., L_n$ is called the Body of the rule. Its meaning is: from facts $L_1, ..., L_n$, deduce fact L_0. To guarantee that the set of ground facts derived from a `Datalog` program is finite, any program must satisfy two safety rules: (i) all the facts in the program must be ground, i.e. without variables, and (ii) for each rule, its head must only contain variables already present in its body. Recently, renewed attention has been brought to `Datalog`, beyond the database community. New applications include data integration, networking and program analysis [HGL11]. The trend is to use `Datalog`'s core and extend it to meed particular needs, like efficient query execution for graphs and relational structures, or the incremental maintenance of views. However, one of the major limitations of `Datalog` is its monotone semantics - the number of resources always increases during computations - which renders impossible the elimination of resources. Among the disconnected lines of research that try to solve this problem we find the works of Zaniolo et al., who extend `Datalog` with choice [GZ01] or with aggregates [WZ00], and Ganzinger and McAllester [GM02], who have allowed facts to be deleted and rules to be selected with priorities.

SQL. We now come to the engineering perspective. The language SQL is currently the query language that is the most used in databases. Its theoretical foundations lies in the relational calculus, the query language naturally associated to the relational algebra, provided that the calculus and algebra are extended with aggregate functions [Klu82]. However, the translation is left implicit: the language SQL has no formal semantics. There are some trials to remedy the situation. Negri et al. [NPS91] proposed a formalization of the 1985 standardized ANSI SQL, as a set of transformation rules from SQL into an extended three value predicate calculus (E3VPC). Contrary to the traditional two-valued logic, the E3VPC includes an unknown value for missing data. Thus, it is possible to safely apply transformations for optimization purposes. Another formalization is later proposed by Gogolla [Gog94], who translates a subset of the language into a tuple calculus. Gogolla points out the problems of specifying the semantics in plain english, like the ambiguity in some constructs like ANY and ALL. The semantics proposed is stricter with respect to the relational algebra, by forbidding duplicate rows in any query result. The formal semantics allows to prove some properties of the language, including query equivalence and the redundancy of some SQL operators.

To conclude, there is a kind of duality between Datalog and SQL, with respect to recursion and aggregation, summed up in the following table.

	Recursion	Aggregation
Datalog	✓	×
SQL	×	✓

The language Datalog makes recursion easy and aggregation difficult, and inversely for the language SQL, which for instance allows recursion either as non-standardized features or as a late extension.

Concurrency Control via Transactions. The relational data model has proven a highly effective means to share data between applications, by providing a powerful mechanism to control concurrency, based on transactions. A transaction, defined as a state transition of a database, satisfies four properties, called *ACID* [Gra81, HR83].

Atomicity: Either the transition is completely executed, or it is not: it is an all or nothing behavior. This property is directly related to concurrency control.

Consistency: A transaction preserves at the end of the transaction the invariant properties holding at the beginning.

Isolation: The execution of a transaction does not depend on the execution on any other concurrent transaction. This property is also directly related to concurrency control.

Durability: Once committed, the transaction cannot be abrogated: its effects are persistent.

Assume two transactions `A` and `B`. Atomicity and isolation induce that their concurrent execution gives one of the the following results: (i) no transition, (ii) transition defined by `A`, (iii) transition defined by `B`, (iv) transition defined by `A` followed by transition defined by `B`, (v) transition defined by `B` followed by transition defined by `A`. Therefore they correspond to a serializability condition. Consistency and durability deal with two orthogonal concerns, the safety of the transition and the correspondence between the logical and physical levels respectively.

1.2.3 Semistructured Data

With the advent of Internet, databases are now encapsulated in server applications, as a persistence tier. Hence client applications, which access data through the presentation tier, deal with data that may no more adhere to the relational model. Likewise, the growth of the memory capacity allow data to be entirely stored in the main memory instead of the file system, leading to an explosion of the possible data formats [FCP+12]. In these alternative data models, data are often qualified as *semistructured*. Initially, it meant that information about the type associated with the data may be contained within the data itself [Bun97], which allows the representation of irregular data; now, we can consider that it simply means that the data model is not relational nor algebraic. This interpretation conforms to the one of the term `NoSQL`, interpreted as Not Only `SQL`[8]: it is used to describe technologies that rely on data models that go beyond the relational model.

Models for semistructured data include key-value stores, document store, graphs, and column-family stores [SF12]. Key-value stores are hashtables where data are stored as key-value pairs, where the key identifies the value. In document stores, data are stored in documents which can

[8] See the web site dedicated to `NoSQL`.

be encoded in XML, Json, YAML, among others formats. Graph databases store data whose relations are represented by links between nodes. In column-family stores, data are stored in column-families, which are sets of rows associated to a primary key. Contrary to traditional relational databases, rows in a column-family do not need to have the same columns. This classification, however, is not strict since some models can fit into more than one category.

Models for semistructured data often use no static type or a loose type system, like Json or YAML. However, Benzaken et al. [BCNS13] show that it is possible to define a rich type system to cover standard definitions for semistructured data. The type system is based on standard structured types and on set-theoretical operations, union, intersection and difference. The idea is to be able to type a function processing a value v as follows: if v has type V_1, then return r_1 else if v has type V_2, then return r_2. Its return type is $R_1 \cup R_2$, if r_1 has type R_1 under the assumption that v has type V_1, and r_2 has type R_2 under the assumption that v has type $(V_2 - V_1)$.

Languages for Semistructured Data. Several languages have been developed to work with semistructured data. Some languages like XQuery were created to work specifically with one format; other languages like UnQL, Jaql and Linq aim at covering any possible format used in NoSQL databases.

XQuery and XPath. XQuery is a functional language for querying XML documents that uses XPath expressions to navigate through specific parts of an XML document. As its name suggests, XPath expressions define a path to a node or a set of nodes in a document. XQuery and XPath are W3C standard recommendations and have a formal semantics[9], which is based on a tree representation of XML documents. In the data model used by XQuery and XPath, each element of the tree is a node with a unique identifier[10]. There are seven kinds of nodes in the data model: document, element, attribute, text, namespace, processing instruction and comment. Upon nodes, the model specifies a set of *accessor* functions. Accessors expose the properties of nodes and are defined for every kind of node. Examples of properties are name, children,

[9] http://www.w3.org/TR/xquery-semantics/
[10] http://www.w3.org/TR/xpath-datamodel/

parent, type and kind. However for some kinds of nodes, there exist several accessors that will return an empty answer. For instance, for a node of kind document, the accessor parent will return an empty sequence. An alternative representation, based on a relational model, is proposed by Benedikt et al. [BK09] for theoretical purposes. An XML tree is a relational structure, whose signature contains three relations: one unary relation for the labels of nodes, a binary parent-child relation between nodes, and an immediate right-sibling relation between nodes. The rest of XPath accessors described in the formal data model can be deduced from these relations. Thus, essentially, a XML document is a relational structure that combines the above signature with a set of attribute functions that map nodes to values.

In addition to XPath expressions, XQuery provides a set of query expressions called FLWOR. The name is an acronym from the constructs for, let, where, order by, and return. FLWOR expressions allow, respectively, to iterate over sequences of nodes, to bind sequences to variables, to filter results on Boolean conditions, to order the result and to yield a result for each evaluated node.

One of XQuery's limitations is the lack of support for document creation or modification. An alternative is XSLT (eXtensible Stylesheet Language Transformation)[11], another W3C standard that is being developed in parallel to the XQuery-XPath suite. Like XQuery, XSLT relies on XPath expressions to transform an XML document into a new XML document, or into another format like Html. Nevertheless, the original document remains unchanged. Hence, a proposal exists to add update functionalities to XQuery called the XQuery Update Facility[12] that extends XQuery to support creation, deletion and modification of nodes in a document.

UnQL. The Unstructured Query Language (UnQL) [Bun97] is a query language for semistructured data based on structural recursion and pattern matching. Structural recursion is used to browse the data using pattern matching to follow the structure of the data. In UnQL, data is represented as trees, where a tree is an atomic value or a set of labeled trees. Trees are built with four constructors that are used for pattern

[11] http://www.w3.org/TR/xslt20/
[12] http://www.w3.org/TR/xquery-update-10/

matching in structural recursion:

Tree	t ::= t ∪ t	Union
	| {l:t}	Subtree
	| {}	Empty Tree
	| a	Leaf

Functions are defined as in ML with pattern matching:

```
fun f(T1 U T2) = f(T1) U f(T2)
  | f({L:T})   = ...
  | f({})      = {}
  | f(V)       = ...
```

As indicated, the first and third line are always the same in every program, paving the way towards a well-foundation of the recursion, so that they can be omitted:

```
fun f({somelabel:T}) = ...
  | f({L:T}) = ...
  | f(V) = ...
```

The language also includes a query of the form

```
select...where...,
```

which can be joined and nested. Queries are combined with pattern matching in the form of path patterns, which requires a certain knowledge of the structure of data. Additionally, these queries can be translated into structural recursive functions. The preceding syntax for trees is extended to cover graphs, by adding markers for input and output nodes, an output node pointing to the unique input node with the same marker. To guarantee the termination of queries over cyclic graphs, structural recursion is given two equivalent semantics: a bulk semantics, in which recursive functions are applied in parallel on all the edges of the graph; and recursive semantics using memorization of recursive calls to avoid infinite loops. Thus, UnQL can be used to query and transform XML documents as well as graph databases.

`Linq`. Initially conceived to solve the problem of impedance mismatch between the relational model and the object model, `Linq` is a query language based on the relational algebra, which according to its author, can be used with semistructured data as well [Mei11] [MBB06]. `Linq` is based on category theory, in particular monads, which leads to a generalization of collections. Thus, data is represented by collection monads and queries are expressed in terms of comprehensions, which transform collections into other collections. See for instance, the following query over Yahoo's weather service:

```
from forecast in Yahoo.WeatherService
where forecast.City == city
select forecast;
```

The language can also be seen as a generalization of relational algebra: it offers an interface for which the relational algebra is one possible implementation. Thus, multiple data sources, such as relational databases and `XML` documents, can be mapped to `Linq`. On the side of programming languages, `Linq` is integrated as an extension of the languages of the `.NET` family, that include `C#` and `VB.NET`, among others. For each implementation, `Linq` adopts the syntax of the host language, which allows the addition of operations specific to the domain targeted by the application. In this way, programmers can define specialized projection or filtering operations. Additionally, this integration allows queries to benefit from type checking over the relational data. For instance, if the query operates over a list of string, only string operations are allowed over the data processed by the query. Although `Linq` was initially created for extending `.NET` languages, other implementations have been made for `Java`, `JavaScript` and `Python`, and others. Nevertheless, one of the pitfalls of the language is the difficulty of implementing custom data providers due to the poor documentation on the parsing of the query, which changes for each host language [Ein11].

`Jaql`. The query language for `Json` (`Jaql`) [BGB+11] is a scripting language to manage `Json` documents, but also other formats like `XML` or relational databases. Its data model is based on the `Json` format, where values can be primitive values, arrays or unordered collections of name-value pairs:

```
value ::= primitive_value
```

```
| value*
| (name,value)*
```

Programs in `Jaql` are expressed as functional compositions, inspired by the Unix pipes, the output of a function being the input of the next function:

```
f1(source)->f2()->f3()
```

representing in this way data flows. The language also includes a path language: a path can be used as a function argument, allowing the navigation into passed data. `Jaql` provides built-in aggregate functions, but it is also possible to write user-defined aggregates.

Discussion. The diversity of the data models induces an analogous diversity of the languages used to compute over these models. However, among the languages described, three languages, namely `UnQL`, `Linq` and `Jaql`, try to bridge the gap between different data models by adapting themselves to different data sources and host languages. There is no precise comparison between the expressive power of these languages that ambition universality. In this direction, Benzaken et al. [BCNS13] have proposed not only a rich type system to describe the `NoSQL` data model, as said before, but also a language with filters that aims at encompassing all the constructs provided by the language `Jaql`, considered as one of the richest `NoSQL` language.

Evolution of the Concurrency Model. The `NoSQL` trend not only brings new data models but also makes the concurrency model evolve. Indeed, whereas databases were the pivotal component for integration, they have been replaced with services. In this distributed context, it is impossible to enforce for databases (or services) the following properties, known as CAP properties [GL12].

Consistency: Transactions can be serialized (atomicity and isolation of the ACID properties).

Availability : Every request receives a response.

Partition Tolerance: In presence of a partition of the network (with no communication between the parts), consistency and availability are preserved.

Another trade-off between the safety property (consistency) and the liveness property (availability) is required [Bre12, GL12]. To ensure availability, the database (as a service) can be replicated, following the cache pattern, which implies that consistency is provided as a best effort. To ensure consistency, in case of a partition failure, availability must be decreased. Another trend is to weaken the consistency property: as serializability is a strong property, it can be replaced with causal consistency, which means that the interleaving of the transactions preserves the causal ordering, or eventual consistency, which gives a convergence criterion for the replicas.

In addition to the preceding trade-off, distributed query processing is a difficult problem, much more difficult than in a centralized context [ÖV11, chap. 6]. Assume that a database is split into multiple shards, corresponding to fragments of relations. The question is to map a query over the whole database to local queries over fragments. The constraint is to optimize the usage of computing resources. Rather than dealing with a whole language like `SQL`, the `NoSQL` trend has produced some efficient solutions to distributed query processing, especially for high data volumes. A typical example is the `MapReduce` framework [DG04]. The `MapReduce` framework allows the parallelization of a job by decomposing it into a map task, which applies a function to each member of a collection, and a reduce task, which aggregates results. For instance, `Jaql` relies on the Hadoop's `MapReduce` framework.

1.3 Specification of an Orchestration Language

From the state of the art, we select the essential requirements for an orchestration language, from the point of view of distributed and service-oriented computing and from the point of view of data computing for resource manipulation. The requirements deal with the logical layer: they define the logical model associated to the language. The objective is to specify a powerful yet minimal language for the orchestration of web services that can be use as the core of a pivot architecture. Although some choices may be arbitrary, we provide a short rationale for each requirement. We use the standard terminology "must/should/may" to express obligations, recommendations and options, as defined in RFC 2119[13].

[13]Cf. IETF's web site. "Must": mandatory – "Should": optional, absence needs to be justified – "May": optional, presence needs to be justified.

As exemplified by mainstream technologies for Web services, service-oriented computing is an efficient solution to organize the exchange of messages in a network-based architecture around agents acting as servers, clients or orchestrators. Thus the language allows network-based orchestrations to be defined. Each orchestration is executed by an agent: it provides and consumes services while maintaining a local state. A service has an interface and is implemented as a set of resources. A resource has an identifier and one or more representations. The following requirements specify this overall picture with more accuracy.

1.3.1 Requirements for Service Orchestration

In the next list of requirements for the orchestration language we essentially follow the classification given in Sections 1.1.2, 1.1.3 and 1.2, however presented in a more compact form:

- communication, synchronization and parallelism,
- fault tolerance,
- services and resources.

A summary of these requirements is given in Table 1.4.

Communication, Synchronization and Parallelism. From the study of these different aspects, we propose that the logical model must satisfy the following requirements. There is a form of duality between network-wide distribution at the Internet scale and agent-wide distribution at the scale of agent's cores.

Requirement 1 (Distributed Architecture – Message Passing)**.** The language must allow orchestrations distributed between agents to be defined. It must use a *message-passing* model: agents exchange messages over *channels*.

Indeed, in the context of service-oriented computing, where systems are physically and logically scattered, there is no notion of shared memory: each agent is responsible of its own data, and data is communicated explicitly in messages. Client programs use messages to make requests to servers, which in turn response with messages.

Requirement 2 (Distributed Architecture – Asynchronous Channels)**.** The language must use *asynchronous* channels.

This requirement is natural in a context of network-wide distribution and allows any communication latency to be modeled. However, as this is the less stringent form of agent synchronization, we add the next extra requirement.

Requirement 3 (Distributed Architecture – Library of Channels)**.** The language should provide a library of channels satisfying the following synchronization properties:

(i) *synchrony*,

(ii) *preservation of the causal order*,

(iii) *broadcast*

and possibly other properties.

These channels will be implemented either over asynchronous channels, or natively, by using the channels of the physical layer.

Requirement 4 (Message Passing – Channel Scope)**.** The scope of a channel must be controlled.

The requirement allows private channels to be defined. It entails a form of location transparency. Assume for instance, that an agent externalizes an internal computation towards a slave agent providing a dedicated channel to launch the computation. The scope of the channel must be restricted to the master agent in order to avoid misuses by other agents.

Requirement 5 (Message Passing – Channel Mobility)**.** An agent must be able to transmit a channel to another agent.

Channel mobility is necessary for service discovery and dynamic routing. Indeed, during an execution, the network topology often needs to evolve: an agent needs to discover another agent that it does not know initially.

Requirement 6 (Message Passing – Scope Extrusion)**.** The scope of a channel transmitted to an agent should be extended to the receiving agent.

When a private channel is transmitted outside its scope, two behaviors are possible: either the target agent becomes able to send a message over this channel, thanks to a scope extrusion, or it does not. The first possibility, scope extrusion, should be the default behavior while the second one seems like a fault.

Requirement 7 (Agent Architecture – Shared Memory)**.** The language must provide for each agent a *shared memory* for its concurrent activities.

Locally, this requirement is natural. This means that data within the agent may be shared by its concurrent activities, for example in the case of simultaneous requests.

Requirement 8 (Agent Architecture – Locks)**.** The language must provide a primitive to lock resources.

Naturally, in an execution context where resources are shared among concurrent activities, locks are necessary in order to avoid race conditions.

Requirement 9 (Agent Architecture – Transactions)**.** The language should allow a transactional mechanism to be programmed for each agent. Transactions must satisfy:

(i) atomicity,

(ii) isolation.

By mechanism, we mean here and in the following a library, a framework, a template, or any other technique. As seen in Section 1.1.2.1, the properties to be satisfied correspond to serializability. We do not impose a specific implementation for the mechanism: the approach can be optimistic or pessimistic. Note that the requirement deals with local transactions, but it could be extended to distributed transactions.

Requirement 10 (Parallelism – Globally Explicit, Locally Implicit)**.** The definition of distributed agents acting in parallel must be explicitly stated. For each agent, the parallelism between local activities should be implicit.

We choose to explicitly define the distribution of agents, in conformance with the practice for web services: as deployment means uploading services on servers, an explicit definition of the distributed services is first

given. An alternative would be an automatic partition of a monolithic program, generating code for each agent, as in `Hop` [SB12]. The second part aims at easing the development of parallel local orchestrations: parallelization, which is difficult, should be automatically performed. Note that an agent execute concurrent activities (reception and sending of messages, local computations) and that runtime environments now provide multicores: parallelization improves performances.

Fault Tolerance We now come to fault tolerance. We limit the requirements to omission and crash faults. Thus, it is the responsibility of the programmer to ensure a correct behavior in the presence of byzantine faults: this tolerance can be enforced by security means resorting to cryptography.

Requirement 11 (Fault Tolerance – Fail-Safe)**.** The language must enforce a *fail-safe* fault tolerance preserving local invariants. It may enforce a stronger fault tolerance.

A local invariant is a property satisfied by an agent and preserved during the execution. The first part of the requirement states a minimal fault tolerance: in case of a message loss or an agent crash, each active agent still behaves safely. Beyond this minimal threshold, fault tolerance becomes costly: any extension to global safety or liveness is therefore optional.

Requirement 12 (Fault Tolerance – Detection and Notification)**.** The language should provide mechanisms for detecting and notifying omission and crash failures.

The implementation of these mechanisms depends on the underlying physical layer used to communicate. Thus the requirement may be impossible to satisfy in some scenarios due to lacking functionalities.

Requirement 13 (Fault Tolerance – Logging)**.** The language must provide mechanisms for logging events or actions.

Logging is clearly useful for recovering from errors. Keeping track of the history may allow the agent to return to a previously stable state.

Services and Resources We now come to requirements specific to services and resources.

Requirement 14 (Services – Correlation)**.** The language must provide a primitive or a mechanism for *correlating* messages.

The distributed orchestrations make agents collaborate. As many collaborations can happen simultaneously, collaborations are generally organized around sessions in order to keep a relationship between the messages exchanged in a given collaboration. A session is identified by a token, with a particular scope and lifetime: the token is generated when the session starts, and then shared between the agents participating to the session. At the end of the session, the token is no more used. For instance, many web servers provide a session token in the first interaction with the client.

Requirement 15 (Resources – Interface)**.** The language must provide a mechanism for *interfacing* with any resource.

A resource can be internal. That is to say, an artifact of the language. The requirement then corresponds to the possibility for a resource to be named and represented in the language. A resource can also be external, like a file. The requirement then aims at improving interoperability.

Requirement 16 (Resources – Representation)**.** The language must provide a *universal* data model with the following properties:

(i) data are human readable,

(ii) data are efficiently parsable,

(iii) data are serializable.

A data model is universal if all data model can be represented in it, particularly the algebraic model, the relational model and others used for semistructured data. Data need to be human readable as the interfaces of services are published, like an application programming interface (API). Data need to be efficiently parsable as they are directly integrated to the language and therefore involved in any computation. Data need to be serializable as they are communicated through the network.

Requirement 17 (Resources – Representation Typing)**.** If the language is typed, its type system may provide the set-theoretical operations union, intersection and difference, and interpret the subtyping relation as subset inclusion.

This optional requirement relies on the success reported by Benzaken et al. [BCNS13] when formalizing a general data model for semistructured data.

Requirement 18 (Resources – Computational completeness)**.** The language must be computationally complete with respect to the data model.

In other words, all function computable over the data model must be expressible in the language: this is the Church Thesis applied to the universal data model. Concretely, it means that any language defined over the data model can be translated, which can be experienced with functional, logic and imperative languages for instance. Moreover, we could add an invariance property: the translation entails a polynomially bounded overhead in time and a constant factor overhead in space.

Requirement 19 (Services – Map/Reduce)**.** The language should provide a mechanism for implementing the Map/Reduce operations.

As seen in Section 1.2.3, the Map/Reduce operations are a de facto standard for distributed computations.

1.3.2 Service Orchestration in Practice

Is there in practice an orchestration language, or equivalent, that fulfills the above requirements? To answer this question, let us analyze the requirements from the point of view of object-oriented frameworks for web-services like `CXF` and orchestration languages like `BPEL`. Table 1.5 summarizes the result of our analysis.

Popular object-oriented frameworks for web-services like `CXF` [14] provide tools to implement both `Restful` and `WS*` applications: they constitute the mainstream practice for service development. They have support for synchronous and asynchronous communication and fail-safe fault tolerance, with some detection and notification mechanisms and logging facilities. Channel mobility is only partly supported in the case of `Restful` applications, thanks to hyper-links. In the case of `WS*` services, channel mobility is a feature that was added afterwards, in the form of addressing, which is rather limited compared to the definition that we provide. Parallelism and concurrency are dealt with `Java` mechanisms, but are often hidden from the developer since generally the application container

[14] Cf. `CXF` web site. We only use `CXF` for the comparison.

Table 1.4: Requirements for Service Orchestration

		MUST	SHOULD	MAY
Communication, Synchronization and Parallelism				
Distributed Architecture	Message passing	✓		
	Asynchronous channels	✓		
	Library of channels		✓	
Message Passing	Channel scope	✓		
	Channel mobility	✓		
	Scope extrusion		✓	
Agent Architecture	Shared memory	✓		
	Locks	✓		
	Transactions		✓	
Parallelism	Globally explicit	✓		
	Locally implicit		✓	
Fault Tolerance				
	Fail-safe	✓		
	Detection and notification		✓	
Services and Resources				
Services	Correlation	✓		
	Map/Reduce		✓	
Resources	Interface	✓		
	Representation	✓		
	Typing			✓
	Computational completeness	✓		

and the database systems handle concurrent access to resources. There are indirect and poor mechanisms for correlation, which is quite left to the responsibility of the developer. There are efficient tools dedicated to the interface with resources, wrapped in `Java` code. The data model is object-oriented, being composed from specific objects. The framework embeds a data binding, allowing the translation with other data models used for serialization, with known limitations [LM07a]. For instance, type information can be lost during the translation process, as for lists, whose elements' type is erased during serialization. Lastly, the computational completeness comes from the underlying object-oriented language, `Java` for `CXF`. Thus, for instance, Map/Reduce operations can be provided as a framework. Finally, to conclude, although it turns out that an object-oriented framework for Web services like `CXF` satisfies a majority of the requirements, this solution is not really satisfactory. Indeed, object-oriented frameworks promote sequential programming, thus hiding the distributed and concurrent aspect of service orientation, which are however fundamental. Indeed, the main problem comes from an impedance mismatch between the communication model and the concurrency model. It is made manifest when considering location transparency, an expected and desirable requirement with the mobility constraints associated to the modern Web. Assume that a local shared resource is outsourced onto a remote server. The necessary concurrency control must be re-implemented in the server, at the interface between the service layer and the program layer. A standard solution is to insert a filter into a pipe of shared filters intercepting the incoming and outgoing messages: the implementation is not straightforward due to the gap between both models with two distinct scales for critical sections, at the upper message level and at the lower action level respectively. Concurrency control is also more demanding in a distributed context, as it has been acknowledged for two decades [WWWK96]: it must cover other specific aspects, like fault tolerance and security. If location transparency is required to ease programming, it must be provided with strong guarantees with respect to these distribution requirements. From the impedance mismatch, we conclude that the mainstream model for service-oriented computing is not scalable with respect to concurrency control.

Pure orchestration languages, like `BPEL`, provide better scalability properties. Based on an `XML` notation, the language `BPEL` provides a grammar for describing business processes in terms of interactions with

other processes, which translates into interactions between web services. Like frameworks for web services, many of the requirements are satisfied by this language. BPEL is based on a message-passing communication model that uses both synchronous and asynchronous message exchanges. Channel mobility is poorly supported, since BPEL is based on WS* standards. BPEL also supports explicit parallelism and concurrency as the execution of activities is represented by a flow-graph and critical sections provide concurrency control for variables. It is also possible to define transactions. Therefore BPEL is more akin of a real orchestration language. The language provides constructs to handle failure, allowing to compensate the effects of any activity that could let the process in an invalid state. It is fail-safe, since in case of failure, subsequent activities are terminated to preserve correctness. Concerning data handling, BPEL relies on the expressiveness of XPath, and XML as a data model. We can presume that it defines a computationally complete language. Ultimately, one of the biggest limitations of BPEL as an orchestration language is the lack of clarity of its theory. Moreover, being an XML dialect it can hardly be considered as a programming language meant to be used by humans.

Finally, we have not found a solution being used in practice that would be based on a formalism. In fact, a formal foundation for an orchestration language is an important requirement related to both service oriented computing and resource manipulation aspects. A formal foundation eases the design, development and use of a programming language, by providing precise and consistent specifications and tools for the correct definition and verification of orchestrations. Hence comes the interest of a language with a formal foundation shown by approaches based on declarative languages. An example of this trend is the BOOM project [ACC+10], which proposes a data-centric design style combined with a declarative language. The idea is to allow systems to be easily distributed by focusing on the state of the system and describing it in terms of collections. At the same time, a declarative language, like Datalog, describes in a natural way the behavior of the system. Other examples of this trend include languages like the join-calculus and Orc, which we discuss further in the following chapter. Our thesis is that the chemical programming fulfills the requirements that we have exposed, while giving a formal foundation to service orchestration, which we show in the next chapter.

	Frameworks `Restful` / `WS*`	`BPEL`
Asynchronous message-passing	✓	✓
Channel Library	-	-
Channel mobility	✓ / -	-
Shared memory with locks	✓	✓
Transactions	-	✓
Explicit Parallelism (locally)	-	✓
Implicit Parallelism (locally)	×	×
Explicit Parallelism (globally)	×	×
Fail-safe fault tolerance	✓	✓
Failure Detection/Notification	-	✓
Logging	-	-
Correlation	- / ✓	✓
Resource Interface	✓	✓
Universal data model	×	✓
Completeness	✓	✓
Map/Reduce	✓	×

(Yes: ✓, No: ×, Partly: -)

Table 1.5: Satisfaction of the Requirements in Practice

Chapter 2

Starting point: The Heta-calculus

In the previous chapter, we presented the basics of distributed computing and service-oriented computing as an instance of it. We finished with a set of requirements that serve as a specification for an orchestration language for services. Now we focus on our approach which starts from a formal model, namely the Heta-calculus, that underlies the implementation of a language for the orchestration of services. The usefulness of this design will be later demonstrated with a use case in a service-oriented scenario. The Heta-calculus is a calculus based on the chemical paradigm. It is an original work led within the *Ascola* team by Thesis Advisor Hervé Grall. And although its design is not a contribution of this thesis, we consider important to describe it in detail due to the strong synergy between the programming language `Criojo` and the Heta-calculus. First, the Heta-calculus provides the formal semantics of `Criojo`. Second, the presentation comes with original contributions, like the impure aspects of the Heta-calculus, that were drawn from the development of the language, the formalization of all the design decisions with respect to the requirements, and the validation against requirements.

In the following, we first review the state of the art on formal models for service-oriented computing and on the previous works that led to the design decisions that were made. Then we present the chemical calculus for service orchestration, namely the Heta-calculus. For the presentation, we follow the standard V-model.

- Requirements: see Section 1.3 "Specification of an Orchestration Language" in the previous chapter.

- Design: see Section 2.2 "Design Decisions" that recapitulates all the design decisions with respect to the requirements.

- Realization: see Section 2.3 "A Chemical Calculus for Orchestration" that defines the syntax and the semantics of the calculus.
- Validation: see Section 2.4 "Validation against Requirements".

2.1 Background

Previously, we said following Carriero and Gelernter[GC92] that "Coordination is the process of building programs by gluing together active pieces", and that in the service-oriented computing field, coordination is split into two related notions, orchestration and choreography. Actually, we adhere to the following overall picture. Agents are programmed with orchestration languages. Their communication (exchange of messages) can be specified or described by another language: when it is a specification and not a descriptive language for a semantic interpretation, we say that it is a choreography language. Carbone et al. [CHY12] show that it is possible to define an approach by synthesis: first, start from a global description, a choreography, then project the choreography to each agent involved, in order to generate the local code in the orchestration language. The synthesis can be proved safe: the collaboration between agent is safe by construction, in that it is free from deadlocks and race conditions. A weak version of the synthesis process generates orchestration types instead of orchestration programs: the safety property can be preserved if the type system is proved to be sound, in that, adapting Milner's slogan, "well-typed orchestrations cannot go wrong". In the following, we only focus on orchestration, which is the necessary first step. The reader interested in choreography can read a recent doctoral dissertation [Mon13] for latest developments.

The Heta-calculus is part of a trend of works searching to give a formal foundation to service-oriented computing. Below we first give an overview of some approaches that relate to the Heta-calculus and belong to this trend. Next, we present the works that directly influenced the design of the Heta-calculus.

2.1.1 Formal Models for Service-Oriented Computing

Many formal models have been proposed for capturing aspects of service-oriented computing with the objective of specifying, implementing or verifying properties of service collaborations. As a general rule, transition

systems are used in these approaches, either as models or as semantic domains for the languages. We present some examples of formalizations based on three standard formalisms,

- finite state automata,
- Petri nets,
- process algebras, either original or built on top of classic algebras.

The 2007 survey performed by ter Beek et al. [tBBG07] may be consulted for more references.

Finite State Automata. Finite state automata is a well-known formalism, based on transition systems, that allows to model different kind of problems including system behaviors and communication protocols. The use of automatic verification tools is rather straightforward with finite state automata, thanks to the relation with logic, which renders the model an interesting solution for formalizing and verifying service-based systems. For instance, Fu et al. propose the use of guarded deterministic automata [FBS04] to model agent behavior in composite web services: a `BPEL` process is translated into an automaton, with an input queue for messages and local variables and where transitions are equipped with guards expressed in `XPath`. Unlike other models, this approach takes into account data semantics for the verification of processes. The conversation of the processes, modeled as the composition of the automata, is then translated into `Promela`, a language to model asynchronous distributed process as deterministic automata, and then verified with the model checker `SPIN` [Hol03]. However, it does not captures channel mobility that has been included in the specification of `BPEL` as *endpoint references*. Analogous to these approaches, is the more recent proposal by Bentakouk et al. [BPZ11], using Symbolic Transition Systems (`STS`) for modeling and validating orchestration specifications written in `BPEL` or other languages like `UML` and `BPMN`. An orchestration specification is translated into `STS`, from which an execution tree is generated, allowing the extraction of some test cases. At the end, an implementation of the orchestration is validated by executing the test cases against a unit test `API` specific for web services called `SOAPUI`. The advantage of this approach over other transition systems is that complex data types used in `BPEL` specifications can be easily mapped to `STS`, allowing to explore different

levels of detail from the specification: from the signature, containing data structures and operations; to the semantics of the communication, defined by message exchanges.

Petri Nets. Introduced by C.A. Petri [Pet62], Petri nets are used to model concurrent systems with synchronous or asynchronous communication. A Petri net is a form of transition system that is represented as a directed graph with two types of nodes, one to represent places (states) and other to represent transitions. The nodes of the graph are connected by arcs that go from places to transitions or from transitions to places. The execution of the Petri net is modeled with tokens that move around the graph. For one token to move from a place to another, the transition between them must be fired. A transition can be fired when all of its input places hold tokens. A certain configuration of the net, where tokens are distributed over the places, is called a *marking*. Markings are useful to analyze properties of the Petri net, like whether certain configurations can be reached from a given initial configuration [Pet77]. Due to its similarity to flowcharts, Petri nets have been used in service-oriented computing to give formal semantics of `BPEL`'s control flow constructs, in order to analyze the properties of processes. Basically, a `BPEL` process is translated into a Petri net and then verified by a model checking tool. Open Workflow Nets, a subclass of the Petri nets model, are specially suitable for modeling web services as they explicitly define communication between nets: input and output places serve as channels that compose an interface to communicate with other nets. The asynchrony of these channels corresponds to the message-passing nature of web-service communication. One of the focuses on this respect is the controllability of web services, as exemplified by the works of Wolf [Wol09] and Massuthe et al. [MSSW08]. A service is said to be *controllable* (or operable) if it is capable of interacting with at least one partner for creating a composition that is correct, according to variable criteria like liveness, and the absence of deadlocks and livelocks. An example of a practical result of this approach is the project `Tools4BPEL` [HSS05, LMSW06], which implements methods and tools, like the model checker `Fiona`, for verifying controllability and for producing operating guides or specifications. Another example of Petri nets used to model web services is given by the research of Ouyang et al. [OVvdA+07]. In this case, control-flow constructs are analyzed, paying special attention to join-conditions and transition-conditions, to de-

tect activity unreachability and race conditions between activities that compete for the same message.

Classic Process Calculi. Process calculi are used to formally specify and verify concurrent systems. A system is represented as a set of independent agents or processes. The objective of a process calculus is to describe the interactions and synchronizations between these agents and, at the same time, to provide tools for analyzing those descriptions, formally reasoning about behavioral equivalence between processes, and proving their properties. Some examples of classic process calculi are Milner's Calculus of Communication Systems (CCS) [BBC+06, WDG+07], Hoare's Communication Sequential Processes (CSP) [Hoa85], the Language Of Temporal Ordering Specification (LOTOS) [ISO89], and the π-calculus [MPW92a, MPW92c], which extends CCS with channel mobility. Process calculi share three characteristics [Pie97]:

- Interactions between process are described as communications, rather than shared variables;
- They all use a small set of primitive operations to describe processes and systems. Usually, these operations include parallel, sequential and alternative composition [Bae05];
- From these primitive operations they derive algebraic laws for manipulating process expressions.

Thus, process calculi allow the description of web services' behavior in terms of processes, eliminating the ambiguities found in notations like BPEL. Moreover, bisimulation analysis can be used to identify a behavioral equivalence between processes in order to replace one service with another or to detect redundancy. Salaun et al. [SBS04] address the problems of service composition, with possible message loss, deadlocks, and incompatible behaviors, by specifying web services with CCS. This calculus allows the verification of the equivalence between processes with bisimulations, as well as safety and liveness properties: a CCS specification is analyzed with the CWB-NC tool, a verification workbench based on deterministic automata, and then translated into a BPEL specification. However, this approach only deals with the behavior of processes, leaving aside other aspects like data abstraction, temporal constraints, channel mobility, and asynchronous communication. Ferrara et al. [Fer04] deal with temporal

logic and data abstraction by mapping BPEL specifications to the LOTOS algebraic language, which is an ISO standard for the specification of distributed systems based on Milner's CCS and Hoare's CSP. Nevertheless, channel mobility is still not taken into account for the mapping.

Other Process Calculi. Other formalizations include original process calculi like the orchestration language Orc [KCM06], a concurrent programming language whose semantics is based on labeled transition systems. Orc provides constructs for sequential, parallel and asymmetric parallel composition of expressions. The fundamental expression in Orc is a site call, where a site can be an external process, like a web service or a service for data manipulation (called a primitive site), or a definition expression. In this model, communication between expressions occurs only in asymmetric parallel composition, where expressions execute in parallel but rest blocked in certain points when a communication needs to be completed.

Besides original calculi, another possibility is the extension of classic process calculi, as exemplified with the calculi produced by the project SENSORIA [CDNP+11]. The extensions can be classified following the main features added. As explained in the presentation of SENSORIA's calculi [CDNP+11], the prominent features deal with conversations between service callers and service callees, built over basic client-service interactions. Conversations are organized around either sessions or correlations, which aims at maintaining the links between the agents involved in the conversation.

- Session: when the conversation starts, a private channel is generated and then used to communicate.
- Correlation: the links between the agents involved in the conversation are deduced from correlations between values exchanged.

Thus, first, the Service Centered Calculus (SCC) [BBC+06, WDG+07] and its variants have been developed to represent session-based conversations. It was inspired by Orc for service composition and the π-calculus for channel mobility. Second, the calculi COWS [LPT07] and SOCK [GLG+06] have been developed to represent correlation-based conversations, following two different techniques.

These original process calculi are not disconnected from the practice since in some cases there is some relationship with the orchestration

language BPEL. For instance, the calculus proposed by Lucchi and Mazzara [LM07b] extends the π-calculus with transactions in order to provide a formal foundation for BPEL, proposing a unique event notification construct for error handling. Likewise, calculi like the PPE-calculus [KvB04], BPEL0 [PZWQ06, PZQ+06], COWS [LPT07], and more recently, Blite [LPT12], also search to simplify BPEL and to give an operational semantics to BPEL, directly relying on its specification. For instance, Blite's semantics is directly based on a subset of BPEL constructs. Thus, Blite's programs defining the behavior of service-oriented applications can be translated into BPEL. In the case of COWS, the objective is to generalize BPEL's constructs to provide a formalization that is independent from any web service technology.

To conclude, we can identify two trends in the preceding works. One is to start from an existing notation like BPEL and to give it a formal interpretation in the form of a transition system like a finite state automata or a Petri net. The other is to provide a formal notation, based on a process calculus, in order to produce applications that can be later translated into BPEL. Our approach directly relates to the second trend, in that we propose an original process calculus to formalize service orchestration. Nevertheless, the difference is that the Heta-calculus proposes a minimalist semantic framework to account for service-oriented computing. The approaches cited above either fall short of the expected requirements, as with classic process calculi, or provide a plethoric syntax to express features specific to services like correlations, sessions or compensations. We now present the origins of the Heta-calculus, which is based on Berry and Boudol's chemical abstract machine, but which is also inspired by the π-calculus and logic programming.

2.1.2 Foundations of the Heta-calculus

The Heta-calculus is partially inspired by Milner's π-calculus [MPW92b], a process calculus with a message-passing model and synchronous and asynchronous communication. The π-calculus is a continuation of CCS (Calculus of Communicating Systems), which follows the same line of thinking as Hoare's CSP: the system is represented as a set of processes that communicate by sending messages through links, or channels. The novelty introduced by the π-calculus is the notion of channel mobility[1]:

[1] Channel mobility in the π-calculus and hyperlinks in Html pages are contemporary (1991–1992): they represent exactly the same concept.

the topology of the system can change as messages carry channels, creating new links between processes. To reduce the complexity, there are only two entities in the calculus: agents and names. Thus, communication is organized around names, which identify both communication channels and variables.

If the communication model of the Heta-calculus is inspired by the π-calculus, since messages are exchanged via channels that are mobile, there is another link, through semantics: Berry and Boudol's chemical abstract machine (abbreviated as cham) [BB90]. It will come as no surprise since the semantics of the π-calculus can be formulated as a chemical abstract machine. The chemical model describes the state of a system in terms of a chemical solution, where floating molecules interact with each other, producing new molecules, according to reaction rules. Other rules, called structural rules, heat and cool the solution to decompose molecules into smaller molecules, or to compose bigger molecules from smaller ones. The effect of these rules, contrary to reaction rules, is reversible. Chemical solutions can be organized in a hierarchy as molecules can contain subsolutions enclosed in membranes. Airlocks in membranes allow communication between chemical solutions: before a reaction, a molecule moves into the airlock, to migrate to the outer solution, or in the reverse direction. Since multiple reactions can occur simultaneously, as long as the molecules involved only participate in one reaction at a time, the cham embeds a natural notion of parallelism, so that concurrent calculus like `CSP` and `CCS` can be implemented by chams. Thus, the distribution into hierarchical solutions and the natural parallelism of the chemical model offers an elegant way to formalize concurrent and distributed systems based on message passing.

However, the cham is not an effective machine. First, some structural rules may be not operationally effective, which may lead to an incorrect implementation of the semantics. To be effective, a set of structural rules should be confluent and coherent [GLP04]. Nevertheless, some of the laws used in the implementation of `CCS` lead to non-confluent heating, as proved by Garg et al. [GLP04]: this is the case for restriction and airlock locks in `CCS`. Second, the set of reaction rules can be infinite, as shown for instance with the encoding of the λ-calculus. Consider for instance the substitution operation in the λ-calculus's beta-reduction $((\lambda x.M)N \rightarrow M[N/x])$, which exhibits not only the usual pattern matching used in chemical rules but also a computation, the substitution in M of x with

N:

$$x^{-}M, N^{+} \rightarrow M[N/x]$$

The reduction implies that an external device produces an infinite number of rules: one for each tuple M, N and x, which leads to an impure aspect that is not described in the semantics.

The join-calculus [FG96] is an important step towards effectiveness. Indeed, its reflexive chemical machine extends the standard cham with the notion of locality and reflection. Thanks to locality, molecules travel directly to the location where they will react, if they match a rule. Reflection allows reactions to extend the machine with new rules, which increases the computational power of the language, whereas the domain of the values is restricted to names, as in the π-calculus. At a glance the join-calculus could be the core of physically distributed programming languages. Nevertheless, it still lacks computational completeness as we show in section 3.1.1.1, since it cannot compute all the transformations of a chemical solution.

Another attempt towards effectiveness is the γ-calculus [BFR06], an higher-order chemical calculus inspired by the λ-calculus, the calculus used to model functional languages. Its higher-order nature brings a form of reflection, rules being first-class citizens, while the presence of guards and of an inertness test brings introspection: it becomes possible to test whether a chemical solution is inert, and to make subsequent computations depend on the result of the test. This feature is fundamental for the Heta-calculus. However, the γ-calculus is more oriented towards parallelism than towards distribution.

In addition to concurrent and distributed computing, chemical models are related to logic programming. As a consequence, the Heta-calculus is also inspired by logic languages like `CHR` and linear logic.

2.1.3 Logic Programming

`CHR` is a multiset rewriting language based on a chemical model. Originally designed for writing constraint solvers, it is now used as a general purpose language [Frü08]. `CHR` is related to term rewriting systems, chemical languages, like `Gamma` [BM93] (General Abstract Model for Multiset Manipulation, the language from which the chemical model originates) and production rule systems (`OPS` [FM77]). `CHR` does not define a data type system, since it works as an embedded language that uses the data

types defined by a host language. The host language must also provide a minimum set of predefined constraints for Boolean values and equality. Such constraints are called built-in constraints. Although the traditional host language for `CHR` is `Prolog`, there also exist implementations for `Java`, `Haskell` and `C`. A `CHR` program is a set of guarded rules, which inspect and modify a constraint store. There are three kind of rules in `CHR`: simplification, propagation and simpagation rules. Simplification rules correspond to rewriting rules, producing new constraints from existing constraints that are removed; propagation rules produce new constraints in a monotonic way; and simpagation rules combine the behavior of simplification and propagation rules. Multiple extensions have been proposed to `CHR`, including sequencing with priorities for rules [DKSD07], negation as absence [WSSD06], and aggregates [SVWSD07]. In the last two extensions, introspection is used to query and accumulate information of the constraint store. Lately, the close relation between `CHR` and linear logic has inspired a new operational semantics based on the notion of persistent constraints [BRF], which solves the problem of trivial non-termination caused by propagation rules.

Linear logic [Gir87] is considered as a logic for resources because it reasons in terms of causal implications, like in the real life. Causal implications cannot be iterated since the conditions are modified after the resources are used. Assume for instance these two implications: $A \multimap B$ and $A \multimap C$. In linear logic, the meaning is akin to the following interpretation: replace A with B, and replace A with C, respectively. If A is replaced by B then the action $A \multimap C$ cannot take place: thus, linear implication is different from standard implication where premises are not consumed in a logical inference. Precisely, linear logic extends the classical logic with updates, and linear resources offer a solution to the limitations of the monotone semantics of logic languages like `Datalog` [SP08].

To conclude, the Heta-calculus is a language with channel mobility, a semantics expressed with a chemical abstract machine and using rules consuming and producing resources. All these features have a long history in programming languages, sketched above.

2.2 Design Decisions

In the section, we recapitulate all the design decisions that have been made for the Heta-calculus with respect to the requirements, following

the same structure as in Section 1.3.1. We also briefly give a rationale for the decisions. A requirement is a property that applies either to the language itself or to the programs written in the language.

Language Property: the requirement describes a property that the language must satisfy. It generally corresponds to a property that characterizes the semantics of the language.

Program Property: the requirement describes a property that some programs must be able to exhibit. It generally corresponds for the language to a syntactic construct and its semantic interpretation. The distance between the construct and the primitives of the language varies from the simple coincidence to a true implementation.

Communication, Synchronization and Parallelism. The Heta-calculus follows a message-passing model with asynchronous communication. Locally, the state of an agent is described as an abstract shared memory, in the form of a chemical soup, akin to a tuple space.

Design Decision 1 (Distributed Architecture – Message Passing)**.** Distributed agents exchange messages defined as atoms $\texttt{Msg}(k, v)$, also denoted $k(v)$, where k is a channel and v a value.

The Heta-calculus therefore follows mainstream process calculi, as described in Section 2.1.

Design Decision 2 (Distributed Architecture – Asynchronous Channels)**.** Communication is asynchronous and therefore can be decomposed in five steps:

- message production,
- message emission,
- message routing,
- message reception,
- message consumption.

These steps will lead to semantic reduction rules.

Design Decision 3 (Distributed Architecture – Library of Channels)**.** A synchronous channel is implemented by a request-response protocol with sender blocking. A channel preserving the causal order is implemented by using queues and synchronous channels, as described by Mattern and Fünfrocken [MF95]. Broadcast is implemented by a specific agent, receiving messages to be broadcasted and sending to all the expected receivers.

The library design is totally standard, aiming at being lightweight. In `Criojo`, the library can directly use the physical layer, when it provides specialized channels, for instance synchronous channels.

Design Decision 4 (Message Passing – Channel Scope)**.** Agents are either orchestrators or firewalls. There is also a root agent, containing all other agents; to allow composition, it is considered as a firewall. A firewall may contain other agents contrary to orchestrators, and filters communication. It maintains a set of provided channels, equal to the union of the channels provided by all the orchestrators inside the firewall, and a subset of private channels. Only messages over channels that are provided but not private can come through a firewall. Only messages over channels that are not provided can go out of a firewall.

The solution is very close to the current practice for services. More fundamentally, it is a way to implement the hiding operator found in the process calculus `CCS` for instance. Note also that the solution departs from a common one used for components, where there are two sets, one for publicly provided channels and the other one for required channels. With this solution, only messages over channels that are required could go out of a firewall, whereas only messages over publicly provided channels can come through a firewall. But with one-way channels and channel mobility, the set of required channels has to be updated in an inefficient way: for example, all the return channels have to become required, in all the firewall traversed.

Design Decision 5 (Message Passing – Channel Mobility)**.** A channel can be a value in a message: if $\texttt{Msg}(k, v)$ is a message, then v can contain channels.

This is the ability found in the π-calculus.

Design Decision 6 (Message Passing – Scope Extrusion)**.** When a message $\texttt{Msg}(k, v)$ goes out of a firewall W, all private channel l provided by

W and occurring in v should be removed from the set of private channels: channel l then becomes public.

This is a specific and weak implementation of scope extrusion as found in the π-calculus. It is required when the channel l is a return channel that is private.

Design Decision 7 (Agent Architecture – Shared Memory)**.** The state of each agent is described as a multiset of atoms. An atom is either a message $\texttt{Msg}(k, v)$ or an atomic fact $R(v)$, where R is a relation symbol and v a value.

The state is therefore split into two parts: the communicational one with messages, the local one with atomic facts. This is a chemical soup, as in the Chemical Abstract Machine (cham).

Design Decision 8 (Agent Architecture – Locks)**.** The state of an agent evolves according to atomic transitions. A transition consumes and produces atoms.

With the atomicity of the transitions, there is the minimal mechanism of locking found in a tuple space or in the chemical soup of a chemical abstract machine. It is sufficient, since it can easily lead to a transactional mechanism.

Design Decision 9 (Agent Architecture – Transactions)**.** The developer must implement the transactional mechanism as a specific agent wrapping a resource manager, by using standard algorithms.

The implementation idea is that dedicated atoms are used for concurrency control. For instance, assume that we follow an optimistic approach. The algorithm can be informally described as follows.

- A client asks for a commit the server managing resources (a database for instance) by sending all the information required to decide committing (typically the last versions read on the server) and a return channel.
- The server consumes the request and a specific atom meaning that the server was waiting for a commit request, and produces a specific atom (i) containing the information sent by the client and the return channel and (ii) meaning that the server deals with a commit.

- The server decides whether a commit is possible from the information sent (typically, accepts the commit if the last versions read by the client are the current versions), commits if necessary and finally sends a reply by using the return channel.
- Once the commit performed, the server produces a new atom indicating that it is again waiting for a new commit request.

Design Decision 10 (Parallelism – Globally Explicit, Locally Implicit)**.** The collaboration between distributed orchestrators is explicitly defined: hierarchy of orchestrators with firewalls, initial state and behavior of each orchestrator. The initial state of an orchestrator is defined as a multiset of atoms. The behavior of an orchestrator is specified as a set of reduction rules, each rule defining a pattern for an atomic state transition.

Globally, parallelism is explicit, thanks to the hierarchy of agents: this is a form of true parallelism, each orchestrator having its own thread of execution that can progress concurrently. Locally, the behavior corresponds to a chemical abstract machine. In particular, parallelism is implicit. Indeed, the local implicit parallelism is an essential property of the chemical abstract machines with the clear advantage of a parallelization based on a condition simple to check: a commutation property stating that two reduction rules can be executed in parallel, as long as the atoms involved only participate in one rule at a time, as in chemistry.

Fault Tolerance. The Heta-calculus concretely ensures a limited form of fault tolerance. However, the Heta-calculus allows faults to be represented in a faithful yet abstract way, as advocated by Gärtner [G̈99]. Thus the Heta-calculus can model detective and corrective mechanisms, as used for fault tolerance. This modelization is abstract and suppose an impure concretization resorting to the physical layer. Recall that we limit to omission and crash faults, leading to losses of messages and unexpected terminations of agents respectively.

Design Decision 11 (Fault Tolerance – Fail-Safe)**.** The semantics of the Heta-calculus assumes that omission and crash faults can happen. Thus, if an agent satisfies a local invariant, then the local invariant is preserved even in presence of omission and crash faults.

Actually, an omission fault is represented as an unfair routing, where a message never moves. Likewise, a crash fault is represented as an unfair execution, where some agent is never selected to be executed. This

representation implies that the absence of omission and crash faults corresponds to fairness assumptions.

Design Decision 12 (Fault Tolerance – Detection and Notification)**.** The Heta-calculus can model detectors for omission and crash failures by adding dedicated rules or instrumenting reduction rules. Upon this model, the developer must implement the mechanisms for notification by instrumenting or adding reduction rules.

More precisely, to model a crash fault, for instance, an agent can hav two specific atoms, `Active`() and `Inactive`(), used to represent the absence or the presence of the crash fault. All the reduction rules of the agent are instrumented in order to consume and produce atom `Active`(). An extra rule models a crash fault: it consumes atom `Active`() and produces atom `Inactive`(). Then the detection mechanism can be implemented: a rule consuming atom `Inactive`() models the detection. Finally, it remains to implement rules for the notification. It remains that this implementation of detection is abstract in that it is based on a model that cannot directly be implemented in a concrete language like `Criojo`: at the concrete level, the use of functionalities of the physical level is required.

Design Decision 13 (Fault Tolerance – Logging)**.** The developer must implement the logging mechanism by instrumenting the reduction rules to generate event logs and by providing the functionalities for dealing with the events generated.

The logging mechanism can be implemented with a dedicated logging agent. An event can be represented as the consumption of a molecule, that is a join of atoms. When it happens, a message is sent to the logging agent. Inside the logging agent, log events can be stocked as atoms in the chemical soup or in dedicated data structures, which will be described in the next paragraph.

Services and Resources.

Design Decision 14 (Services – Correlation)**.** The Heta-calculus allows the correlation between two messages $\texttt{Msg}(k_1, v_1)$ and $\texttt{Msg}(k_2, v_2)$ to be represented as a common value v shared by v_1 and v_2.

The decision entails two consequences:

- the values must be structured; moreover, a form of convention must exist in order to identify a component of a value as the correlation value;

- a transition must be triggerable when two correlated atoms are present, which implies that variables are not *linear* (with a unique occurrence) in patterns.

Design Decision 15 (Resources – Interface)**.** The Heta-calculus represents a resource as an orchestrator with channels defining its interface and with a multiset of atomic facts defining its state. Specifically, an external resource can also contain in addition *impure* atoms $R(v)$ and a set (possibly infinite) of reduction rules transforming each impure atom $R(v)$ into *pure* atoms.

Recall that a resource can be internal, that is an artifact of the language, or external, like a file. For an external resource, impure atoms allows an agent to be defined as a wrapper of the resource. Note that the wrapper then satisfies the *black box principle*: the implementation of impure atoms can evolve without observable effect provided that the same result is returned, in other words, provided that the (possibly infinite) set of reduction rules is still correctly implemented. This design decision leads to two languages, the pure Heta-calculus where impure atoms are banned, and the impure Heta-calculus, where impure atoms are allowed. The interest of the impure Heta-calculus comes from its capacity to model in a more direct way effects in `Criojo` programs, like Input-Output effects, but more generally to coordinate programs written in other languages.

Design Decision 16 (Resources – Representation)**.** The Heta-calculus describes the state of each agent as a relational structure defined as a multiset[2] of atoms:

- messages $\texttt{Msg}(k, v)$, where k is a channel and v a value,

- atomic facts $R(v)$, where R is a relation symbol and v a value.

The set of values v is defined as a term algebra over a signature declared in each program.

[2]This is a variant of the standard definition, which deals with sets.

The data model encompasses both the algebraic and relational models. It is very general, since relational structures are for instance the models of logic.

Design Decision 17 (Resources – Representation Typing)**.** The Heta-calculus is not typed.

An extension with a type system enriched with set operators (union, intersection, difference, as required) would be welcome but is not trivial. Indeed, as developed by Benzaken et al. [BCNS13], this type system requires the use of a semantic interpretation, as in Domain Theory.

Design Decision 18 (Resources – Computational Completeness)**.** The Heta-calculus uses *introspection* to ensure completeness. Introspection allows queries over the chemical solution associated to each agent to be defined. It essentially allows to check whether a reduction rule can be triggered.

There is a balance for the language used to define guards to be found. Too weak, the Heta-calculus is not computationally complete. Too strong, the complexity for guard evaluation is too high. Actually, computational completeness is still an open question for the Heta-calculus, which could benefit from antecedents: see for instance the work of Ganzinger and McAllester [GM02].

Design Decision 19 (Services – Map/Reduce)**.** The developer must implement the Map/Reduce operations by providing two agents, one for the Map operation, another for the Reduce operation.

Here is a design pattern that gives a general solution. Assume a function f that transforms messages. The function $\texttt{Map}(f)$ is applied to a multiset of messages m, producing another multiset of messages $f(m)$. The Reduce operator computes a result r from this latter multiset by applying a binary operation, assumed associative. It suffices to assign an agent (or several agents) to the Map operator and an agent to the Reduce operator. When a Map agent receives a message, it computes the result of the Map application and then sends the resulting message to the Reduce agent. The Reduce agent progressively reduces the received messages to produce the result.

To conclude, we have described a set of design decisions, which will be directly applied in the design of the Heta-calculus presented in the next

section. We come back to the requirements and the decisions made in Section 2.4 that deals with the validation of the design against requirements.

2.3 A Chemical Calculus for Orchestration

We now present the Heta-calculus, the formal calculus for orchestration that defines the foundations of the programming language `Criojo`. Its syntax and its semantics are based on the general framework of chemical abstract machines. We first introduce the framework, and the main innovation, introspection. We then instantiate the framework to get the Heta-calculus, with its syntax and its semantics directly inherited from the chemical framework. Finally, in the next section, we will validate the calculus against the requirements, showing that the design decisions indeed lead to their satisfaction.

Before the formalization, we start by a small example with a client and a server deployed in the web and implementing a ping-pong interaction. The server provides a channel `ping` while the client provides a channel `pong` to get the response. The server also manages a local counter: when it receives a request over channel `ping`, it sends the current value of the counter to the client and increments the counter. The initial state of the program can be described as follows.

$$\begin{array}{lll} \texttt{Web}[& \texttt{Client}[\texttt{Begin}() \,\&\, \texttt{Provided}(\texttt{pong})] & \\ \quad \& & \texttt{Server}[\texttt{Counter}(0) \,\&\, \texttt{Provided}(\texttt{ping})] &] \end{array}$$

It describes the hierarchy of agents, the web containing the client and the server, and their internal initial state. The initial state of the client contains an atom, `Begin`(), and the declaration of the provided channel `pong` whereas the initial state of the server contains an atom `Counter(0)`, giving the initial value `0` to the counter, and the declaration of the provided channel `ping`. The behavior of the client and the server is described

by the following rules.

$$
\begin{aligned}
&\texttt{Client}[\texttt{Begin}() \,\&\, S] \\
&\quad \rightarrow \texttt{Client}[\texttt{ping}(\texttt{pong}) \,\&\, \texttt{Wait}() \,\&\, S] \\
&\texttt{Client}[\texttt{pong}(N) \,\&\, \texttt{Wait}() \,\&\, S] \\
&\quad \rightarrow \texttt{Client}[\texttt{Print}(N) \,\&\, \texttt{End}() \,\&\, S] \\
&\texttt{Server}[\texttt{ping}(K) \,\&\, \texttt{Counter}(N) \,\&\, S] \\
&\quad \rightarrow \texttt{Server}[K(N) \,\&\, \texttt{Counter}(N+1) \,\&\, S]
\end{aligned}
$$

The client first sends the request to the server, then waits for the response, and finally prints the value received. The server indefinitely replies to requests by sending the value of the counter and incrementing its value. The communication rules can be defined as follows, in a generic way: they are not user-defined, contrary to the preceding rules.

$$
\begin{aligned}
&\texttt{Web}[K(V) \,\&\, M[\texttt{Provided}(K) \,\&\, S] \,\&\, S'] \\
&\quad \rightarrow \texttt{Web}[M[K(V) \,\&\, \texttt{Provided}(K) \,\&\, S] \,\&\, S'] \\
&\texttt{Web}[M[K(V) \,\&\, S] \,\&\, S'] \\
&\quad \rightarrow \neg(\texttt{Web}[M[\texttt{Provided}(K) \,\&\, S_1] \,\&\, S_1'] \rightarrow \top)\ ? \\
&\qquad \texttt{Web}[K(V) \,\&\, M[S] \,\&\, S']
\end{aligned}
$$

The first rule is used for incoming messages: if $K(V)$ is a message in the web, then it can be delivered to agent M if channel K is provided by M. Symmetrically, the second rule is used for outgoing messages: if $K(V)$ is a message in agent M, then it can be sent if channel K is not provided by M. The non-provision is expressed thanks to a control guard, an introspective mechanism.

The execution of the program produces a trace, defined as a sequence

of states starting from the initial state.

$$
\begin{aligned}
& \texttt{Web}[\ \texttt{Client}[\texttt{Begin}() \,\&\, \texttt{Provided}(\texttt{pong})] \\
& \quad \&\ \texttt{Server}[\texttt{Counter}(0) \,\&\, \texttt{Provided}(\texttt{ping})]\ ] \\
\Rightarrow\ & \texttt{Web}[\ \texttt{Client}[\texttt{ping}(\texttt{pong}) \,\&\, \texttt{Wait}() \,\&\, \texttt{Provided}(\texttt{pong})] \\
& \quad \&\ \texttt{Server}[\texttt{Counter}(0) \,\&\, \texttt{Provided}(\texttt{ping})]\ ] \\
\Rightarrow\ & \texttt{Web}[\ \texttt{Client}[\texttt{Wait}() \,\&\, \texttt{Provided}(\texttt{pong})] \,\&\, \texttt{ping}(\texttt{pong}) \\
& \quad \&\ \texttt{Server}[\texttt{Counter}(0) \,\&\, \texttt{Provided}(\texttt{ping})]\ ] \\
\Rightarrow\ & \texttt{Web}[\ \texttt{Client}[\texttt{Wait}() \,\&\, \texttt{Provided}(\texttt{pong})] \\
& \quad \&\ \texttt{Server}[\texttt{ping}(\texttt{pong}) \,\&\, \texttt{Counter}(0) \,\&\, \texttt{Provided}(\texttt{ping})]\ ] \\
\Rightarrow\ & \texttt{Web}[\ \texttt{Client}[\texttt{Wait}() \,\&\, \texttt{Provided}(\texttt{pong})] \\
& \quad \&\ \texttt{Server}[\texttt{pong}(0) \,\&\, \texttt{Counter}(1) \,\&\, \texttt{Provided}(\texttt{ping})]\ ] \\
\Rightarrow\ & \texttt{Web}[\ \texttt{Client}[\texttt{Wait}() \,\&\, \texttt{Provided}(\texttt{pong})] \,\&\, \texttt{pong}(0) \\
& \quad \&\ \texttt{Server}[\texttt{Counter}(1) \,\&\, \texttt{Provided}(\texttt{ping})]\ ] \\
\Rightarrow\ & \texttt{Web}[\ \texttt{Client}[\texttt{pong}(0) \,\&\, \texttt{Wait}() \,\&\, \texttt{Provided}(\texttt{pong})] \\
& \quad \&\ \texttt{Server}[\texttt{Counter}(1) \,\&\, \texttt{Provided}(\texttt{ping})]\ ] \\
\Rightarrow\ & \texttt{Web}[\ \texttt{Client}[\texttt{Print}(0) \,\&\, \texttt{End}() \,\&\, \texttt{Provided}(\texttt{pong})] \\
& \quad \&\ \texttt{Server}[\texttt{Counter}(1) \,\&\, \texttt{Provided}(\texttt{ping})]\ ]
\end{aligned}
$$

This simple example highlights some essential concepts:

- a program in the Heta-calculus describes a distributed orchestration, its initial state and the behavior of each agent;
- there are two kinds of atoms, atomic facts like `Begin`() and messages like `pong`(0);
- channels are also values, like `pong`;
- control guards allow a rule to be triggered after the state has been introspected.

2.3.1 Introspective Chemical Abstract Machine

The main innovation introduced by the Heta-calculus with respect to the framework defined by Berry and Boudol [BB90] for chemical abstract machines is *introspection*. Here is a description of the introspective chemical

abstract machine. The Heta-calculus is a specific language interpreted over this chemical machine.

The formal definition of the syntax is given in Table 2.1.

Value Pattern	v	$::=$	$f\ v^*$	(term)
		$\mid$	V	(variable)
Atom Pattern	a	$::=$	$R(v)$	(atomic fact)
		$\mid$	c	(cell)
		$\mid$	A	(variable)
Cell Pattern	c	$::=$	$M[s]$	(membrane with solution)
Solution Pattern	s	$::=$	$\emptyset$	(empty solution)
		$\mid$	$a \,\&\, s$	(insertion)
		$\mid$	S	(variable)
Program	p	$::=$	$c\,\{r^*\}$	(initial cell { rules })
Rule	r	$::=$	$c \to g\,?\,c$	(head $\to$ guard? conclusion)
Guard	g	$::=$	$\top$	(true)
		$\mid$	$\bigwedge g^*$	(conjunction)
		$\mid$	$\neg(c \to g)$	(control guard)

Table 2.1: Introspective Chemical Abstract Machine – Syntax

A program is defined as a set of reduction rules with an initial state. An initial state defines a (closed[3]) cell, a membrane enclosing a chemical solution. A chemical solution is interpreted as a multiset[4] of atoms, possibly empty. It is described as a pattern, a sequence of atom patterns terminated by the empty solution or by a solution variable, representing the rest of the solution. An atom is either an atomic fact or another cell. A rule $(c_1 \to g\,?\,c_2)$ defines the possible transformation of a cell: if the cell matches the cell pattern c_1 and if the guard g is satisfied, then the

[3]In the following, we omit the qualifier "closed", which means that there are no free variables. When we have an expression with free variables, we use the qualifier pattern. Example: value pattern (open value) versus value (closed value).

[4]A *multiset* is a set where each element can have multiple occurrences.

cell is transformed into the instantiation of the cell pattern c_2. Precisely, its head c_1 (on the left hand side) is a pattern containing free variables: a variable can have multiple occurrences, which allows correlation: this is a difference with the join-calculus [FG96], which entails some implementation problems as we will see in Section 3.2.2.2. The free variables in c_1 become bound in the guard g and in the conclusion c_2 on the right hand side of the rule. The guard g is a conjunction of control guards. A control guard $\neg(c' \to g')$ checks whether an hypothetic rule $(c' \to g' \,?\, \ldots)$ can trigger in the chemical solution associated to the cell matching the pattern c_1 and returns true if the rule cannot trigger and false otherwise. For instance, the guard

$$\neg(\texttt{Web}[M[\texttt{Provided}(K) \,\&\, S_1] \,\&\, S_1'] \to \top)$$

is satisfied if the agent M does not provide channel K.

The semantics is expressed through an inference system defining a non-deterministic transition relation, denoted $\Rightarrow$, and defined over cells. The transition relation allows traces to be generated for each program, starting from the initial cell declared in the program. In the semantics, to each syntactic kind corresponds a semantic kind: we obtain values, atoms, cells and chemical solutions (we use greek letters for these semantic entities). The main point is that chemical solutions are interpreted as multisets. As usual, given a pattern P and a valuation τ assigning semantic expressions (values, atoms or solutions) to variables, we denote by $P[\tau]$ the result of the substitution in P of the variables X with $\tau(X)$. A guard g is evaluated with respect to a cell γ containing atoms and a valuation τ. The domain of the valuation is equal to the set of the variables bound by the head. The control guard $\neg(c' \to g')$ is interpreted as the impossibility for a rule $(c' \to g' \,?\, \ldots)$ to be triggered. Formally, it is interpreted as the non-existence of a valuation τ' extending τ, binding all the free variables of cell pattern c' that are not bound by τ and satisfying the following properties:

(i) cell γ matches pattern c' with valuation $\tau.\tau'$ (τ extended with τ'),

(ii) and guard g' is satisfied with respect to cell γ and valuation $\tau.\tau'$.

$$(\gamma = c'[\tau.\tau']) \wedge (\gamma \models_{\tau.\tau'} g').$$

Finally, there are two generic inference rules for semantic transitions, the rule [CHEMICAL REACTION] allowing the transformation of a cell after a

matching and a satisfaction of the guard, and [MEMBRANE], allowing the transformation inside an enclosing cell.

The semantics is detailed in Table 2.2. Given a program $c\{r_0, \ldots, r_n\}$, where c is a (closed) cell and $r_0, \ldots, r_n$ are rules, it becomes possible to generate a trace: it starts with cell c and continues with cells c' resulting from transitions computed from axioms [CHEMICAL REACTION] using rules r_i and inference rules [MEMBRANE]. A program is not deterministic: multiple traces can be generated. Moreover, confluence is not required.

Value	ξ	$::=$	$f\,\xi^*$	(term)
Atom	α	$::=$	$R(\xi)$	(atomic fact)
		$\mid$	γ	(cell)
Cell	γ	$::=$	$M[\sigma]$	(membrane with solution)
Solution	σ	$::=$	$\emptyset$	(empty multiset)
		$\mid$	$\alpha \,\&\, \sigma$	(multiset insertion)

$$
\begin{array}{lllll}
\gamma & \models_\tau & \top & \stackrel{\text{def}}{\Leftrightarrow} & \top \\
\gamma & \models_\tau & \bigwedge_i g_i & \stackrel{\text{def}}{\Leftrightarrow} & \bigwedge_i (\gamma \models_\tau g_i) \\
\gamma & \models_\tau & \neg(c \to g) & \stackrel{\text{def}}{\Leftrightarrow} & \neg(\exists \tau'.(\gamma = c[\tau.\tau']) \wedge (\gamma \models_{\tau.\tau'} g))
\end{array}
$$

$$
\frac{(c_1 \to g \,?\, c_2 \in p) \quad (c_1[\tau] \models_\tau g)}{c_1[\tau] \Rightarrow c_2[\tau]} \quad \text{[CHEMICAL REACTION]}
$$

$$
\frac{\gamma_1 \Rightarrow \gamma_2}{M[\gamma_1 \,\&\, \sigma] \Rightarrow M[\gamma_2 \,\&\, \sigma]} \quad \text{[MEMBRANE]}
$$

Table 2.2: Introspective Chemical Abstract Machine – Semantics

The main differences with the standard framework [BB90] are the following.

- The chemical abstract machine is introspective, thanks to control guards. This is the major point.
- The rules are more effective since there are no reversible rules.

- The chemical solutions are described with patterns that can resort to a solution variable, with a unique occurrence, leading to a more general formulation while avoiding a complex matching.

2.3.2 Syntax and Semantics of the Heta-calculus

The syntax and the semantics of the Heta-calculus is defined via an instantiation of the chemical framework. The instantiation determines

- the signature for values, atoms and cells,
- some generic reduction rules for communication,
- the form of the reduction rules specific to each program written in the Heta-calculus.

The values are either channels, denoted k and K for channel variables, or standard terms over an algebra. In the following, the algebraic signature is left implicit. It is often tacitly assumed to contain tuple constructors, which are often omitted. For instance, we write $(v_1, \ldots, v_n)$ instead of $c_n(v_1, \ldots, v_n)$, where c_n is the constructor of n-tuple. The atomic facts are split into messages $\texttt{Msg}(k, v)$ (often simplified as $k(v)$), where k is a channel belonging to some finite given set and v a value, and standard atomic facts $R(v)$, where R is a predicate belonging to some finite given set (disjoint from the set of channels). Cells, called *agents*, are split into firewalls $W[s]$ and orchestrators $O[s]$. We assume that orchestrators are also firewalls. The firewalls that are not orchestrators can contain other agents while the orchestrators cannot contain other agents. There is also a root agent, called `Web`. Table 2.3 sums up the instantiation for syntactic elements.

The semantics of the Heta-calculus is given by the introspective chemical abstract machine. A program of the Heta-calculus contains two kinds of rules: generic rules that are common to all programs and define communication, and specific rules that define the behavior of orchestrators.

First there are two generic rules[5] for communication. These rules, presented in Table 2.4, are generic: all programs of the Heta-calculus implicitly inherit from them. Both rules express asynchronous communication. Rule `[OUT]` allows a message $K(V)$ to go out of an agent W_2

[5] Actually, they correspond to rule schemas, parametrized by the membranes occurring in them.

Channel Pattern	h	::=	k	(channel)
		|	K	(variable)
Value Pattern	v	::=	$f\ v^*$	(term)
		|	h	(channel)
		|	V	(variable)
Atom Pattern	a	::=	$\texttt{Msg}(h, v)$	(message)
		|	$R(v)$	(atomic fact)
		|	c	(agent)
		|	A	(variable)
Agent Pattern	c	::=	$W[s]$	(firewall)
		|	$O[s]$	(orchestrator)
Solution Pattern	s	::=	$\emptyset$	(empty solution)
		|	$a \,\&\, s$	(insertion)
		|	S	(variable)

Table 2.3: Heta-calculus – Syntax

`[OUT]`		$W_1[W_2[K(V) \,\&\, S_1] \,\&\, S_2]$
	$\rightarrow$	$\neg(W_1[W_2[\texttt{Provided}(K) \,\&\, S_1'] \,\&\, S_2'] \rightarrow \top)\,?$
		$W_1[K(V) \,\&\, W_2[S_1] \,\&\, S_2]$
`[IN]`		$W_1[K(V) \,\&\, W_2[\texttt{Provided}(K) \,\&\, S_1] \,\&\, S_2]$
	$\rightarrow$	$\neg(W_1[W_2[\texttt{Private}(K) \,\&\, S_1'] \,\&\, S_2'] \rightarrow \top)\,?$
		$W_1[W_2[K(V) \,\&\, \texttt{Provided}(K) S_1] \,\&\, S_2]$

Table 2.4: Heta-calculus – Semantics – Generic Rules for Communication

if the agent W_2 does not provide the channel K. Thus each agent maintains a set of atoms $\texttt{Provided}(k)$ giving all the channels provided by the orchestrators inside the firewall. Rule `[IN]` allows a message $K(V)$ to come into an agent W_2 if the agent W_2 provides the channel K as a non

private channel. Thus each agent maintains not only the set of provided channels but also a set of atoms $\texttt{Private}(k)$ giving the private channels. The set of atoms $\texttt{Provided}(k)$ and $\texttt{Private}(k)$ is initially declared when the root agent `Web` is defined: it is required that a channel that is private is also provided. In a static scenario with no mobility, these sets are not assumed to evolve. However, if the scope extrusion of channels induced by channel mobility is required, then the set of private channels may evolve, thanks to a revision of rule `[OUT]`. Assume that some channel k is able to extrude channels K': to ease matching, we assume that the message has then the form $k(K', v)$. The rule for channel k is modified as follows.

$$\begin{array}{lll} \texttt{[OUT']} & & W_1[W_2[k(K', V) \,\&\, \texttt{Private}(K') \,\&\, S_1] \,\&\, S_2] \\ & \to & \neg(W_1[W_2[\texttt{Provided}(k) \,\&\, S_1'] \,\&\, S_2'] \to \top)\,? \\ & & W_1[k(K', V) \,\&\, W_2[S_1] \,\&\, S_2] \end{array}$$

The channel K' is no longer private. For instance, after extrusion, it can be used as a response channel.

Second, each program defines a specific set of reduction rules for orchestrators. Table 2.5 gives the form of these rules, as well as the grammar generating the guards g used, given an orchestrator O.

$$g \;::=\; \top \mid \bigwedge g^* \mid \neg(O[s] \to g)$$

$$\texttt{[LOCAL]} \quad O[s_1] \to g\,?\,O[s_2]$$

Table 2.5: Heta-calculus – Semantics – Local Rules

The local rule $O[s_1] \to g?O[s_2]$ describes a transformation of the chemical solution enclosed in orchestrator O: if the enclosed chemical solution matches pattern s_1 producing valuation τ and if the guard is satisfied after an instantiation with τ, then the solution is transformed into s_2, again after an instantiation with τ. Recall also that an orchestrator contains no other agent. Therefore, in the head pattern s_1 and the conclusion pattern s_2, and in the head patterns s of the control guards, we can find patterns for messages or for atomic facts but not for agents. Note

that if a message pattern $k(v)$ occurs in head pattern s_1, it is reasonable to assume that k is a provided channel: otherwise, rule `[OUT]` and the local rule `[LOCAL]` could both consume the same message, which is a bit counter-intuitive. However, in the current version of the Heta-calculus, we do not define syntactic constraints to ensure these reasonable properties: the programmer needs to take care of them. Local rules can be classified into three main categories, depending on the use of the unique possible solution variable.

Cleaning rules: these rules are used to remove all the atoms except some finite specific multiset. A typical use is to clean a chemical solution, once the result has been computed.

Form:

$$O[a_1 \,\&\, \ldots \,\&\, a_m \,\&\, S] \to g \,?\, O[b_1 \,\&\, \ldots \,\&\, b_n \,\&\, \emptyset],$$

where $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$ are atom patterns, S the solution variable and $\emptyset$ the empty multiset.

Conversion rules: these rules are used to convert a finite specific multiset into another one.

Form:

$$O[a_1 \,\&\, \ldots \,\&\, a_m \,\&\, \emptyset] \to g \,?\, O[b_1 \,\&\, \ldots \,\&\, b_n \,\&\, \emptyset],$$

where $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$ are atom patterns, and $\emptyset$ the empty multiset.

Standard rules: these rules are used for standard computations where some finite specific multiset is consumed and another finite specific multiset is produced, the remaining atoms being preserved.

Form:

$$O[a_1 \,\&\, \ldots \,\&\, a_m \,\&\, S] \to g \,?\, O[b_1 \,\&\, \ldots \,\&\, b_n \,\&\, S],$$

where $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$ are atom patterns, and S the solution variable.

With these both sets of rules, for communication and for computation respectively, it is easier to evaluate the determinism of the language than in the general framework of introspective chams. Thus, with simple disciplines for the definition of provided and private channels, routing can become deterministic. Inside an orchestrator, local rules are still non

deterministic. However, critical pairs are necessarily pairs of chemical solutions, so that there exists a particular simple criterion to ensure local confluence. For instance, if a critical pair comes from two rules involving disjoint sets of messages and atomic facts during their evaluation, then it is joinable, by a simple commutation, which implies local confluence. This criterion can also be used for parallelization, beyond confluence: both rules can also be triggered in parallel. For instance, rules defined in distinct orchestrators can be executed in parallel: this is a form of true parallelism. The question is deepened later, in Section 2.4.2.

To terminate, we develop an instructive example showing that it is possible to encode the inequality of variables occurring in the head. An equality between variables is simply represented by using a unique variable for all the equal variables. Consider the following rule using an inequality between two variables occurring in the head.

$$O[s_1(V,W)] \to g \wedge (V \neq W) \,?\, O[s_2].$$

We seek to replace it by rules following the definition that we have given, namely without inequalities in guards. The pair (V,W) of variables in pattern $s_1(V,W)$ just means that the pattern uses variables V and W. Let $\mathtt{D}$ be an unary relation symbol, used to denote the domains of V and W. We can initialize $\mathtt{D}$ as follows.

$$O[s_1(V,W)] \to (g \wedge \neg(O[\mathtt{D}(V) \,\&\, S] \to \top)) \,?\, O[\mathtt{D}(V) \,\&\, s_1(V,W)]$$
$$O[s_1(V,W)] \to (g \wedge \neg(O[\mathtt{D}(W) \,\&\, S] \to \top)) \,?\, O[\mathtt{D}(W) \,\&\, s_1(V,W)]$$

These rules add $\mathtt{D}$ atoms in the solution, without duplicates and with no other effect. Then it suffices to extend the head to enforce the inequality.

$$O[\mathtt{D}(V) \,\&\, \mathtt{D}(W) \,\&\, s_1(V,W)] \to g \,?\, O[s_2]$$

Indeed, since for a given V, there is at most one atom $\mathtt{D}(V)$, we can deduce that if the solution matches the head pattern with valuation τ, then $\tau(V) \neq \tau(W)$.

For instance, a binary relation $\mathtt{A}$ can be decomposed into a diagonal relation $\mathtt{R}$ and an irreflexive relation $\mathtt{I}$. The decomposition could be simply defined as follows, with inequalities in guards.

$$O[\mathtt{A}(X,Y) \,\&\, S] \to (X \neq Y) \,?\, O[\mathtt{I}(X,Y) \,\&\, S]$$
$$O[\mathtt{A}(X,X) \,\&\, S] \to \top \,?\, O[\mathtt{R}(X,X) \,\&\, S]$$

The translation gives the following program.

$$O[\mathtt{A}(X,Y)\,\&\,S]\to(\neg(O[\mathtt{D}(X)\,\&\,S']\to\top))\;?\;O[\mathtt{A}(X,Y)\,\&\,\mathtt{D}(X)\,\&\,S]$$
$$O[\mathtt{A}(X,Y)\,\&\,S]\to(\neg(O[\mathtt{D}(Y)\,\&\,S']\to\top))\;?\;O[\mathtt{A}(X,Y)\,\&\,\mathtt{D}(Y)\,\&\,S]$$
$$O[\mathtt{A}(X,Y)\,\&\,\mathtt{D}(X)\,\&\,\mathtt{D}(Y)\,\&\,S]\to\top\;?\;O[\mathtt{I}(X,Y)\,\&\,S]$$
$$O[\mathtt{A}(X,X)\,\&\,S]\to\top\;?\;O[\mathtt{R}(X,X)\,\&\,S]$$

Assume that initially orchestrator O contains four atoms

$$\mathtt{A}(0,0),\mathtt{A}(1,1),\mathtt{A}(0,1),\mathtt{A}(1,0).$$

After two reductions with the last rule, we obtain the following solution.

$$\mathtt{R}(0,0),\mathtt{R}(1,1),\mathtt{A}(0,1),\mathtt{A}(1,0).$$

The last two rules cannot apply with the present solution, contrary to the first two rules, which gives two atoms `D`.

$$\mathtt{R}(0,0),\mathtt{R}(1,1),\mathtt{A}(0,1),\mathtt{A}(1,0),\mathtt{D}(0),\mathtt{D}(1).$$

The third rule can now apply, which gives the following solution.

$$\mathtt{R}(0,0),\mathtt{R}(1,1),\mathtt{I}(0,1),\mathtt{A}(1,0).$$

After a new generation of both atoms `D`, the third rule can again apply, which gives the final solution.

$$\mathtt{R}(0,0),\mathtt{R}(1,1),\mathtt{I}(0,1),\mathtt{I}(1,0).$$

2.4 Validation against Requirements

Let us now revisit the requirements of Section 1.3 by validating them against the previous definition of the Heta-calculus.

2.4.1 Distribution and Concurrency

We review requirements dealing with distribution and concurrency: distributed architecture and its message passing model, agent architecture with its shared memory model, and fault tolerance.

2.4.1.1 Global Message Passing Architecture

By definition, communication in the Heta-calculus conforms to a *message-passing* model: agents communicate by exchanging atoms that represent messages. Communication is *asynchronous* since messages go out and into an agent in an independent way. The ping/pong example at the beginning of the section, along with the definition of the rules `[IN]` and `[OUT]`, serve as illustration. Thus, by construction the Heta-calculus satisfies the requirement of asynchronous message-passing. Nevertheless, a *library of channels* needs yet to be completed in the resulting implementation.

Library of Channels. To illustrate how to construct channels with different synchrony properties over existing asynchronous channels, let us consider the simple example of a synchronous channel k. Assume an orchestrator using channel k with rules

$$O[s_1] \to g \,?\, O[s_2].$$

We describe a sequence of transformations allowing synchrony for k. First, we assume two relation symbols `Active` and `Wait` used to describe the state of the agent: when the agent is in state `Wait`, it is waiting for an acknowledgement over channel `ack` related to a message sent over k, otherwise, it is in state `Active`. The rules become

$$O[\texttt{Active}() \,\&\, s_1] \to g \,?\, O[\texttt{Active}() \,\&\, s_2].$$

Second, the conclusions are modified: each occurrence of a message $k(v)$ in s_2 is replaced with an atomic fact $\texttt{Send}(k, v)$: we get s_2'.

$$O[\texttt{Active}() \,\&\, s_1] \to g \,?\, O[\texttt{Active}() \,\&\, s_2'].$$

Third, a rule is added to describe the transition from `Active` to `Wait` when a message over k is sent.

$$O[\texttt{Active}() \,\&\, \texttt{Send}(k, v) \,\&\, S] \to O[\texttt{Wait}() \,\&\, k(v, \texttt{ack}) \,\&\, S].$$

Fourth, the reverse transition is added, when the acknowledgement message is received.

$$O[\texttt{Wait}() \,\&\, \texttt{ack}() \,\&\, S] \to O[\texttt{Active}() \,\&\, S].$$

Fifth, the agent providing channel k transforms the rule for asynchronous communication

$$O'[k(v) \,\&\, s_1] \to g \,?\, O'[s_2].$$

into a rule for synchronous communication, by adding an acknowledgement:

$$O'[k(v, K) \,\&\, s_1] \to g \,?\, O'[K() \,\&\, s_2].$$

Channel Mobility and Extrusion. By construction, the Heta-calculus provides *channel mobility*, *channel scope*, and *scope extrusion*. First, channels are also values and can be transmitted inside messages. Then, as seen in the definition of rules `[IN]` and `[OUT]`, agents have predicates `Private` and `Provided` to control the scope of channels: messages can only come into the agent throughout provided channels that are not private. Nevertheless, channels may evolve from a private status (element of `Private`) to a public status (no more element of `Private`) if we assume the revised rule `[OUT']` for scope extrusion. However, this form of scope extrusion differs from the one in the π-calculus: it constitutes a weak, but sufficient, form since name conflicts after extrusion are possible. To avoid name conflicts, a naming discipline is required, for instance the one associated to `URI`s.

Let us revisit and refine the initial example, the ping-pong interaction, by adding scope extrusion. The initial state now declares the return channel `pong` as private, whereas in the introductory example, all the channels were assumed to be public.

$$\begin{aligned} \texttt{Web}[\;\; & \texttt{Client}[\texttt{Begin}() \,\&\, \texttt{Provided}(\texttt{pong}) \,\&\, \texttt{Private}(\texttt{pong})] \\ \& \;\; & \texttt{Server}[\texttt{Counter}(0) \,\&\, \texttt{Provided}(\texttt{ping})] \;\;\;] \end{aligned}$$

The reduction rules are not modified.

$$\begin{aligned} & \texttt{Client}[\texttt{Begin}() \,\&\, S] \\ & \quad \to \texttt{Client}[\texttt{ping}(\texttt{pong}) \,\&\, \texttt{Wait}() \,\&\, S] \\ & \texttt{Client}[\texttt{pong}(N) \,\&\, \texttt{Wait}() \,\&\, S] \\ & \quad \to \texttt{Client}[\texttt{Print}(N) \,\&\, \texttt{End}() \,\&\, S] \\ & \texttt{Server}[\texttt{ping}(K) \,\&\, \texttt{Counter}(N) \,\&\, S] \\ & \quad \to \texttt{Server}[K(N) \,\&\, \texttt{Counter}(N+1) \,\&\, S] \end{aligned}$$

The communication rules are modified, precisely by adding scope extrusion to rule `[OUT]` for the outgoing message `ping(pong)` which gives the

following rule.

$$\begin{array}{l}\texttt{Web}[\texttt{Client}[\texttt{ping}(\texttt{pong}) \,\&\, \texttt{Private}(\texttt{pong}) \,\&\, S] \,\&\, S'] \\ \quad \to \neg(\texttt{Web}[\texttt{Client}[\texttt{Provided}(\texttt{ping}) \,\&\, S_1] \,\&\, S_1'] \to \top) \,? \\ \qquad \texttt{Web}[\texttt{ping}(\texttt{pong}) \,\&\, \texttt{Client}[S] \,\&\, S']\end{array}$$

Thus, the atomic fact $\texttt{Private}(\texttt{pong})$ is consumed, the channel `pong` becoming public. The rule `[IN]` for the response $\texttt{pong}(N)$ is instantiated as follows.

$$\begin{array}{l}\texttt{Web}[\texttt{pong}(N) \,\&\, \texttt{Client}[\texttt{Provided}(\texttt{pong}) \,\&\, S] \,\&\, S'] \\ \quad \to \neg(\texttt{Web}[\texttt{Client}[\texttt{Private}(\texttt{pong}) \,\&\, S_1] \,\&\, S_1'] \to \top) \,? \\ \texttt{Web}[\texttt{Client}[\texttt{pong}(N) \,\&\, \texttt{Provided}(\texttt{pong}) \,\&\, S] \,\&\, S']\end{array}$$

Initially, the control guard evaluates to false, because of the presence of atom $\texttt{Private}(\texttt{pong})$, but after extrusion, it evaluates to true, enabling the reduction rule.

2.4.1.2 Local Shared Memory Architecture

The Heta-calculus satisfies the requirements related to agent architecture: *shared memory* and *locks*. The chemical solution enclosed in an agent corresponds to a shared memory where resources are shared between the rules. Since transitions are atomic, locking can be modeled with the consumption of a specific resource. Concerning *transactions*, they can be easily implemented in the Heta-calculus. In the following example, instead of the optimistic approach sketched in Section 2.2, we give a shorter solution with a simple server managing a resource represented as an atom $\texttt{Val}(V)$. The aim is to define a transaction composed of a `read` operation followed with a `write` operation. Initially, the state of the server `S` is

$$\texttt{S}[\texttt{Provided}(\texttt{read}) \,\&\, \texttt{Provided}(\texttt{write}) \,\&\, \texttt{Val}(0)].$$

Its rules are defined as follows.

$$\begin{array}{c}\texttt{S}[\texttt{read}(K) \,\&\, \texttt{Val}(V) \,\&\, X] \to \top \,?\, \texttt{S}[K(V) \,\&\, X] \\ \texttt{S}[\texttt{write}(V) \,\&\, X] \to \top \,?\, \texttt{S}[\texttt{Val}(V) \,\&\, X]\end{array}$$

Thanks to the consumption of atoms (here $\texttt{Val}(V)$), it is therefore easy to implement a transaction with a lock.

2.4.1.3 Fault Tolerance

The Heta-calculus enforces a *fail-safe* fault tolerance, since the loss of messages or the crash of an agent can be directly modeled in an execution. It is also possible to model detectors. For instance, by declaring two atoms `Active` and `Inactive` to represent the state of the agent, we can detect an agent failure through a transition from `Active` to `Inactive` when a crash fault occurs. Every rule of the agent

$$O[s_1] \to \top \,?\, O[s_2]$$

becomes

$$O[\texttt{Active}() \,\&\, s_1] \to \top \,?\, O[\texttt{Active}() \,\&\, s_2].$$

An additional rule models a crash fault:

$$O[\texttt{Active()} \,\&\, S] \to \top \,?\, O[\texttt{Inactive}() \,\&\, S]$$

For message loss, a detector can be modeled in a firewall, for instance with a rule like this.

$$W[K(V) \,\&\, S] \to \top \,?\, W[\texttt{Loss}(K, V) \,\&\, S].$$

Then some other rules can use atom $\texttt{Loss}(K, V)$ to notify agents, to log the omission error, and so on, in a straightforward manner.

2.4.2 Parallelism

The semantics of the Heta-calculus is not parallel. However, as we have already seen, there is a particularly simple criterion for parallelization: two rules involving disjoint sets of atoms during their evaluation commute, and can even be executed in parallel. As this is always the case for two rules belonging to two distinct agents, especially orchestrators, we can consider that *parallelism* is explicit at the global level with the hierarchy of agents; on the contrary, it is implicit at the orchestrator level as some analysis is required.

It remains that this important analysis for parallelization inside an orchestrator presents some issues. First, is it easy to generalize to any number of rules, beyond two? Second, what is the impact of introspection? Indeed, introspection requires to consider the whole process of evaluation, and not only the consumption of head atoms.

To answer these questions, we formalize the criterion: to the best of our knowledge, it has never been formalized for chemical machines, but belongs to folklore. We prove the generalization and use a more abstract statement in order to easily deal with introspection.

A Labeled Transition System. Given a program written in the Heta-calculus, we consider the transition system defined over cells with the transition relation $\Rightarrow$. Actually, we limit ourselves to an orchestrator and its enclosing firewall, which is the problematic case. We can label each transition with a pair whose first component is the reduction rule applied and second component is the valuation applied. Recall that the reduction rule applied is

- either a rule instance of the generic rule schema `[IN]` for incoming messages,
- a rule instance of the generic rule schema `[OUT]` for outgoing messages,
- or a specific rule `[LOCAL]` for a local reduction in an orchestrator.

For instance, if the semantic transition $\gamma_1 \Rightarrow \gamma_2$ is the axiom

$$\frac{(h \to g \,?\, c \in p) \quad (c_1[\tau] \models_\tau g)}{c_1[\tau] \Rightarrow c_2[\tau]} \quad [\texttt{CHEMICAL REACTION}]$$

then the transition is labeled with

$$(h \to g \,?\, c, \tau).$$

Consider a transition $\gamma_1 \overset{(r,\tau)}{\Rightarrow} \gamma_2$, where r is a local rule and τ a valuation. We denote γ_2 by $\gamma_1\langle(r,\tau)\rangle$. Likewise, the composition of two transitions (or more) is expressed with a semi-colon: $\gamma_1\langle(r_1,\tau_1);(r_2,\tau_2)\rangle$ represents the cell obtained after two transitions, labeled with (r_1,τ_1) and (r_2,τ_2) respectively. When the transitions commute, we write: $\gamma_1\langle(r_1,\tau_1)|(r_2,\tau_2)\rangle$ for the final cell. More generally, we write $\gamma_1\langle(r_1,\tau_1)|\ldots|(r_n,\tau_n)\rangle$ to mean that there exists a cell γ_{n+1} such that for all permutation ρ, we have:

$$\gamma_{n+1} = \gamma_1\langle(r_{\rho(1)},\tau_{\rho(1)});\ldots;(r_{\rho(n)},\tau_{\rho(n)})\rangle.$$

For the transition $\gamma_1 \overset{(r,\tau)}{\Rightarrow} \gamma_2$, there exists a solution σ^W and atom solutions σ_1 and σ_2 such that $\gamma_1 = W[O[\sigma_1] \,\&\, \sigma^W]$ and $\gamma_2 = W[O[\sigma_2] \,\&\, \sigma^W]$, where

O is the orchestrator considered and W its enclosing firewall. We can now define the consumption $H_{r,\tau}$ and the production $C_{r,\tau}$ of atoms due to the triggering of rule r by the following equation:

$$\sigma_2 = \sigma_1 - H_{r,\tau} \uplus C_{r,\tau}.$$

The operation $A - B$ is equal to the multiset difference, where the multiplicity of each element is equal to the difference of its multiplicities from A to B; we assume that B is included in A, which means that the multiplicity of each element in B is less than its multiplicity in A. The operation $A \uplus B$ is equal to the multiset union, where the multiplicity of each element is equal to the sum of its multiplicities in A and B. Of course, the above equation allows multiple solutions, and at least one, (σ_1, σ_2): we need to define a way to select a pertinent solution. A standard reduction rule has the form

$$O[a_1 \,\&\, \ldots \,\&\, a_m \,\&\, S] \to g \;?\; O[b_1 \,\&\, \ldots \,\&\, b_n \,\&\, S],$$

where $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$ are atom patterns, and S a solution variable. The associated transition becomes

$$O[H \uplus \tau(S)] \Rightarrow O[C \uplus \tau(S)],$$

where τ is the valuation involved, and H and C are multisets of atoms, defined as follows:

$$\begin{aligned} H &= (a_1 \,\&\, \ldots \,\&\, a_m \,\&\, \emptyset)[\tau], \\ C &= (b_1 \,\&\, \ldots \,\&\, b_n \,\&\, \emptyset)[\tau]. \end{aligned}$$

The pair (H, C) is a natural solution. But often, some atoms a_i are preserved, which means that we have some equalities $a_i[\tau] = b_j[\tau]$. The associated transition becomes:

$$O[H' \uplus P \uplus \tau(S)] \Rightarrow O[C' \uplus P \uplus \tau(S)],$$

where H', C' and P are multisets of atoms with $H' \cap C' = \emptyset$. We now get two natural solutions: (H', C') and $(H' \uplus P, C' \uplus P)$. As we will see below, the best choice with respect to the criterion for parallelization is to select the smallest multisets, that is (H', C'): thus, the pair $(H_{r,\tau}, C_{r,\tau})$ will always represent the smallest solution, with respect to inclusion. It also satisfies $H_{r,\tau} \cap C_{r,\tau} = \emptyset$.

In the following, we limit ourselves to standard reduction rules. We do not consider the case of cleaning or conversion rules: indeed, they are rarely parallelizable since they remove the remaining of the solution (corresponding to $\tau(S)$).

Commutation and Parallelization. We can now define a binary relation between rules to express that these rules are parallelizable. Actually, it is better to define the relation between rule instances, that is labels (r, τ), where r is a rule and τ a partial valuation. The extended definition allows the same rule to be parallelizable: it is useful for a rule that is parametrized with a session identifier. For instance, consider the rule

$$O[\mathtt{req}(K, A, S) \,\&\, \mathtt{State}(V, S) \,\&\, X] \\ \to \top \,?\, O[K(f(V, A), S) \,\&\, \mathtt{State}(g(V, A), S) \,\&\, X].$$

Variable S represents the session identifier. Agent O maintains a state $\mathtt{State}(V, S)$ for session S. The agent replies to the request by using channel K, a value $f(V, A)$ computed from state V and input A, and the session identifier passed in the request, and updates the state associated to the session with $g(V, A)$. Clearly, several instances of the rule can be executed in parallel, provided that the session identifiers are pairwise distinct. In that case, the relation will be defined between rule instances $(r, (S \mapsto s))$, where r is the rule and $(S \mapsto s)$ the valuation assigning s to S.

We say that rule instances (r_1, τ_1) and (r_2, τ_2) *commute*, or are *commutable*, if for all cell γ and for all valuations τ_1' and τ_2', we have the following property:

> if there are two transitions from γ labeled with $(r_1, \tau_1.\tau_1')$ and $(r_2, \tau_2.\tau_2')$ respectively,
>
> $$\gamma \Rightarrow \gamma\langle(r_1, \tau_1.\tau_1')\rangle \\ \gamma \Rightarrow \gamma\langle(r_2, \tau_2.\tau_2')\rangle$$
>
> then
>
> **Commutativity:** the transitions compose and commute:
>
> $$\gamma\langle(r_1, \tau_1.\tau_1'); (r_2, \tau_2.\tau_2')\rangle = \gamma\langle(r_2, \tau_2.\tau_2'); (r_1, \tau_1.\tau_1')\rangle.$$

The relation expresses a commutativity property, which already implies local confluence. But actually it also implies a disjointness property, which paves the way towards parallelization. We say that rule instances (r_1, τ_1) and (r_2, τ_2) are *disjoint*, if for all cell γ and for all valuations τ_1' and τ_2', we have the following property:

if there are two transitions from γ labeled with $(r_1, \tau_1.\tau_1')$ and $(r_2, \tau_2.\tau_2')$ respectively,

$$\gamma \Rightarrow \gamma\langle(r_1, \tau_1.\tau_1')\rangle$$
$$\gamma \Rightarrow \gamma\langle(r_2, \tau_2.\tau_2')\rangle$$

then

Disjointness: the associated consumptions are disjoint.

$$H_{r_1,\tau_1.\tau_1'} \cap H_{r_2,\tau_2.\tau_2'} = \emptyset.$$

We have the following implication between both previous properties.

Proposition 2.4.1 (Commutativity implies Disjointness)**.** Assume that (r_1, τ_1) and (r_2, τ_2) commute. Then they are disjoint.

Proof. Consider a cell γ and valuations τ_1' and τ_2' such that there are two transitions from γ labeled with $(r_1, \tau_1.\tau_1')$ and $(r_2, \tau_2.\tau_2')$ respectively. Let $H_1 = H_{r_1,\tau_1.\tau_1'}$, $H_2 = H_{r_2,\tau_2.\tau_2'}$, $C_1 = C_{r_1,\tau_1.\tau_1'}$ and $C_2 = C_{r_2,\tau_2.\tau_2'}$. Let $I = H_1 \cap H_2$. Let H_1' and H_2' be multisets satisfying $H_1 = H_1' \uplus I$ and $H_2 = H_2' \uplus I$. The transitions can be written for some R

$$H_1' \uplus I \uplus H_2' \uplus R \Rightarrow C_1 \uplus H_2' \uplus R$$

and

$$H_1' \uplus I \uplus H_2' \uplus R \Rightarrow H_1' \uplus C_2 \uplus R.$$

Assume that I is not empty. After the first transition, we have

$$C_1 \uplus H_2' \uplus R = C_1 \uplus H_2' \uplus I \uplus R'$$

for some sub-multiset R' of R, since $H_1 \cap C_1 = \emptyset$, hence $I \cap C_1 = \emptyset$ and the second transition must be possible. Thus, the first rule must express the presence of two I, and not only one I. Likewise for the second rule, by symmetry. It means that R' must contain I, and so on, which is a contradiction. Finally, I is empty: the sets H_1 and H_2 are disjoint. □

Without introspection, that is with rules without guards (precisely with guards always true), the commutativity property is equivalent to the disjointness property. With introspection, that is with rules with control guards, the equivalence is no more valid. A typical example of non

commutativity is the production of an inhibitor for the second transition, as in chemistry. For instance, the rules

$$\begin{aligned} O[\mathtt{A()} \,\&\, S] &\rightarrow \top \,?\, O[\mathtt{B()} \,\&\, S] \\ \text{and } O[\mathtt{C()} \,\&\, S] &\rightarrow \neg(\mathtt{B()} \,\&\, X \rightarrow \top) \,?\, O[\mathtt{D()} \,\&\, S] \end{aligned}$$

are disjoint but do not commute. Finally, the impact of introspection is limited to this non-equivalence: disjointness needs to be replaced with commutativity, as a criterion for parallelization, as shown by the following proposition. In the following, we denote the solution associated to a cell γ by $\gamma.\Sigma$:

$$O[\sigma].\Sigma = \sigma.$$

Proposition 2.4.2 (Elementary Parallelization)**.** Assume that (r_1, τ_1) and (r_2, τ_2) commute.
For all cell γ, for all valuations τ_1' and τ_2',
if there are two transitions from γ labeled with $(r_1, \tau_1.\tau_1')$ and $(r_2, \tau_2.\tau_2')$ respectively,

$$\begin{aligned} \gamma &\Rightarrow \gamma\langle(r_1, \tau_1.\tau_1')\rangle \\ \gamma &\Rightarrow \gamma\langle(r_2, \tau_2.\tau_2')\rangle \end{aligned}$$

then

Parallelization: the final solution can be computed from the initial transitions as follows:

$$\begin{aligned} &\gamma\langle(r_1, \tau_1.\tau_1')|(r_2, \tau_2.\tau_2')\rangle.\Sigma \\ &\quad = \gamma.\Sigma - (H_{r_1,\tau_1.\tau_1'} \uplus H_{r_2,\tau_2\tau_2'}) \uplus (C_{r_1,\tau_1.\tau_1'} \uplus C_{r_2,\tau_2.\tau_2'}). \end{aligned}$$

Proof. Due to commutativity, the final solution is equal to

$$(\gamma.\Sigma - H_{r_1,\tau_1.\tau_1'} \uplus C_{r_1,\tau_1.\tau_1'}) - H_{r_2,\tau_2\tau_2'} \uplus C_{r_2,\tau_2.\tau_2'},$$

which gives the intended solution thanks to the disjointness property. □

Generalization. Finally we show the generalization, that is parallelization for n rule instances.

Theorem 2.4.3 (Parallelization)**.** Consider a family of rule instances $\big((r_i, \tau_i)\big)_i$. Assume that for all distinct i and j, we have that (r_i, τ_i) and (r_j, τ_j) commute.

For all cell γ, for all valuations $(\tau'_i)_i$,
if for all i, there is a transition from γ labeled with $(r_i, \tau_i.\tau'_i)$,

$$\gamma \Rightarrow \gamma\langle(r_i, \tau_i.\tau'_i)\rangle$$

then

Commutativity: all the transitions compose and commute,

$$\gamma \Rightarrow^n \gamma\langle(r_1, \tau_1.\tau'_1)|\ldots|(r_n, \tau_n.\tau'_n)\rangle$$

Parallelization: the final solution can be computed from the initial transitions as follows:

$$\begin{aligned}&\gamma\langle(r_1, \tau_1.\tau'_1)|\ldots|(r_n, \tau_n.\tau'_n)\rangle.\Sigma\\ &\quad= \gamma.\Sigma - (H_{r_1,\tau_1.\tau'_1} \uplus \ldots \uplus H_{r_n,\tau_n.\tau'_n}) \uplus (C_{r_1,\tau_1.\tau'_1} \uplus \ldots \uplus C_{r_n,\tau_n.\tau'_n}).\end{aligned}$$

Proof. Assume that for all distinct i and j, we have that (r_i, τ_i) and (r_j, τ_j) commute. Hence, they are disjoint. Let γ be a cell, and $(\tau'_i)_i$ a family of valuations. Assume that for all i, $\gamma \Rightarrow \gamma\langle(r_i, \tau_i.\tau'_i)\rangle$. By disjointness, we have: for all distinct i and j, $H_{r_i,\tau_i.\tau'_i} \cap H_{r_j,\tau_j.\tau'_j} = \emptyset$.

First, it is straightforward to prove by induction over n that

$$\gamma\langle(r_1, \tau_1.\tau'_1); \ldots; (r_n, \tau_n.\tau'_n)\rangle$$

is well-defined. Just use the assumption that any pair of rule instances (r_i, τ_i) commute, which entails well-definedness.

Second, to prove commutativity, we use the fact that any permutation can be decomposed into a sequence of elementary transpositions (involving one and only one exchange of contiguous values). For an elementary transposition, the result is trivial, since any pair of rule instances (r_i, τ_i) commute.

Third we show by induction over the cardinal n property P_n:

$$\gamma\langle R_1|\ldots|R_n\rangle.\Sigma = \gamma.\Sigma - (H_1 \uplus \ldots \uplus H_n) \uplus (C_1 \uplus \ldots \uplus C_n),$$

where $R_i = (r_i, \tau_i.\tau'_i)$, $H_i = H_{R_i}$ (atoms consumed) and $C_i = C_{R_i}$ (atoms produced).
P_1 is trivial, P_2 is true by assumption (commutativity) and by the preceding proposition (parallelization).
Assume $n > 2$. Assume the Inductive Hypothesis: for all $m < n$, P_m.

P_{n-2} gives:

$$\gamma\langle R_1|\ldots|R_{n-2}\rangle.\Sigma = \gamma.\Sigma - (H_1 \uplus \ldots \uplus H_{n-2}) \uplus (C_1 \uplus \ldots \uplus C_{n-2}).$$

P_{n-1} gives:

$$\gamma\langle R_1|\ldots|R_{n-2}|R_{n-1}\rangle.\Sigma = \gamma.\Sigma - (H_1 \uplus \ldots \uplus H_{n-2} \uplus H_{n-1}) \uplus (C_1 \uplus \ldots \uplus C_{n-2} \uplus C_{n-1})$$

and

$$\gamma\langle R_1|\ldots|R_{n-2}|R_n\rangle.\Sigma = \gamma.\Sigma - (H_1 \uplus \ldots \uplus H_{n-2} \uplus H_n) \uplus (C_1 \uplus \ldots \uplus C_{n-2} \uplus C_n).$$

We deduce:

$$\begin{aligned}&\big(\gamma - (H_1 \uplus \ldots \uplus H_{n-2}) \uplus (C_1 \uplus \ldots \uplus C_{n-2})\big)\langle R_{n-1}\rangle.\Sigma \\ &\quad = \gamma.\Sigma - (H_1 \uplus \ldots \uplus H_{n-2} \uplus H_{n-1}) \uplus (C_1 \uplus \ldots \uplus C_{n-2} \uplus C_{n-1})\end{aligned}$$

and

$$\begin{aligned}&\big(\gamma - (H_1 \uplus \ldots \uplus H_{n-2}) \uplus (C_1 \uplus \ldots \uplus C_{n-2})\big)\langle R_n\rangle.\Sigma \\ &\quad = \gamma.\Sigma - (H_1 \uplus \ldots \uplus H_{n-2} \uplus H_n) \uplus (C_1 \uplus \ldots \uplus C_{n-2} \uplus C_n),\end{aligned}$$

which can be rewritten as follows:

$$\begin{aligned}&\big(\gamma - (H_1 \uplus \ldots \uplus H_{n-2}) \uplus (C_1 \uplus \ldots \uplus C_{n-2})\big)\langle R_{n-1}\rangle.\Sigma \\ &\quad = \big(\gamma.\Sigma - (H_1 \uplus \ldots \uplus H_{n-2}) \uplus (C_1 \uplus \ldots \uplus C_{n-2})\big) - H_{n-1} \uplus C_{n-1}\end{aligned}$$

and

$$\begin{aligned}&\big(\gamma - (H_1 \uplus \ldots \uplus H_{n-2}) \uplus (C_1 \uplus \ldots \uplus C_{n-2})\big)\langle R_n\rangle.\Sigma \\ &\quad = \big(\gamma.\Sigma - (H_1 \uplus \ldots \uplus H_{n-2}) \uplus (C_1 \uplus \ldots \uplus C_{n-2})\big) - H_n \uplus C_n,\end{aligned}$$

thanks to assumptions $H_i \cap H_j = \emptyset$ and H_i included in $\gamma.\Sigma$. Since (r_{n-1}, τ_{n-1}) and (r_n, τ_n) commute, we deduce by the preceding proposition

$$\begin{aligned}&\big(\gamma - (H_1 \uplus \ldots \uplus H_{n-2}) \uplus (C_1 \uplus \ldots \uplus C_{n-2})\big)\langle R_{n-1} \mid R_n\rangle.\Sigma \\ &\quad = \big(\gamma.\Sigma - (H_1 \uplus \ldots \uplus H_{n-2}) \uplus (C_1 \uplus \ldots \uplus C_{n-2})\big) - (H_{n-1} \uplus H_n) \uplus (C_{n-1} \uplus C_n)\end{aligned}$$

then, thanks to assumptions $H_i \cap H_j = \emptyset$ and H_i included in $\gamma.\Sigma$,

$$\begin{aligned}&\big(\gamma - (H_1 \uplus \ldots \uplus H_{n-2}) \uplus (C_1 \uplus \ldots \uplus C_{n-2})\big)\langle R_{n-1} \mid R_n\rangle.\Sigma \\ &\quad = \gamma.\Sigma - (H_1 \uplus \ldots \uplus H_{n-2} \uplus H_{n-1} \uplus H_n) \uplus (C_1 \uplus \ldots \uplus C_{n-2} \uplus C_{n-1} \uplus C_n)\end{aligned}$$

and finally by transitivity P_n. □

We will see later how to concretely apply the criterion for parallelization inside orchestrators. Its interest comes from the reduction that it performs: indeed it involves a sequence of verifications for pairs of rule instances, instead of verifications for all their subsets. Thus the verification entails a complexity in n^2, instead of 2^n, if there are n rule instances. Note however that standard inference rules to optimize are not valid. Thus the binary relation "commutable", which is symmetric and (essentially) irreflexive[6], is not transitive. For instance, the rules

$$\mathtt{O}[\mathtt{A}() \,\&\, X] \to \top \,?\, \mathtt{O}[\mathtt{B}() \,\&\, X] \quad \text{and} \quad \mathtt{O}[\mathtt{C}() \,\&\, X] \to \top \,?\, \mathtt{O}[\mathtt{D}() \,\&\, X]$$

commute, likewise the rules

$$\mathtt{O}[\mathtt{C}() \,\&\, X] \to \top \,?\, \mathtt{O}[\mathtt{D}() \,\&\, X] \quad \text{and} \quad \mathtt{O}[\mathtt{A}() \,\&\, X] \to \top \,?\, \mathtt{O}[\mathtt{B'}() \,\&\, X],$$

but the rules

$$\mathtt{O}[\mathtt{A}() \,\&\, X] \to \top \,?\, \mathtt{O}[\mathtt{B}() \,\&\, X] \quad \text{and} \quad \mathtt{O}[\mathtt{A}() \,\&\, X] \to \top \,?\, \mathtt{O}[\mathtt{B'}() \,\&\, X],$$

do not commute. In the same vein, the complement of the relation "commutable" is not transitive.

2.4.3 Services and Resources

Values in the Heta-calculus can be structured and hence can contain keys to correlate messages. Then, *correlation* is supported, thanks to equalities that can be expressed in heads. A good example is the case of a server agent that serves multiple clients:

$$W[\mathtt{Server}[\mathtt{Provided}(\mathtt{op}) \,\&\, \mathtt{Session}(0)] \,\&\, \mathtt{ClientA}[s_a] \,\&\, \mathtt{ClientB}[s_b]].$$

For each client the server keeps a `Session` variable that serves to correlate the messages in a conversation with a particular client:

$$\begin{aligned}&\mathtt{Server}[\mathtt{op}(K) \,\&\, \mathtt{Session}(V) \,\&\, s_1]\\ &\quad \to \top \,?\, \mathtt{Server}[\mathtt{Corr}(V, K) \,\&\, \mathtt{Session}(V+1) \,\&\, s_2]\\ &\mathtt{Server}[\mathtt{Corr}(V, K) \,\&\, \mathtt{Result}(V, X) \,\&\, S] \to ?\, \mathtt{Server}[K(X) \,\&\, S]\end{aligned}$$

In this example, the result, once computed, is correlated with the request, in order to send the response. The Heta-calculus also provides a way to

[6] Actually, for special rule instances r, for instance the identity rule, we can have: r and r commute.

implement *Map/Reduce* operations. We propose an example showing how to encode these operations. Consider we want to perform the following computation: for any integer in a stream, multiply it by 2 and then sum all the results. Here is the program, without representing the clients.

$$\texttt{Web}[\texttt{Map}[\texttt{Provided}(\texttt{twice})] \,\&\, \texttt{Reduce}[\texttt{Provided}(\texttt{sum}) \,\&\, \texttt{Result}(0)]]\{$$
$$\texttt{Map}[\texttt{twice}(V) \,\&\, S] \to \top \,?\, \texttt{Map}[\texttt{sum}(2 * V) \,\&\, S]$$
$$\texttt{Reduce}[\texttt{sum}(V) \,\&\, \texttt{Result}(V') \,\&\, S] \to \top \,?\, \texttt{Reduce}[\texttt{Result}(V + V') \,\&\, S]\}$$

We alleviate the definition by omitting the arithmetic computations, directly given in the conclusions. A client then sends the integers in the stream over channel `twice`.

The Heta-calculus provides a universal model for *representing* resources since the chemical solution associated to an agent is akin to a relational structure, as expected.

Additionally, the calculus allows to *interface* with any resource. For an internal resource, the Heta-calculus exactly follows the design decision given in Section 2.2: an interface of channels, a state as a relational structure (a multiset of atoms precisely). For an external resource, the interfacing principle is to use impure relations R producing impure atoms $R(v)$, whose reduction is defined by a possibly infinite set of reductions of the following form.

$$O[R(v) \,\&\, S] \to \top \,?\, O[s]$$

Thus, atom $R(v)$ can be interpreted as a call to an external native function. All the useful information is passed through the argument v. Impure atoms are useful not only to model impure effects (side effects) but also to embed another language into the Heta-calculus. For further examples, see Section 3.1.3.

The *typing* requirement is trivially satisfied since the Heta-calculus is untyped. The extension with a type system using standard set operations is foreseen.

Finally, for verifying *computational completeness* we show that introspection increases the expressive power. To express the computability limitation for standard chams, we need some terminology. By a standard cham, we mean a cham without introspection, in other words a cham where rules have only a guard that is true. Without loss of generality, we only consider one agent (an orchestrator, with no communication). Relations are arbitrarily split into two classes, the class of observable relations

and the class of unobservable ones. A *transformation* is a binary relation over multisets of atoms built from symbols of observable relations. A transformation T is *computable* by a standard cham if there are

(i) a finite set of local rules

$$O[s_1] \to \top \; ? \; O[s_2]$$

and

(ii) a multiset σ_1^- of initial atoms built from symbols of unobservable relations

such that for all multiset σ_1^+ in the input domain of the transformation T, we have:

(i) for all multiset σ_2^+ associated to σ_1^+ by T, there exists an execution starting from the solution $\sigma_1^- \uplus \sigma_1^+$ and terminating with the solution $\sigma_2^- \uplus \sigma_2^+$, where σ_2^- is some multiset of final atoms built from symbols of unobservable relations, and

(ii) all execution starting from the solution $\sigma_1^- \uplus \sigma_1^+$ terminates, with a final solution $\sigma_3^- \uplus \sigma_3^+$, where σ_3^- is some multiset of final atoms built from symbols of unobservable relations and where (σ_1^+, σ_3^+) belongs to T.

We can now formally specify the following limitation: a standard cham cannot compute a cloning transformation. We denote $\langle\, R()^p \,\rangle$ the multiset containing p occurrences of atom $R()$.

Proposition 2.4.4 (Clone Problem)**.** Given a symbol R of an observable relation with arity zero, the transformation T equal to

$$(\langle\, R()^n \,\rangle, \langle\, R()^{2n} \,\rangle)_{n \in \mathbb{N}}$$

cannot be computed by a standard cham.

Proof. Suppose for a contradiction that there exists a standard cham computing transformation T. Let n be a natural number. There exists an execution starting from $\sigma_1^- \uplus \langle\, R()^n \,\rangle$ and terminating with $\sigma_2^- \uplus \langle\, R()^{2n} \,\rangle$. Then since the guards are always true, by applying the chemical reactions and by maintaining apart an atom $R()$, we deduce an execution starting

from $\sigma_1^- \uplus \langle\, R()^{n+1} \,\rangle$ and reaching $\sigma_2^- \uplus \langle\, R()^{2n+1} \,\rangle$. The last solution cannot be final by assumption. Hence there exists a rule that can be triggered. If the rule consumes less than $2n+1$ atoms $R()$, then $\sigma_2^- \uplus \langle\, R()^{2n} \,\rangle$ cannot be a final solution, contradiction. Therefore, for each n, there exists a rule that consumes $2n+1$ atoms $R()$. This is a contradiction, since the cham has a finite set of rules. □

Introspection increases the expressive power since we can solve the clone problem.

Proposition 2.4.5 (Clone Problem Revisited)**.** Given a symbol R of an observable relation with arity zero, the transformation T equal to

$$(\langle\, R()^n \,\rangle, \langle\, R()^{2n} \,\rangle)_{n\in\mathbb{N}}$$

can be computed by an introspective cham.

Proof. Consider the following program in pure Heta-calculus.

$$O[\texttt{One}() \,\&\, R() \,\&\, S] \to \top \;?\; O[\texttt{One}() \,\&\, C() \,\&\, C() \,\&\, S]$$
$$O[\texttt{One}() \,\&\, S] \to \neg(O[R() \,\&\, S'] \to \top) \;?\; O[\texttt{Two}() \,\&\, S]$$
$$O[\texttt{Two}() \,\&\, C() \,\&\, S] \to \top \;?\; O[\texttt{Two}() \,\&\, R() \,\&\, S]$$
$$O[\texttt{Two}() \,\&\, S] \to \neg(O[C() \,\&\, S'] \to \top) \;?\; O[\texttt{Three}() \,\&\, S]$$

Then its execution starting from the local solution $\langle\, \texttt{One}(), R()^n \,\rangle$ always terminates with the solution $\langle\, \texttt{Three}(), R()^{2n} \,\rangle$, for any n. □

To conclude, the Heta-calculus satisfies all the requirements in a satisfactory way: Table 5.1, in Conclusion, provides an overview. We now come to practice, by presenting an implementation of the Heta-calculus.

Chapter 3

Criojo: the Heta-calculus made concrete

This chapter is dedicated to Criojo, the language that aims at concretizing the theory presented in Chapter 2. Criojo allows the definition of agent behaviors via a set of rules, according to a general schema. From this starting point, we open perspectives into two directions. From the developer point of view, we present a tutorial explaining how to program with Criojo: essential features, like program definitions, introspection, interfacing with external resources, modularity and composition, distribution, are explained by examples. The part culminates with the presentation of examples that show that Criojo encompasses four paradigms for programming: concurrent, functional, sequential and logical. From the implementer point of view, we discuss the major challenges for the implementation and we explain our choices. We describe how to translate the Heta-calculus into a concrete programming language. Then we describe implementation details of our prototype. The overall picture is as follows. From a given description, rules are generated. Then, the generated rules are executed on a chemical machine. There are several possibilities for the implementation of the language, as having a compiler or interpreting on a virtual machine, or both, depending on the abstraction level of the virtual machine. Our choice for the current implementation in Scala, was to describe the rules directly in the host language in the form of an internal domain-specific language (DSL), which was the best option for a prototype, and to implement the chemical machine as an interpreter.

3.1 Programming with Criojo in Scala

In this section we present the basic functionalities of Criojo with some examples. As we go through the examples, we informally revisit the semantics of the Heta-calculus, which is the theory behind Criojo.

Criojo is implemented as a Scala API with an *internal Domain Specific Language* (internal DSL), also called *embedded language*. An internal DSL is a kind of mini-language created within an existing host language, by using a subset of its grammar, and adding new features to the host language without actually modifying it. The principal advantage of an internal DSL is that it does not need a compiler, so the language implementor can focus on the implementation of the semantic features of the embedded language without syntactic concerns. However, internal DSLs are somehow limited by the programming model of the host language: the type system, the syntactic constructs available.

In this section we begin by showing how to define the behavior of agents with Criojo. Next, we show how distribution allows Criojo agents to communicate and collaborate. We finish with a set of more complex examples that show the completeness of Criojo.

3.1.1 Defining Agent Behavior

An agent is an independent computational unit, whose state is represented by a chemical *solution* and whose behavior is defined as a set of reaction rules. Resources in the agent are represented in terms of a *relational structure*: a predicate applied to terms expresses an atomic fact and defines extensionnally a *relation*, considered as a multiset. Besides standard atoms, there are other atoms, messages, built from specific relations (and corresponding to the relation Msg parametrized with a channel, $\texttt{Msg}(k, -)$, in the Heta-calculus). The multiset of atoms within the agent constitute its state in the form of a chemical solution. The state of the agent is modified by reaction rules that generate new atoms by consuming existing atoms from the solution. Some of the new atoms stays in the local solution, while others are exported, depending on the type of relation that defines them. There are two types of relations in a Criojo agent: *Local Relations*, which are used only internally within the agent; and *Channels*, which allow the communication with other agents, by transporting messages from one agent to another. Details of agent communication through channels are further explained in the next section.

The following is an example of a simple Criojo agent, call authAgent, that validates, based on her registry status, wether an user has the right to access a resource in a Web application. A user status can be Guest, Registered or Admin. Resources can be of public access, restricted to registered members, or restricted to administrators. The initial state of

the program contains the user status and the resource access type. The program produces a Ok atom, containing the id of the user and the id of the resource if the user has the right to access the resource.

```
1  val authAgent = new Agent(){
2    val Guest, Admin, Registered = Rel[UUID]
3    val PublicAccess, MembersAccess, AdminAccess = Rel[UUID]
4    val PublicResource, MembersOnlyResource, AdminResource = Rel[UUID]
5    val Ok = Rel[UUID, UUID]
6    val uid, rid = Var[UUID]
7
8    rules(
9      Guest(uid) --> PublicAccess(uid),
10     Registered(uid) --> (PublicAccess(uid) & MembersAccess(uid)),
11     Admin(uid) -->
12           (AdminAccess(uid) & PublicAccess(uid) & MemberAccess(uid)),
13
14     (PublicResource(rid) & PublicAccess(uid)) --> Ok(rid, uid),
15     (MembersOnlyResource(rid) & MemberAccess(uid)) --> Ok(rid, uid),
16     (AdminResource(rid) & AdminAccess(uid)) --> Ok(rid, uid)
17   )
18 }
```

First, the snippet shows how agents are created by instantiating the class Agent, from the Criojo API. In line(2), we declare three local relations Guest, Registered and Admin of type UUID, a utility type used for unique identifiers [1]. Then, in line (4), we declare two UUID variables uid, and rid. The rules in lines (7-14) consist of a pattern at the left side of the arrow and a join of atoms on the right, forming the conclusion. For example, the rule in line (8) can be read as follows: every time the solution contains an atom Registered matching the left pattern, that atom will be consumed in order to produce new atoms PublicAccess and MemberAccess, indicating that a user of type Registered has access to public resources and to resources reserved for members. The rest of the rules in lines (12-14) compare the resource's access restriction against the user access rights generated in lines (7-9): a public resource can be accessed by any one

[1] http://en.wikipedia.org/wiki/Universally_unique_identifier

with public access, an so on. Note a difference with the Heta-calculus: the solution variables are omitted in the rules. Note also that the program produces no result in the case of a user with insufficient rights for accessing a resource, since none of the rule applies. In fact, in order to provide a failure response it is necessary to create a rule that verifies the absence of the corresponding access atom, hence the importance of introspection.

We now present three fundamental features of Criojo: *introspection* for expressing all possible transformations of the solution, *adaptability* for interacting with external components, and *modularity* for reusing rules.

3.1.1.1 Agent Introspection

Thanks to guards, Criojo offers introspection: an agent is capable of introspecting its own state. Thus, the execution of a rule (where the agent is left implicit)

$$s_1 \to g \,?\, s_2$$

depends on the satisfaction of its guard g. When the guard is `True`, it can be omitted. The pattern s_1 binds variables in g and s_2. If the variables in s_2 are all bound by the binder pattern s_1, it is not the case for guard g. Variables in g can also be bound by the left pattern of a guard control occurring in g. The syntax of guards in Criojo is based on that of the Heta-calculus, with the addition of some syntactic sugar:

Guard	g	::= `True`	
		$\mid g$ && g	(and)
		$\mid g \parallel g$	(or)
		$\mid$ `Not`$(s \to g)$	(control)
		$\mid$ `Abs`(s)	(absence)

Indeed, the *or* and *absence* guards are not present in the definition of the Heta-calculus. The guard `Abs`(s) is a shortcut for `Not`$(s \to$ `True`$)$, which verifies the existence of the molecule s in the solution. The *or* guard simulates having two identical rules with disjoint guards. In other words, a rule $s_1 \to (g_1 \| g_2) \,?\, s_2$ could be translated into

$$s_1 \to g_1 \,?\, s_2 \text{ and } s_1 \to g_2 \,?\, s_2.$$

In the following example, we produce the clone of a relation, using the absence guard to test the absence of a molecule in the solution, guaranteeing in this way the termination of the program.

```
1 (One() & R(x)) −−> (One() & S(x) & S(x)),
2 One() −−> Abs(R(x)) ?: Two(),
3 (Two() & S(x)) −−> (Two() & R(x))
```

In this program rules 1,2 and 3 are executed in sequence, without loops. In the first phase, Rule 1 transforms each atom `R` into two atoms `S`. The second phase occurs in Rule 3, where each atom `S` is transformed into an atom `R`, obtaining two copies of each initial atom `R`. The transition between the two phases is made by Rule 2, when all the initial `R` atoms have been consumed.

Additionally, `Criojo` offers support for native guards, an extension that can be encoded in the Heta-calculus by using two rules and impure atoms.

$$\begin{array}{lll} \text{Guard} & g ::= \ldots & \\ & \mid x\ \texttt{op}\ y & \text{(boolean operation)} \end{array}$$

Native guards are treated as an extension based on native tests performed in the host language. For example, in the following rule we use the expression `x > y` as a guard in order to compute the maximum value v for atoms `V`(x):

```
1 !V(x) −−> (Abs(Max(x)) && Not(V(y)−>{y>x})) ?: Max(x)
```

The rule produces an atom `Max` with the maximum value, while keeping all the `V` atoms in the solution. The bang (`!`) symbol indicates that the atom `V(x)` is persistent in the solution for this rule. Thus, writing `!L(x) −−> R(x)` is the same as writing `L(x) −−> (R(x) & L(x))`. It is important to note that the expressions in the guard cannot contain free variables. They must be bound either to the variables in the guard or to the variables in an outer guard.

3.1.1.2 `Criojo` Adapters

An adapter allows the use of a component in a context different from the one it was initially intended, in order to allow collaboration with other components. In the case of `Criojo`, an adapter wraps an external component, abstracting it in terms well suited to the chemical machine. Concretely, it simulates a possible infinite set of rules that generate the

atoms corresponding to the resources provided by the wrapped component. Adapters in Criojo are defined as special types of relations called *Native Relations*[2] which are associated with native functions that transform a collection of terms into a molecule, that is to say, a collection of atoms. The result of applying a native relation to a set of terms is the execution of its function, followed by the introduction into the solution of the returned value, a molecule.

Let us now show with an example how to integrate the host language's functionalities into Criojo. In the following extract we use a native invocation to populate the solution with a given number of atoms.

```
1  val Init  = Rel[Int]
2  val L = Rel[Int,Int]
3  private val _genList = NativeFun[Int]{
4    case WrappedValue(max:Int)::_ =>
5     (0 to max).map(i => L(i,max-i)).toList
6    case _ => List[Atom]()
7  }
8  rules(
9    Init(x) --> _genList(x)
10 )
```

The NativeFun constructor used in line (2) associates the native relation _genList to a function with type List[Term] => List[Atom]. The argument type is a list as the arity of the native relation can be any natural number. The arity here is assumed to be equal to one: only the first term of the list is pertinent. The type parameter, here equal to Int, determines the Scala type of the values wrapped in the list. We use Scala's pattern matching in line (4) for extracting the actual type and value of the term, which in this case is an Int wrapped in a Criojo term. In line (5) the list of atoms is produced by iterating max number of times. The result is a list of atoms L(i,v), representing an indexed list where i is the index, and v the corresponding value. Thus, each time the rule in line (9) produces an atom _genList(n), the associated function is executed, returning n atoms L(i,n-i), which are introduced into the solution.

This simple example shows us how thanks to Criojo's adaptability

[2] In the Heta-calculus, we said *impure* relations. In Criojo, they are called *native* as they are implemented in the host language.

some computations can be optimized by exploiting the host language's functionalities from within agents. Yet the real importance of this feature resides on the possibility of interacting with external components such as input/output devices, database systems or web services. In this way Criojo's adaptability adheres to the *black box principle*, which is a derived requirement for the orchestration of web services we already mentioned in Section 2.2.

3.1.1.3 Modularity and Composition

Modularity and separation of concerns can be achieved in Criojo via agent collaboration. Another way of assuring the single responsibility principle is by factorizing behavior in independent modules which can be later combined within an agent. To illustrate modularity in Criojo we define the following agent that computes a Sierpinski triangle and prints the result in a SVG file by using the module called SVGPainter.

```
1  val sierpinskiAgent = new Agent with SVGPainter{
2    ...
3   rules(
4    Sierpinski(h,n) −−> srpsk(h,0,h,n),
5
6    srpk(x,y,h,n) −−> {n>0} ?:  (srpsk(x, y, h/2, n−1) & srpsk(x−h/2,
       y+h/2, h/2, n−1) & srpsk(x+h/2, y+h/2, h/2, n−1)),
7
8    srpsk(x,y,h,n) −−> {n===0} ?:  (paintTriangle(x−>y, (x−h)−>(y+h),
       (x+h)−>(y+h))),
9   )
10 }
```

The declaration in line (1) states that the agent sierpinskiAgent is as usual an instance of class Agent, but combined this time with the module SVGPainter, which is actually a Scala trait [3]. The module SVGPainter contains the rules for creating SVG images. It provides a relation paintTriangle for painting a triangle with the given coordinates. Line (4) of the program corresponds to the initialization phase: upon receiving a message Sierpinski(h,n), the agent initiates the com-

[3] Traits in Scala differ from *interfaces* in that traits allow partial implementation.

putation of an isosceles Sierpinski triangle of height `h` and of base `2.h`, and whose apex is located at `x=h,y=0`. The corresponding points of the triangle are `(0, h), (h, 0), (2h, h)`. The parameter `n` is the desired number of iterations. The iterative rule in line (6) transforms a triangle into three reduced copies, whose height is half of the height of the initial triangle. Each copy is positioned in such a way that it touches the other two at a corner. In line (8), the triangles are painted, when the desired depth is obtained i.e., when `n` is equal to zero. The coordinates of the corners of each triangle are send in the form of tuples, expressed with the notation `x -> y`.

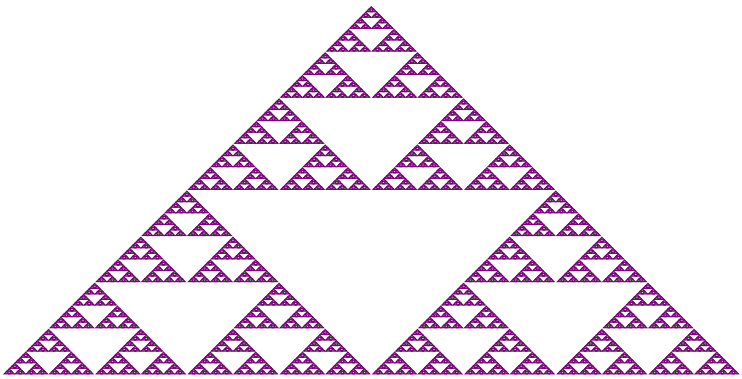

Figure 3.1: A Sierpinski triangle generated with `sierpinskiAgent`

The `Criojo` modules are an alternative to separate agents. Semantically, the solutions are equivalent, if we neglect the possibility of name conflicts and introspection: this is a form of location transparency. But syntactically, the solution with modules is more concise. Thus, `Criojo` offers re-usability thanks to modularity and agent composition. In this way, we can factorize common behavior into `Scala` modules that can be used in different contexts.

3.1.2 Distribution in `Criojo`

Besides being a language for writing rules, `Criojo` is above all a language for distributed computing. Agents in `Criojo` can communicate with each other by exchanging messages through channels. We make the distinction between two types of channels: *input channels* are provided by the agent we are defining, while *output channels* refer to the channels provided by

another agent. The output channel of one agent corresponds to the input channel of another agent.

To show how distribution works in Criojo, we revisit the authentication example from Section 3.1.1. In this version, the validation is distributed into two separate agents. The new authAgent deduces the user's access rights from her profile and compares them with the resource's access restrictions that it obtains from another agent called resAuth.

```
1  val authAgent = new Agent("authAgent", LocalGateway){
2    val getRes = OutputChannel("getResourceAccess", "resAuth")
3    val resAccess = InputChannel("resAccess")
4
5    val Guest, Admin, Registered, Resource, Done = Rel[UUID]
6    val User = Rel[UUID, String]
7    val Error = Rel[UUID, UUID]
8    val uid, rid = Var[UUID]
9    val access = Var[String]
10
11   rules(
12     Guest(uid) --> (User(uid, "Public") & Done(uid)),
13     Registered(uid) -->
14       (User(uid, "Public") & User(uid, "Members") & Done(uid)),
15     Admin(uid) -->
16       (User(uid, "Admin") & User(uid, "Public") &
17        User(uid, "Members") & Done(uid)),
18
19     Resource(rid) --> getRes(rid, resAccess),
20
21     (resAccess(rid, access) & User(uid, access) & Done(uid)) -->
       Ok(rid, uid),
22
23     (resAccess(rid, access) & Done(uid)) -->
24       {Abs(User(uid, access))} ?: Error(rid, uid)
25   )
26 }
```

The new agent defines two channels: `getRes` and `resAccess`. The output channel `getRes` in line (2) makes reference to the channel located at the agent `resAuth`. The input channel `resAccess` declared in line (3) is sent as a parameter of the outgoing message in line (17). Upon receiving a response through this channel, the rule in line (19) compares the resource's access restrictions with the user's access rights. When a match is found, the result is an atom `Ok` with the id of the user and the id of the resource. If no matching `User` atom is found in the solution, the rule in line (21) is executed, producing an error atom for that user with that resource. The atom `Done(uid)` is used as a token for guaranteeing that the rule is executed only after the three rules in (10, 11, 13) have been executed.

For this example we use a simple local bus called `LocalGateway` for handling the communication, where the agents can be located by name and the channels by their own name and the providing agent. The architecture is therefore flat, with a unique firewall containing orchestrators. The details for the implementation of more complex architectures and protocols is explained in Section 3.3.4.

3.1.3 More Examples

In order to show the expressivity of the language `Criojo`, we now present examples using four idiomatic programming paradigms: concurrent, functional, sequential and logic programming.

3.1.3.1 Concurrent programming: the π-calculus

As an example of concurrent programming, we implement the asynchronous π-calculus [SW01, chap. 5], where a process p is defined as follows.

p	$::= 0$	The empty process
	$\mid p \parallel p$	Parallel composition or concurrency
	$\mid \overline{x}\, y$	Message emission
	$\mid x(y).p$	Message reception
	$\mid\, !x(y).p$	Replication
	$\mid \nu x.p$	Name creation

Names are central in the π-calculus, identifying both communication channels and variables. In order to avoid confusion with `Criojo` channels,

in the following a π-calculus channel will be called a name; while a channel will always refer to a `Criojo` channel. Also, for simplicity we only consider replication in the case of message reception. For the implementation of the π-calculus cham we define two predicates: `Pi` for processes and `New` for creating new names. Thus the state of an agent implementing a π-calculus process is represented by the molecule $\texttt{Pi}(p)$ & $\texttt{New}(n)$, where p is a process and n is a counter for generating new identifiers. A name is represented by the pair `(l, n)`, where `l` is a unique channel provided by the agent, and `n` is an identifier generated by the agent. Additionally, we define `Criojo` expressions (through `Scala` types) for representing π-calculus expressions, which gives the following translations: `snd(l,n,x)` for $\overline{(l,n)}\,x$ (message emission), `rcv(l,n,x).dot(p)` for $(l,n)(x).p$, `nu(x,p)` for $\nu x.p$, and so on. The following rules provide the translation of the π-calculus into `Criojo`.

```
1  (Pi(nu(x,p)) & New(n)) −−> (_doSub(p,(l,n),x,pi) & New(n+1)),
2  (Pi(snd(k,c,x)) & Pi(rcv(k,c,y).dot(p))) −−> _doSub(p, x, y, pi),
3  (Pi(snd(k,c,x)) & Pi(!rcv(k,c,y).dot(p))) −−> (_doSub(p,x,y,pi) &
     Pi(!rcv(k,c,y).dot(p))),
4  Pi(p1 || p2) −−> (Pi(p1) & Pi(p2))
5  pi(p) −−> Pi(p)
```

It is also easy to provide a distributed version. The π-calculus particle $\overline{(k,n)}\,x$ is represented in `Criojo` as the message `k(n,x)`, when `k` is different from the local channel `l` used to represent names. Then we define two additional rules that allow the π-calculus cham to receive messages on its unique channel `l`, and to export messages to external channels. Note that the rule in line (6) only executes when channel `k` is different from the local channel `l`.

```
6  l(n,x) −−> Pi(snd(l,n,x)),
7  Pi(snd(k,n,x)) −−> {k =!= l} ?: k(n,x)
```

Note that a programming discipline is required to prevent a remote name from being locally used as a receiving name. Thanks to `Criojo`'s capacity to interface with native functions, we have defined the native function `_doSub(p, y, x, pi)` that performs a substitution `[x:=y]` in process `p` and sends the result trough a local channel `pi`. With this implementation, we can write the following π-calculus process composed of three parallel

components.

$$\begin{aligned} \texttt{p1} &= \overline{\texttt{ping}}\ \texttt{pong} \\ \texttt{p2} &= !\texttt{ping}(k).\nu x.\overline{k}\, x \\ \texttt{p3} &= \texttt{pong}(x).\overline{\texttt{result}}\, x \\ \texttt{p4} &= (\texttt{p1} \parallel \texttt{p2} \parallel \texttt{p3}) \end{aligned}$$

The π process expressed in Criojo syntax gives the following atom that is introduced into the π-cham.

```
1 Pi(snd(l, ping, pong))
2    || !rcv(l, ping, k).dot(nu(x, snd(l, k, x)))
3    || rcv(l, pong, x).dot(snd(l, result, x))
```

Initially, the cham has the state [New(0)]. Upon reception of the atom, the rule in line (4) splits the atom into three atoms corresponding to the three components of the process (p1, p2, p3):

```
    Pi(snd(l, ping, pong)))
&   Pi(!rcv(l, ping, k).dot(nu(x, snd(l, k, x))))
&   Pi(rcv(l, pong, x).dot(snd(l, result, x)))
```

By executing the rule in line (3), followed by the rule in line (5), the two first atoms are transformed into the molecule

```
Pi(nu(x,snd(l, pong, x))) & Pi(!rcv(l,ping,k).dot(nu(x,snd(l,k,x))))
```

After performing the substitution in line (1), the new atom at left becomes

```
Pi(snd(l, pong, (l,0)))
```

According to rule (2), the new atom reacts with the remaining initial atom producing the final state

```
[Pi(snd(l,result,(l,0))) & New(1)
& Pi(!rcv(l,ping,k).dot(nu(x,snd(l,k,x))))]
```

3.1.3.2 Functional Programming

As an example of functional programming, we present inductive types with recursive operations written in Criojo. For a concrete application

in the case of natural numbers, let us implement a program that computes the factorial of a number n. In this example, the natural numbers are represented by the terms `0` and `Succ(n)`. First, we implement a request-response protocol with the following two rules.

```
1 fact(ret, s, n) −−> (Resp(ret, s, n) & Comp(s, n)),
2 (Resp(ret, s, n) & Res(s, n, r)) −−> ret(s, n, r)
```

The first rule handles the set-up. It initializes the computation by generating an atom `Comp`, that is in charge of keeping the ongoing computation. It also produces a `Resp` atom that keeps the session identifier `s`, the return channel `ret` and the argument `n`. Once we have the result in an atom `Res`, the second rule sends the response through the return channel `ret`.

The recursive operation is executed by the following rules, following a top-down approach.

```
3 Comp(s , 0) −−> Res(s, 0, 1),
4 Comp(s, Succ(n)) −−> (Mult(s, Succ(n)) & Comp(s, n)),
5 (Res(s, n, r) & Mult(s, Succ(n))) −−> _doMult(s, Succ(n), r, res),
6 res(s, n, r) −−> Res(s, n, r)
```

The rule in line (3), handles the base case, or the end of the recursion, storing a partial result in `Res`. Line (4) shows the rule that recurses over `Succ(n)`, producing the atoms `Mult` that keep the multiplications to perform later on line (5). Finally, in line (5) the result is produced by performing the pending multiplications, in a bottom-up movement. Since we cannot directly compute `(n+1) * r`, we use a native relation called `_doMult`, equivalent to performing the multiplication in a recursive way. The result of the multiplication is retrieved by the rule in line (6). An example execution of the program with initial state O[`fact(ret, s, Succ(Succ(0)))`], for computing `2!` produces the fol-

lowing trace:

$$
\begin{aligned}
& O[\texttt{fact(ret, s, Succ(Succ(0)))}] \\
\Rightarrow\ & O[\texttt{Resp(ret, s, Succ(Succ(0)))} \,\&\, \texttt{Comp(s, Succ(Succ(0)))}] \\
\Rightarrow\ & O[\texttt{Resp(...)} \,\&\, \texttt{Mult(s, Succ(Succ(0)))} \,\&\, \texttt{Comp(s, Succ(0))}] \\
\Rightarrow\ & O[\texttt{Resp(...)} \,\&\, \texttt{Mult(s, Succ(Succ(0)))} \,\&\, \texttt{Mult(s, Succ(0))} \,\&\, \texttt{Comp(s, 0)}] \\
\Rightarrow\ & O[\texttt{Resp(...)} \,\&\, \texttt{Mult(s, Succ(Succ(0)))} \,\&\, \texttt{Res(s, Succ(0), 1)}] \\
\Rightarrow\ & O[\texttt{Resp(ret, s, ...)} \,\&\, \texttt{Res(s, Succ(Succ(0)), 2)}] \\
\Rightarrow\ & O[\texttt{ret(s, Succ(Succ(0)), 2)}]
\end{aligned}
$$

This example gives us the general form for recursive operations over the natural numbers:

- an initialization rule, following the request,
- rules for handling the base case and the recursive cases, following a top-down approach,
- a reduction phase, producing the result from the base case and the pending computations, following a bottom-up approach,
- the sending of the response.

3.1.3.3 Sequential Programming: a Variant of Dijkstra's Guarded Commands Language

Recall the cloning example given in Section 3.1.1.1.

```
1 (One() & R(x)) −−> (One() & S(x) & S(x)),
2 One() −−> Abs(R(x)) ?: Two(),
3 (Two() & S(x)) −−> (Two() & R(x))
```

Thus, sequencing in Criojo can be managed with the aid of tokens. We now show that this solution is general, by translating into Criojo a language for sequential programming, namely a variant of Dijkstra's Language of Guarded Commands. In this language, the cloning example would be expressed as follows.

```
1 do(R() −−> S(), S()) ; do(S() −−> R())
```

The command `do` represents a loop that terminates when the rule inside cannot longer be executed. In this way, commands can be sequenced. If we add a blocking alternative, we get the following variant of Dijkstra's language of guarded commands.

Script	$p ::= \texttt{skip} \mid p\,;\,p \mid \texttt{if}\,\{c\} \mid \texttt{do}\,\{c\}$
Guarded Command Set	$c ::= r \triangleright p \mid c \parallel c$
Guard Rule	$r ::= s \multimap g\,?\,s$

There are two differences with respect to Dijkstra's language: first, the only atomic action is the empty one *skip*, second, the guard of the command becomes a one-shot rule, with a side-effect, called a guard rule, thus compensating the lack of actions. In the following, a guard rule with no message ($\emptyset \multimap g\,?\,\emptyset$) is simply denoted by its guard (g). The empty action is also omitted, $r \triangleright \texttt{skip}$ becoming r.

Let us now translate this version of Dijkstra's language into `Criojo`. The translation of each command depends on two tokens, B for "Begin" and E for "End", which are used to manage the scheduling of commands.

$$\begin{aligned}
\mathcal{D}(\texttt{skip})_{B,E} &= B \to \texttt{True}\,?\,E \\
\mathcal{D}(p_1\,;\,p_2)_{B,E} &= \nu\, I.\mathcal{D}(p_1)_{B,I}, \mathcal{D}(p_2)_{I,E} \\
\mathcal{D}(\texttt{if}\,\{c\})_{B,E} &= \mathcal{D}(c)_{B,E} \\
\mathcal{D}(\texttt{do}\,\{c\})_{B,E} &= \mathcal{D}(c)_{B,B}, (B \to \mathcal{G}(c)\,?\,E)
\end{aligned}$$

The empty script converts the begin token into the end token. The sequence $p_1\,;\,p_2$ requires an intermediate fresh token (cf. $\nu\, I.-$), which corresponds to the end of p_1 and the beginning of p_2. The translation of the alternative and the loop depends on the translation of the associated set of guarded commands. Note the differences: for the loop, the translation uses the same token, allowing a repetition, and adds a rule to quit the loop, when its guard rules cannot be fired. A guarded command is translated into a rule and the translation of the continuation script. Their sequencing results from the use of an intermediate fresh token.

$$\begin{aligned}
\mathcal{D}((s \multimap g\,?\,s') \triangleright p)_{B,E} &= \nu\, I.(s, B \to g\,?\,s', I), \mathcal{D}(p)_{I,E} \\
\mathcal{D}(c_1 \parallel c_2)_{B,E} &= \mathcal{D}(c_1)_{B,E}, \mathcal{D}(c_2)_{B,E}
\end{aligned}$$

Finally, given a set c of guarded commands, the guard $\mathcal{G}(c)$ expresses that the guard rules cannot be fired, by using control guards.

$$\begin{aligned} \mathcal{G}((s \multimap g \,?\, s') \triangleright p) &= \neg(s \to g) \\ \mathcal{G}(c_1 \parallel c_2) &= \mathcal{G}(c_1) \wedge \mathcal{G}(c_2) \end{aligned}$$

Note that the premise s can only contain local messages. Allowing external messages could lead to race conditions, since external messages can arrive and leave at any time. Hence, the termination of a loop should only depend on internal messages.

The previous cloning example is translated into the following Criojo program, which resembles the solution we had initially.

```
1 (R() & B() --> (S() & S() & I1()),
2 (S() & I()) --> (R() & I2()),
3 B() --> {Abs(R())} ?: I(),
4 I() --> {Abs(S())} ?: E(),
5 I1() --> B(),
6 I2() --> I()
```

Here is the trace that reflects the states of the agent throughout the execution of the program [4]:

$$\begin{array}{rlrl} & O[\texttt{R()} \,\&\, \texttt{B()}] & \Rightarrow^{+} & O[\texttt{S()} \,\&\, \texttt{S()} \,\&\, \texttt{I}] \\ \Rightarrow & O[\texttt{S()} \,\&\, \texttt{R()} \,\&\, \texttt{I2()}] & \Rightarrow & O[\texttt{S()} \,\&\, \texttt{R()} \,\&\, \texttt{I()}] \\ \Rightarrow^{+} & O[\texttt{R()} \,\&\, \texttt{R()} \,\&\, \texttt{I()}] & \Rightarrow & O[\texttt{R()} \,\&\, \texttt{R()} \,\&\, \texttt{E()}] \end{array}$$

To conclude, although it is possible to emulate sequential programming in Criojo by using tokens, the ideal would be to have dedicated constructs in Criojo, in other words to embed Dijkstra's language of guarded commands in Criojo. Nevertheless, it is not possible at the moment. We further discuss a possible extension to Criojo introducing dedicated constructs for sequential programming in Conclusion, Section 5.1.

3.1.3.4 Logic programming

Given that the Heta-calculus provides the theoretical foundations of Criojo and was influenced by logic programming, Criojo shares many features

[4] For simplification we have omitted some intermediate steps.

with logic languages like Datalog. A rule whose guard is true can be considered as an inference rule. However, contrary to Datalog, in Criojo the premises are consumed, as in Linear Logic. Thus, in order to translate Datalog rules into Criojo, we have to add the premises to the conclusion.

For instance, given a binary relation R, assume that we want to compute its reflexive and transitive closure. Here is the corresponding program in Datalog. Note that the predicate U defines the universe for values x.

$$\begin{aligned} R^*(x,x) &\leftarrow U(x) \\ R^*(x,z) &\leftarrow R(x,y) \wedge R^*(y,z) \end{aligned}$$

Following the preservation principle, a first attempt to translate the second inference rule would give the following rule, where U, R and Rt correspond to U, R and R^* respectively. Note that to abbreviate, we could have used the bang operator in front of the left atom patterns.

```
1 U(x) --> U(x) & Rt(x, x),
2 (R(x, y) & Rt(y, z)) --> (R(x, y) & Rt(y, z) & Rt(x, z))
```

However, this results in a loop generating an infinite number of atoms Rt(x, x) and Rt(x, z). To avoid this indefinite generation, we can require that an atom is either absent in the solution, or present with a unique occurrence. Thanks to introspection, we can force this condition with a guard.

```
1 U(x) --> Abs(Rt(x,x)) ?: U(x) & Rt(x, x),
2 (R(x,y) & Rt(y,z)) --> Abs(Rt(x,z)) ?: (R(x,y) & Rt(y,z) & Rt(x,z))
```

But there is still a problem: assume we now want to compute the Cartesian product R^2 of a unary relation R, which is performed as follows in Datalog.

$$R^2(x,y) \leftarrow R(x) \wedge R(y)$$

A naive translation would give the following rule in Criojo.

```
1 (R(x) & R(y)) --> Abs(R2(x,y)) ?: (R(x) & R(y) & R2(x,y))
```

Nevertheless, this rule cannot generate $R^2(x,x)$, which requires two atoms $R(x)$ in the solution. To solve the problem, we can either increase the number of occurrences of each atom in the solution, or require a linearity condition for Datalog rules. Both options are akin. We opt for the second alternative: it forbids a rule where there are two atoms belonging to the

same relation in the premises. The program needs to be rewritten as follows.

```
1  R(x) −−> Abs(R1(x)) ?: (R1(x) & R(x)),
2  (R(x) & R1(y)) −−> Abs(R2(x,y)) ?: (R(x) & R1(y) & R2(x,y))
```

The result is a program that duplicates each atom `R` in the solution, and that produces atoms `R2(x,y)` with all the possible combinations of atoms `R`. For example, the trace of the program initiating with the state $O[\texttt{R(1)} \,\&\, \texttt{R(2)}]$ is as follows

$$\begin{array}{ll} & O[\texttt{R(1)} \,\&\, \texttt{R(2)}] \\ \Rightarrow^{+} & O[\texttt{R(1)} \,\&\, \texttt{R(2)} \,\&\, \texttt{R1(1)} \,\&\, \texttt{R1(2)}] \\ \Rightarrow^{+} & O[\texttt{R(1)} \,\&\, \texttt{R(2)} \,\&\, \texttt{R1(1)} \,\&\, \texttt{R1(2)} \,\& \\ & \texttt{R2(1,1)} \,\&\, \texttt{R2(2,2)} \,\&\, \texttt{R2(2,1)} \,\&\, \texttt{R2(1,2)}] \end{array}$$

To conclude, we have shown that it is possible to translate into `Criojo` programs written in a variety of languages:

- concurrent languages, like the π-calculus,
- functional languages,
- sequential languages, like Dijkstra's guarded command language, and
- logic languages, like `Datalog`.

In each case, the translation is quite straightforward.

3.2 Towards an Efficient Implementation

We now describe the path from theory to practice, in other words from the Heta-calculus to `Criojo`. We start with the overall distributed architecture and terminate with the implementation of the chemical abstract machine associated to orchestrators.

3.2.1 Hierarchy of Communicating Agents

The whole architecture, illustrated in Figure A.1, is defined with the following components.

Orchestrator: implementation of an orchestrator containing a chemical solution and a set of reduction rules

Gateway: firewall part of an orchestrator used to send and receive messages

Firewall: implementation of a pure firewall used to transmit messages

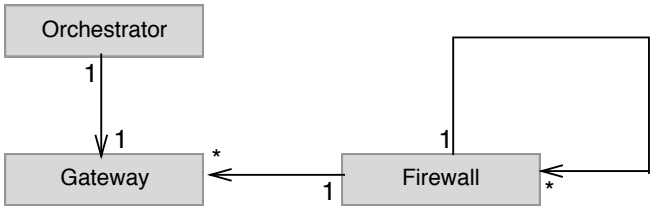

Figure 3.2: Class Diagram – Components of the Architecture.

A firewall has a parent firewall, except the topmost one, and child firewalls. A gateway has a parent firewall but no child: indeed, it is associated to a unique orchestrator, and conversely. Clearly, it is directly derived from the hierarchy of agents in the Heta-calculus, with the slight difference induced by the separation of functionalities between the gateway and the orchestrator. The different components can be distributed, for instance as Web services or around a bus. Communication between components is asynchronous, and essentially one-way. Responses are only required for acknowledgment, which can be useful to detect message losses.

Here are the abstract attributes of each component. They are deduced from the syntax and the operational semantics of the Heta-calculus.

- Orchestrator
  - A set of rules
  - A chemical solution defined as a multiset of atoms
  - A multiset of incoming messages to be added to the chemical solution before the next round

  - A multiset of outgoing messages to be sent to the gateway at the end of the last round

- Gateway, Firewall
  - A set of provided channels
  - A subset of private channels
  - The complement of public channels (computed attribute)

How to implement the sets of provided channels and of private channels? For the set of private channels, it is simple to explicitly declare in a firewall the channels that are private. For the set of provided channels, it is not efficient as an explicit declaration entails a lot of redundancy: indeed, a channel is provided by a firewall if it is provided by an orchestrator inside the firewall. In other words, all the firewalls between the root firewall and the orchestrator must declare the channel as provided. A simple solution is to implicitly declare provided channels. The namespace of channels is defined thanks to the tree structure of firewalls: the name of a channel in an orchestrator is the concatenation of the names of the enclosing firewalls, from the root firewall to the orchestrator (that is also a firewall), followed with a local name. For instance, a channel called `k` in orchestrator `O` enclosed in the root firewall `Web` is called `Web.O.k`. Likewise, firewalls can be named following the same manner, which gives its hierarchical name. The naming strategy is reminiscent of the one for `URI`. However, ambiguities are possible: we can imagine in a firewall two firewalls with the same name. Thus, two channels could have the same name, which is useful for load balancing for example. With this naming strategy, we can now define the implicit declaration: a firewall implicitly declares as a provided channel any channel whose name extends the hierarchical name of the firewall. Orchestrators explicitly declare the channels that they provide: thus, messages over inexisting channels are eventually filtered when they arrive at destination.

Following the reduction rules defined in the operational semantics, we describe the activities of each component. Each orchestrator has three concurrent activities.

Cham execution: activity possibly split into parallel activities. Parallelization, corresponding to the parallel execution of rules, is briefly studied further, at the end of Section 3.2.2.

Control of the multiset of incoming messages: two atomic operations, the addition of messages coming from the gateway and the transfer of the multiset into the chemical solution

Control of the multiset of outgoing messages: two atomic operations, the addition of messages coming from the chemical solution and the emission of the messages in the multiset towards the gateway

A gateway provides three atomic services.

Internal reception: reception of a message from the associated orchestrator, which entails a further emission

External reception: reception of a message from the parent firewall, which entails a further transmission to the orchestrator

Channel inspection: test to determine whether a channel is publicly provided, that is declared as provided and non private. The set of private channels can evolve, because of a scope extrusion during a message emission.

A firewall also provides three atomic services.

Internal reception: reception of a message from a child firewall, which entails a further emission

External reception: reception of a message from the parent firewall, which entails a further emission

Channel inspection: test to determine whether a channel is publicly provided, that is declared as provided and non private. The set of private channels can evolve, because of a scope extrusion during a message emission.

Here is the execution flow. The execution of an orchestrator is organized around *rounds*. An orchestrator produces and consumes atoms at each round. At the start of a round, it updates its chemical solution with the incoming messages, computes the multisets of the atoms consumed and produced by triggering one or more reduction rules, finally updates its chemical solution and the multiset of outgoing messages. At the end of the round, the messages produced by an orchestrator are sent to its gateway. When a gateway receives a message over some channel k from its orchestrator,

- if k is provided by the gateway, then the gateway sends the message back to the orchestrator (self emission for the orchestrator),

- otherwise, it transmits the message to the parent firewall.

When a firewall receives a message over some channel k from a child firewall (internal reception),

- if k is provided by the firewall, then the firewall transmits the message to one of its child publicly providing the channel,

- otherwise, it transmits the message to the parent firewall.

When a firewall receives a message over some public channel k from its parent firewall (external reception),

- it transmits the message to one of its child publicly providing the channel.

When a gateway receives a message over some public channel k from its parent firewall (external reception),

- it transmits the message to the associated orchestrator, by adding it to the multiset of incoming messages.

Note that when a firewall or a gateway sends a message to its parent firewall, it may update the set of private channels by removing extruded channels. Note also that pending messages are possible in a firewall: messages over a channel that is provided but is not public. No specification is given for these messages: they require specific actions.

We now describe the implementation of the consumption and production of atoms inside an orchestrator.

3.2.2 Chemical Evaluation of Local Reduction Rules

We first formalize the evaluation algorithm for local rules from the operational semantics, and its inference system. We then show how to efficiently implement the algorithm by incrementalization.

3.2.2.1 From the Operational Semantics to an Evaluation Algorithm

We describe as an algorithm the inference system defining the execution inside an orchestrator: see Table 2.2 for the inference system and Table 2.5 for the form of the rules. To simplify, we omit cleaning and conversion rules and only consider standard rules. Recall that these rules preserve the solution variable:

$$O[a_1 \,\&\, \ldots \,\&\, a_m \,\&\, S] \to g \,?\, O[b_1 \,\&\, \ldots \,\&\, b_n \,\&\, S],$$

where $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$ are atom patterns, and S the solution variable.

First, we decompose a rule $O[h] \to g \,?\, O[c]$ into a pre-rule, a rule without a conclusion, defined as a tree containing head patterns corresponding to the head h of the rule and the heads of the guard g, . Here is the formal definition of pre-rules, where we deliberately omit orchestrator O to simplify.

Rule	r	$::=$	$h \to g \,?\, c$	(head $\to$ guard ? conclusion)
Guard	g	$::=$	$\bigwedge_i(\neg(h_i \to g'_i))$	(conjunction of control guards)
Pre-rule	pr	$::=$	$h(pr^*)$	(head with sequence of pre-rules)

The decomposition of rules into pre-rules is defined with operator T (or "Transformer").

$$\begin{aligned} T(h \to g \,?\, c) &= T(h \to g) \\ T\big(h \to \textstyle\bigwedge_i(\neg(h_i \to g'_i))\big) &= h\big(T(h_i \to g'_i)\big)_i \end{aligned}$$

We can now determine the candidate valuations. Semantically, given a rule $h \to g \,?\, c$ and a chemical solution σ, we seek to determine the set of candidate valuations defined as follows:

$$\{\tau \mid (h[\tau] = \sigma) \wedge (\sigma \models_\tau g)\}.$$

This set is computed by the function C (or "Candidates"): it takes as arguments the pre-rule associated to the rule and the chemical solution to recursively produce the set of candidate valuations, with the help of two adjoint functions:

- function M (or "Matching") for pattern matching with respect to chemical solutions,

- function J (or "Join") for joining a valuation and a set of valuations.

$$\begin{array}{rcll} C(h(pr_i)_i, \sigma) & = & \text{let} & (V_i = C(pr_i, \sigma))_i \\ & & & V = M(h, \sigma) \\ & & \text{in} & \{\tau \in V \mid \bigwedge_i (J_{\mathrm{BV}_i}(\tau, V_i) = \emptyset)\} \end{array}$$

The join operator is defined as follows, given a set X of variables, a valuation τ and a set V of valuations, all these valuations having a domain containing X:

$$J_X(\tau, V) = \{\tau' \in V \mid \forall x \in X.\tau'(x) = \tau(x)\},$$

The set BV_i is the set of the variables bound by head h in the head h_i of pre-rule pr_i:

$$\mathrm{BV}_i = \mathrm{Var}(h) \cap \mathrm{Var}(h_i).$$

The matching function is defined as follows:

$$M(h, \sigma) = \{\tau | h[\tau] = \sigma\}.$$

To compute function C, we resort to an efficient strategy, *incrementalization* [Liu00]. Indeed, recall that the execution of the local rules inside an orchestrator are organized with rounds. At each round, there are a consumption and a production of atoms. Rather than computing at each round from scratch the function C, we can compute its variation. Precisely, we follow the general and systematic approach to incrementalization formalized by Liu [Liu00] that can be stated as follows. To compute the result of a program f over some input $x \oplus \delta$, where x is the initial data and δ the variation added with the binary operator $\oplus$ (assumed left associative), the approach aims at deriving a new program computing an incremental version of f, precisely by using

- the return value of $f(x)$,
- the intermediate values computed during the evaluation of $f(x)$,
- or some auxiliary values that can computed during the evaluation of $f(x)$ but that are not required.

It is a form of memoization. Here is description of the intended benefits. First, assume we need to compute the following sequence.

$$f(x) \to f(x \oplus \delta_1) \to f(x \oplus \delta_1 \oplus \delta_2) \to \ldots \to f(x \oplus \delta_1 \oplus \ldots \oplus \delta_n).$$

Then an incremantalization transforms the computation sequence into the following sequence.

$$(f(x), C_0) \to (f(x \oplus \delta_1), C_1) \to (f(x \oplus \delta_1 \oplus \delta_2), C_2) \\ \to \ldots \to (f(x \oplus \delta_1 \oplus \ldots \oplus \delta_n), C_n),$$

where each context C_i represents intermediate or auxiliary values computed at each step. The incremental function, denoted by φ, computes the transition at each step:

$$\varphi(f(x \oplus \delta_1 \oplus \ldots \oplus \delta_{n-1}), C_{n-1}) = (f(x \oplus \delta_1 \oplus \ldots \oplus \delta_n), C_n).$$

Incrementalization is efficient in time and space complexity when the following conditions are met.

- The initial state $(f(x), C_0)$ can be computed with the same complexity as $f(x)$ (ideally, a linear complexity).
- Each transition can be computed with a complexity linear with respect to the variation δ_i.

If the variations δ_i are small with respect to the initial value x, then the sequence can be computed with a better complexity: $n + p$ versus $n.p$, where n is the complexity for the initial computation $f(x)$ and p is the number of steps in the sequence.

In the next section, we present an incremental version for the evaluation of rules.

3.2.2.2 Rule Evaluation with Automata

We restrict ourselves to the matching function: in other words, we deal with (standard) rules without guards (precisely with guard $\top$, always true). To compute the set of candidate valuations during a round, we need to find a match between a sequence of atoms in the solution and the head pattern. For incrementalization, we simply memoize partial matches during a round. The adequate structure is a state machine (an automaton).

Thus, the state of the matching for the rule is represented with a state machine, a common technique for pattern matching. At the beginning, when the solution is empty, all the machines are at their initial state. Upon arrival of atoms, rules are triggered and their state machines

advance towards their final state. When a state machine arrives at its final state, the corresponding rule becomes ready to be triggered. A similar solution has already been explored in the `jocaml` [FM98] system, one of the implementations of the join-calculus, where join-patterns are compiled into finite state machines. The difference with respect to our approach is that the `jocaml` implementation only considers linear patterns, and hence does not take into account correlated variables: the extension requires some extra care as we will see.

States in the state machine are represented by *matching vectors*. A matching vector is a sequence of 0 and 1 that respectively reflects the absence or presence of atoms matching a pattern. Transitions are labeled after the atom patterns in the rule head. For example, the rule

$$A(x) \,\&\, B(x,y) \to C(x,y)$$

is represented by the state machine $M = (\Sigma, S, S_0, S_\mathrm{F}, \delta)$, where the alphabet Σ, the set of states S with initial state S_0 and final state S_F are defined as follows

$$\begin{array}{ll} \Sigma = \{A, B\} & S = \{(00), (01), (10), (11)\} \\ S_0 = \{(00)\} & S_\mathrm{F} = \{(11)\} \end{array}$$

and the transition function δ is defined by

$$\begin{array}{ll} \delta((00), A) = (10), & \delta((00), B) = (01), \\ \delta((10), B) = (11), & \delta((01), A) = (11). \end{array}$$

During the execution, complementary information is added to each state: partial valuations, computed from the matching between patterns and atoms, and partial sequences of the identifiers associated to the matching atoms. Therefore, the addition or removal of an atom leads, respectively, to the addition or removal of its corresponding valuations. Shown below is the addition algorithm, explained with an example, followed by the removal algorithm. Notice that atom identifiers are indispensable for removing atoms, and thus valuations, from the state machine.

The algorithm executed when adding an atom $X^{id}(v)$, where id is the identifier of the atom, is as follows.

1. **Update**: For each rule $(R_1(p_1) \,\&\, \ldots \,\&\, R_n(p_n) \to \ldots)$, for each transition $M_1 \xrightarrow{R_i(p_i)} M_2$ ($M_1(i) = 0$ and $M_2(i) = 1$) in the corresponding state machine, if atom $X^{id}(v)$ matches pattern $R_i(p_i)$:

(a) Calculate the valuation $\theta : \{x_1 = v_1, ..., x_m = v_m\}$ from the matching between $X^{id}(v)$ and $R_i(p_i)$.

(b) If M_1 is the initial state (0^n), add the tuple $(i \mapsto id, \theta)$ to M_2, where $i \mapsto id$ is the map that associates i with the atom's id.

(c) Otherwise, for each tuple (I_1, θ_1) in M_1, where I_1 is a map of atom ids: if $R_i(p_i)[\theta_1]$ matches $X^{id}(v)$, add $(I_1 + (i \mapsto id), \theta_1 + \theta)$ to M_2. Because of the matching condition (requiring a match with the partial valuation already computed), conflicts between θ_1 and θ are avoided.

2. **Firing**: For each valuation in the final state (1^n), evaluate its guard. If the guard is satisfied, then the valuation becomes a candidate valuation for firing.

   Note: to ensure fairness, rules and valuations are chosen in a FIFO order.

Consider for instance the rule:

$$A(x) \,\&\, B(x, y) \to C(x, y)$$

To this rule corresponds the state machine of Fig. A.2, after receiving the atoms $B^1(a, b)$, $A^2(c)$ and $A^3(a)$. Initially, the solution and the state machine are empty. First, the atom $B^1(a, b)$ arrives and the valuation $\{x = a, y = b\}$ is added to the state (01). Then, the atom $A^2(c)$ arrives and the valuation $\{x = c\}$ is added to the state (10); but, since there is no match between $A(x)[x = a; y = b]$, that is $A(a)$ and $A^2(c)$, no valuation is added to the state (11). Finally, the atom $A^3(a)$ arrives and the valuation $\{x = a\}$ is added to the state (10). Because this time there is a match, the valuation $\{x = a, y = b\}$ is added to the state (11). Furthermore, each valuation is coupled with a sequence of the ids of the atoms that produce them.

When a rule is triggered, all the atoms consumed in the reaction are removed from the solution, along with their respective valuations. Thanks to atom identifiers, the algorithm for removing an atom $X^{id}(v)$ is as simple as removing from the state machine all the tuples (M, θ), such that $id \in M$.

It is possible to parallelize the computations performed by several rules, as explained in Section 2.4.2. First, we define the relation "parallelizable with" over the set of pertinent rule instances: see its definition

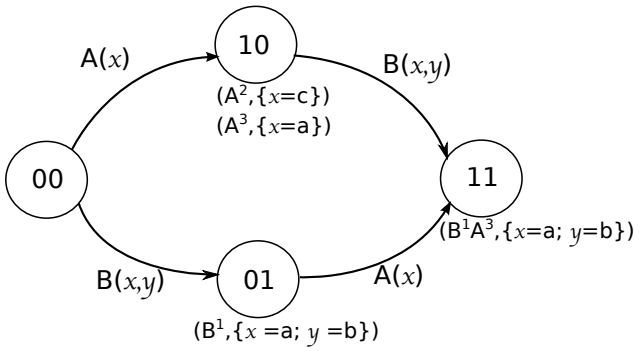

Figure 3.3: The state machine corresponding to the rule $A(x) \,\&\, B(x,y) \to C(x,y)$ after receiving the atoms $B^1(a,b), A^2(c)$, and $A^3(a)$.

in Section 2.4.2. Recall that a rule instance is a rule partially instantiated with a valuation. For instance, if a rule uses a session variable for each atom, the instantiation will assign a real session identifier to the session variable. For each rule instance, maintain a state machine to incrementally compute at each round the set of candidate valuations. Then pick up a valuation in some non-empty sets of candidate valuations, while satisfying the following safety property: the valuations selected belong to rule instances that are pairwise parallelizable. Finally apply to the chemical solution the updates induced by the valuations selected. Note that the current implementation does not resort to parallelization for local rules.

Finally, one question remains open: how to deal with guards? If incrementalization is a success for the matching function, it remains that the evaluation of rules also involves the evaluation of guards: its incrementalization is still an open question. Indeed, consider a pre-rule: for each head in the pre-rule, a state machine is defined, and the corresponding matching function M can be computed incrementally. When a set of the candidate valuations associated to a state machine evolves, passing from an empty set to a non empty set, or conversely, the evaluation of the guards using this head needs to be updated. The efficient incrementalization of this evaluation is difficult because of the binding of variables between different levels in the rule, since any head can bind variables in its guards. The current implementation does not resort to incrementalization for the evaluation of guards. The impact is not so severe because the structure of pre-rules is rather flat, with usually zero or one levels of

guards.

3.3 Implementation Details

We now come to the concrete implementation. `Criojo` is essentially a language for writing reduction rules whose semantics is based on the Heta-calculus. In the general schema, rules are generated from a set of definitions and then executed. Rules can be generated from the compilation of scripts written with a concrete syntax, or reified from a definition using the syntax of a host language. The generated rules can be then executed either by compiling them into concrete objects that can be executed on a concrete machine, or by interpreting them, in which case they are executed over an abstract machine. Thus, several combinations are possible between the approaches chosen for generating and executing rules, as shown by Figure 3.4.

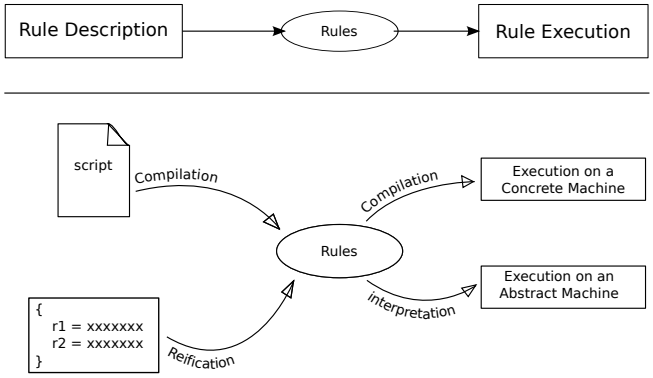

Figure 3.4: General schema for rule definition and execution.

In the current implementation in `Scala` we have opted for defining the rules by extending the host language's syntax with an internal DSL, and then to interpret them and execute them over the Java Virtual Machine. The choice of `Scala` as a host language was made mainly based on the facility that `Scala` offers for building internal DSLs, which facilitates the production of a prototype. Indeed `Scala` allows to extend its syntax by overloading operators and thanks to type classes and implicit conversions, it is possible to *lift* `Criojo` expressions to support native operations. For example, if a `Criojo` variable is declared as `val x = Var[Int]`, we can use the expression `x + 1` in a rule even if the `+` method is not defined for

the type Var. Thus, defining Criojo rules using Scala's syntax allows a Criojo cham to be seamlessly integrated within a Scala program, at the same time that Scala functionalities can be called from within the cham.

The concrete architecture is a simplification, faithful, of the general model described in Sections 3.2.1 and 3.2.2. Each activity corresponds to a Scala *actor* [HO06]. An actor is a concurrent computational unit which communicate through asynchronous message passing. This separation of functionalities allows optimizations to be added later. Note also that currently, there is no parallelism allowed by Criojo inside an orchestrator for rule evaluation. The hierarchy of agents is flat: there is no intermediate firewalls, between orchestators and the root firewall. This is the main simplification.

The remaining of the section details important aspects of the implementation, namely rule reification, Criojo's adaptors, the type system and the communication layer.

3.3.1 Rule Reification

As already said, Criojo is a language for writing rules. For the prototype implementation we have opted for reifying the rules from a definition written in an internal DSL, thus allowing the integration of a cham within a Scala program. In the examples of Section 3.1, rules are defined by expressions like the following one:

```
1 rules(
2    (A(x1, x2) & B(y1, y2) ) −−> C(x1, y2)
3 )
```

The expression inside rules() is syntactic sugar for calling the constructor of the RuleDefintion class which is the Scala representation of a rule. The --> operator combines the two molecules on the left and right hand, plus an eventual guard, to produce an instance of RuleDefinition. Thus the expression above corresponds to the following:

```
1 new RuleDefinition(head = List(A(x1,x2),B(y1,y2)),
2                    body = List(C(x1,x2)),
3                    guard = EmptyGuard)
```

Then, the rules() method takes a sequence of rule definitions and reifies them as instances of the type Rule belonging to the API. The

implementation of the `Rule` type is the choice of the implementor: it computes the evaluation of rules, as specified in Section 3.2.2, by using a state machine.

3.3.2 Implementing `Criojo`'s Adapters

The implementation of adapters in `Criojo` corresponds to the impure form of the Heta-calculus: it is based on devices that finitely simulate a possible infinite number of reduction rules. To understand how this kind of device is implemented in `Criojo`, let us quickly revisit the execution of a rule. The state of the rule is a finite state machine, where each state represents a partial match of the pattern in the head of the rule. For example, the rule $A() \,\&\, B() \to C()$ is represented by the states $(00), (01), (10), (11)$. Associated to each state, there is a set of valuations. A valuation is a mapping that gives a value to each variable in the head of the rule. When the final state (11) is reached and the guard of the rule is satisfied, the rule is ready to be executed: one of the valuations associated to the final state is passed to the atoms in the conclusion of the rule to produce new atoms. In fact, an atom expression in the conclusion is implemented as a partial function with type:

```
(List[Term]) => Valuation => Atom.
```

This function produces a new atom by applying the valuation selected to the list of (open) terms that form the arguments of the relation associated to the atom.

In order to allow the implementation of adapters, we compose this function with a native function, with type

```
Atom => List[Atom]
```

In the case of regular atoms, the function could be considered as the natural embedding. In the case of a *native atom*, a native function is called: it allows to replace the native atom in the solution by the atoms produced by the native function. Figure 3.5 illustrates how native atoms operate, compared to regular atoms.

3.3.3 Type System

The Heta-calculus has no type system. Nevertheless, types in a language like `Criojo` are necessary to guarantee the safety of programs by preventing run-time errors due to illegal operations. For this reason the

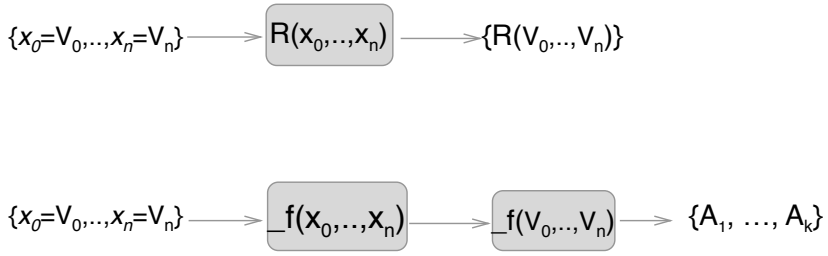

Figure 3.5: Regular Atoms and Native Atoms.

implementation extends the Heta-calculus by defining a type system that can be adapted to the type system of the host language.

Starting from the definition of relations, every term in `Criojo`'s is typed. Since a relation is a function that applied to a set of terms produces an atom, atoms too are typed and every variable that appears in a rule must be typed accordingly. Figure 3.6 shows the class hierarchy representing `Criojo`'s grammar for terms.

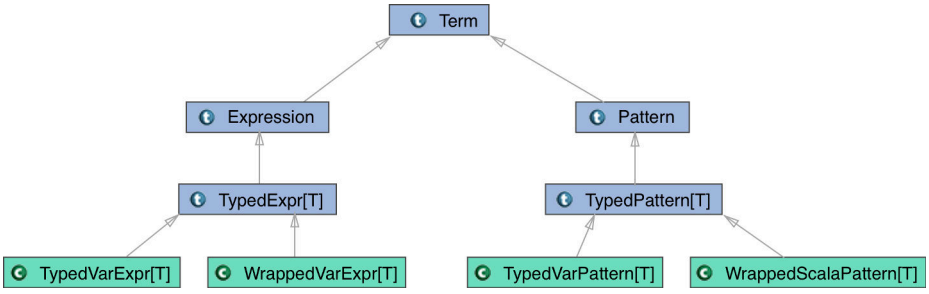

Figure 3.6: `Criojo`'s Term Grammar

In this model patterns and expressions are independent types, thereby guaranteeing the well-formed definition of rules: the terms in the head of the rule can only be patterns, while the terms in the conclusion can only be expressions. A pattern is a sequence of terms that provides a template for testing the presence of values matching the pattern in the solution. An expression, as in the mathematical sense, is a well-formed combination of symbols that can be evaluated and reduced. Nevertheless, the separation between patterns and expressions is not always clear: variables and constant values can be both patterns and expressions. For instance, consider

the list case: $x :: list$ is an expression representing the concatenation of an element x to a list, while the pattern $x :: \_$ is a template that matches with any expression of type list having x at the head. But not every expression can be used as a pattern. Such is the case of expressions like $x + y$, which would be problematic to have as a pattern: matching would be ambiguous, due to the presence of an operation. Thus, we can think of patterns as a subset of expressions, those based on constructors.

From the programmer point of view, the distinction between patterns and expressions is transparent, because we provide a mechanism to obtain patterns from expressions whenever it is possible. The same mechanism must be provided for new types added to the type system. In order to illustrate let us examine the implementation of lists, which is based on `Scala`'s `List` type. To begin, we present the following `Criojo` program that searches an element in a list:

```
1 val x,y = Var[Int]
2 val rest = Var[List[Int]]
3 val Search = Rel[Int, List[Int]]
4 val Result = Rel[Boolean]
5 rules(
6   Search(x, y::rest) −−> {x === y} ?: Result(True)
7   Search(x, y::rest) −−> {x =!= y} ?: Search(x, rest)
8 )
```

It is possible to declare a variable `Var[List[T]]` at any time. Nevertheless in order to use the `::` constructor as a pattern or expression we have to *lift* a list expression `Expression[List[T]]` so that it supports the `::` operator. In `Scala` this means creating a type that provides the `::` method, and then telling the compiler how to convert an `Expression[T]` into this type. The following listing shows how to declare a type `CanConcat` that implements the concatenation operator, which returns an expression of `List[Int]`:

```
1 class CanConcat[+T](l:Expression[List[T]]){
2   def ::[ U>: T](x:Expression[U]):Expression[List[U]]=
3     new Expression[List[U]]{
4       ...
5     }
6 }
```

Since expressions are transformed into patterns when needed, a similar class has to be declared for transforming a list pattern into a type implementing a `::` method:

```
1 class ConcatPattern[+T](l:Pattern[List[T]]){
2   def ::[ U>: T](Pattern[U]):Pattern[List[U]]=
3     new Pattern[List[U]]{
4       ...
5     }
6 }
```

Finally, for guaranteing that the program is well-typed and compiles, an implicit method is required to transform a list expression into `CanConcat` [5]. Such a method takes as argument an expression of type `List[T]` and returns an object of type `CanConcat[T]`, by calling the constructor `new CanConcat()`.

3.3.4 Communication Layer

`Criojo`'s communication model is based on asynchronous message exchange between agents. The flexibility of this model allows the language to adapt to different communication strategies.

As already said, in `Criojo`, the hierarchy of agents is flat: this is a simplification with respect to the model presented in Section 3.2.1: there are orchestrators enclosed in a root firewall. Each orchestrator, simply called agent in the following, is associated to an *Atom Gateway*, an interface for handling incoming and outgoing messages. The function of the gateway is to transform atoms into the format used by the implemented protocol, and to translate incoming messages into `Criojo` messages. Its implementation depends on the communication protocol used by the agent. For instance, if we implement an for working with `HTTP`, the atom gateway will transform an `HTTP GET` into a `Criojo` message including a channel for receiving the answer. The response of the agent, which is a `Criojo` message, is then translated into a `XML` or `Json` document and included into the `HTTP` response. The whole process is summarized in Fig 3.7.

[5] Implicit methods are executed by the compiler whenever a type T1 is required instead of a type T2.

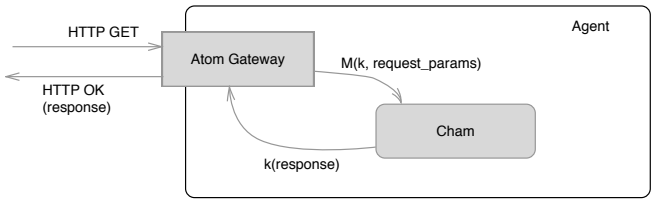

Figure 3.7: Communication Model – Example with `HTTP`

Now, in order to explain the implementation of concrete atom gateways, we use as example the implementation of the communication layer in our prototype. The communication layer is based on a Message Oriented Middleware (`MOM`) on which we implement a point to point message model. Besides an implementation based on a `MOM`, we find alternatives like an implementation with web services. The `MOM` infrastructure handles the exchange of messages between agents in an asynchronous and reliable way [Cur04]: messages are sent via the `MOM`, which handles the delivery of the messages with a store and forward mechanism. As messages are persisted, we can guarantee their eventual delivery if their receivers are not available, just like the postal service. Messages on the `MOM` are stored in queues, the way in which messages are retrieved form the queue depends on one of two message models: publish/subscribe or point-to-point. A publish/subscribe model is similar to a news channel, where clients subscribe to a topic in order to receive messages from a specific subject. In the point to point message model, on the contrary, a provider sends a message intended for a single consumer, and once the message has been delivered, it is removed from the queue. One more advantage of a `MOM` system is that senders and receivers are decoupled, thus allowing the communication between agents using disparate technologies, for instance a `Criojo` agent implemented in `Scala` and another one implemented in `JavaScript`.

On top of the `MOM` infrastructure we build the architecture depicted in Figure 3.8, consisting of a message bus composed of several nodes. Agents are connected to the nodes via *bus connectors* that handle the sending and reception of messages. Bus connectors are `MOM` clients capable of creating message queues as well as new nodes, allowing in this way the dynamic evolution of the network.

In order to connect the `Criojo` agents to the bus, we create a special kind of atom gateway called `BusAtomGateway` that has to be combined

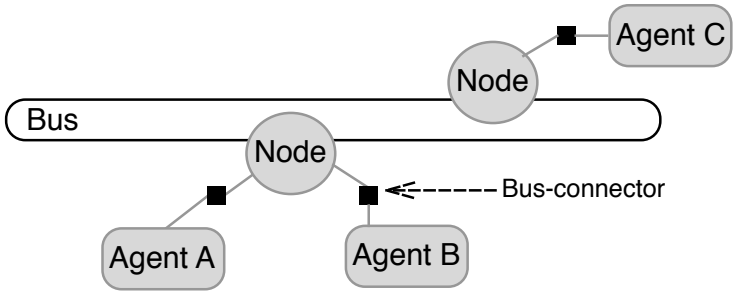

Figure 3.8: Implementation of communication layer.

with a bus connector. The bus connector relies on a `MessageHandler` for handling incoming messages. For this reason, `BusAtomGateway` is also of type `MessageHandler`. When the bus connector consumes a message, the message is transformed into an atom by the `BusAtomGateway`, which then immerses the atom into the solution. When an atom is exported from the cham, the `BusAtomGateway` makes the corresponding transformation and sends it to the bus via the bus connector.

In practice, a major challenge in the implementation of the communication layer is the problem of the agent's visibility when it stands behind a `NAT` (Network Address Translation) or firewall, since its addresses cannot be exposed to the Internet. One possible solution is the implementation of a `STUN` (Simple Traversal of UDP through `NAT`s) service, in the same way services like `Skype` and others `P2P` do to allow communication between clients whose `IP` address cannot be directly accessed. The `STUN` protocol allows an agent to determine the public `IP` and port it has allocated in the `NAT`, corresponding to its private `IP` address and port. With the `STUN` server standing on the other side of the `NAT`, agents can send binding requests to it for obtaining their `IP` and port as seen from the `STUN` server perspective, which are in fact the address and port attributed by the `NAT`. Nevertheless, this solution does not work in every context due to the heterogeneity of `NAT` schemes and the lack of standardization. In this case, another solution may involve the use of relays to get around the `NAT`.

Location transparency can be limited by the choice of the transport layer, for example in the case of the `HTTP` example, where the communication is not symmetric. Indeed, a client knows the server but the server cannot delegate the response to other server in a simple way.

To conclude the chapter, the language `Criojo` is still a prototype. See Table 5.2 in Conclusion for the detail of the functionalities implemented. However, features like control guards and the integration capabilities of `Criojo` are already an asset for the orchestration of services. First, control guards bring the possibility of introspecting the solution, for instance with absence guards `Abs`, which give the language more expressive power. Then, the capacity of interfacing with external resources, thanks to adapters, not only eases the implementation of the language by taking advantage of the existing features in the host language, but also provides an advancement towards interoperability. In conclusion, `Criojo` turns out to be already useful, as we show now with an application to interoperability problems in the context of service-oriented computing.

Chapter 4

A Pivot Solution to Interoperability Problems

Interoperability, in the context of service oriented computing, is not always straightforward. Assume, for instance, that you want to automatize the organization of your photos, which are managed by two different photo management systems, like `Picasa` and `Flickr`. You may quickly face interoperability problems, namely *adaptation*, *integration* and *coordination* problems. Indeed `Picasa` and `Flickr` interfaces differ not only from a functional point of view, as both interfaces use distinct resource models for organizing photos; but also from a communicational one, since `Flickr` provides both `Restful` and `WS*` services, while `Picasa` only provides `Restful` services. Therefore, an adaptation is needed when a client application that orchestrates `Picasa` services must evolve to orchestrate services from other providers such as `Flickr`, or conversely; or even when it must evolve from a `Restful` interface to a `WS*` interface, in the case of `Flickr`. An integration is needed when the client application must orchestrate both `Picasa` and `Flickr` services. A coordination is needed when two scripts, possibly written in distinct languages, must cooperate to orchestrate services provided by one system.

Typically, developers solve interoperability problems by implementing design patterns, or one of their variations. The *adaptation* problem can be solved with the Adapter pattern: an adapter built between the client and the new service provider allows to switch from one service provider to another without modifying the client. The *integration* problem can be solved with the Facade pattern: an intermediate component built between the client and the two service providers offers a common representation for the two resource models. Finally, the *coordination* problem can be solved with the Mediator pattern: a mediator component allows the coordination of two or more scripts by combining their results.

However, the three solutions rely on an architecture with a common framework between orchestration languages and interfaces. Our proposal

is a pivot architecture where scripts written in different languages are translated into a pivot language and then executed over different interfaces and data models. Since `Criojo` is a (presumably) universal language for interfacing resources and for orchestrating services, we use it as the language at the center of the pivot architecture.

In this chapter we first explain the pivot architecture, and state the features required in a pivot language. Then we further describe each one of the three interoperability problems and the proposed solution based on the pivot architecture with `Criojo` as the pivot language. The chapter is an extended version of our paper [LGL10].

4.1 The Pivot Architecture

The pivot architecture we propose is a middleware built around a universal orchestration language, called a *pivot language*. In this architecture, scripts written in existing orchestration languages, like `SQL` or `Java` frameworks, are compiled into the pivot language and then executed over different interfaces, like `Picasa`'s or `Flickr`'s.

To be effective, the pivot architecture relies on three assumptions for the pivot language: (i) that any orchestration language can be compiled into such language, (ii) that the pivot language can interact with different resource interfaces, and (iii) that the design patterns used to solve interoperability issues can be encoded in this language. We turn these assumptions into three requirements for the pivot language.

Universality for Compiling. In order to compile scripts written in different orchestration languages into the pivot language, we need a multi-paradigm language. Concretely, the pivot language must support compilation from imperative languages like `Java`, functional languages like `XQuery`, concurrent languages like `BPEL`, and logic languages like `YQL` or `SQL`. Note however that this is an approximate classification since each language also presents features from other paradigms.

Universality for Interfacing. Service interfaces differ not only from a functional point of view, but also from a communicational one, as illustrated by the case of `Flickr` and `Picasa`. A universal language for representing resources is therefore required, as well as a middleware layer capable of interfacing with different sources.

Expressivity. The pivot language must be expressive enough to allow the different software design patterns to be encoded. We consider

that this last requirement derives from the first one because a certain level of expressivity is required for allowing compilation from the different programming paradigms.

A possible choice for a pivot orchestration language is to use an existing one like `Scala` which clearly provides enough expressive power for interfacing with different resources and for coding the design patterns. Moreover, its asynchronous communication model based on actors reflects the message passing model that makes part of the requirements for an orchestration language. Nevertheless, we consider that this approach would have two drawbacks. First, proving the practicability of the solution in concrete cases would probably require an excessive implementation effort: on the contrary, chemical rules are very abstract and concise. Second, the experimentation would not emphasize the concepts that are essential for designing a pivot language. Actually with standard languages, there is a gap between the communication model, based on the exchange of messages, and the local computational model, generally concurrent, as described in Section 1.3.2. With a chemical programming model, as defined here, the gap disappears. Thus, we propose to use `Criojo` as a pivot orchestration language. Being an implementation of the Heta-calculus, `Criojo` follows a minimalist and more foundational approach, and concretely realizes the specification presented in Section 1.3.

4.2 Implementing the Pivot Architecture with `Criojo`

This section shows how `Criojo` is used as a pivot language in the pivot architecture, and how it can be used for solving interoperability problems. We present three different scenarios exposing the adaptation, integration and coordination problems, respectively, and propose a solution based on the pivot architecture, with `Criojo` as the pivot language. Our proposal is based on the implementation of design patterns [GHJV94]. Concretely, we implement the *Adapter pattern* to solve the adaptation problem, the *Facade pattern* to solve the integration problem, and the *Mediator pattern* to solve the coordination problem.

4.2.1 Adaptation: the Adapter Pattern

It is possible for a client application to change the provider of a service, because either the user likes novelty, or the current service is no longer available, etc. In any case, an adaptation is required and preferably in a way not requiring the complete refactoring of the client. One solution to this problem is the implementation of the adapter pattern, transforming the interface of a service into another interface, the one expected by the client.

To show how the adapter pattern can be implemented with `Criojo` and the pivot architecture, we take the case of a client application that is connected to `Flickr` and that uses Yahoo's `YQL` query language to query the number of photos taken in a given range of time, and we are going to adapt the client to work with `Picasa`.

First, let us briefly describe `YQL`, which is a `SQL`-like language proposed by Yahoo, allowing applications to query `Restful` services as if they were tables in a relational database. Services in `YQL` are invoked via queries written in a sub-set of `SQL`, where the queried table is a representation of the service. The table in question is called an *Open Data Table*, an `XML` document that maps the service's input and output parameters to columns. Client applications send their queries to the `YQL` service, which translates the query into an `HTTP` request and forwards it to the target service. The response is formated as a set of rows, as an `XML` or `Json` document and sent back to the client. The whole process is summarized in Figure 4.1:

1. The client application sends a query to Yahoo's `YQL` service.

2. The `YQL` service translates the `YQL` query into a `HTTP GET` request and forwards it to the target service.

3. The `YQL` service treats the response and forwards it back to the client.

In our example, we have mapped the `Flickr` method called `flickr.photo.getCounts` to an open data table with the same name. The `Flickr` method takes as parameter a list of date pairs, representing ranges, and returns the number of photos taken in each range. The open data table represents the method as a database table with three columns: {`count, from_date, to_date`}. Thus it can be called with the following query.

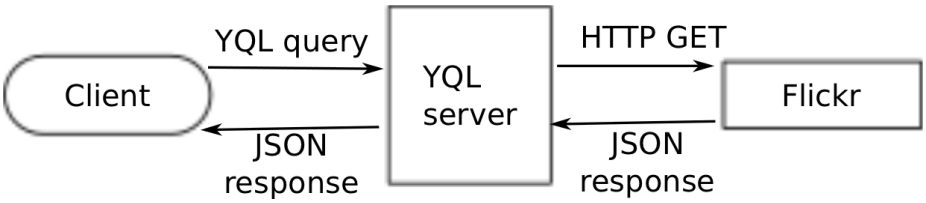

Figure 4.1: YQL Query Execution

```
SELECT count FROM photos.getCounts
WHERE from_date="1262307600"
AND to_date="1342132817"
```

Now, in order to adapt the client application so that it can switch from Flickr to Picasa without too many modifications, we propose to replace the YQL server with a Criojo component that implements the adapter pattern. Figure 4.2 shows the configuration of the new architecture including the Criojo component and the Picasa service: the YQL service is replaced by a Criojo component that can communicate with both Flickr or Picasa. The Criojo component is detailed in Figure 4.3. The Criojo

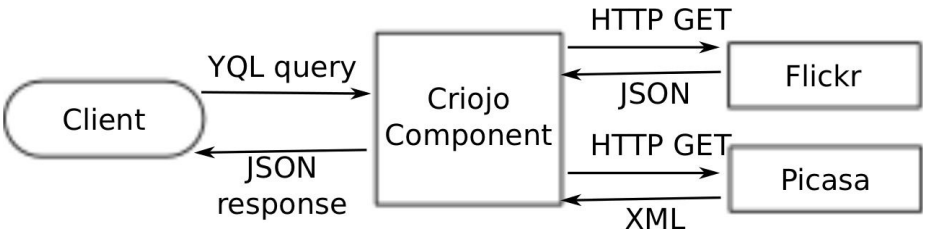

Figure 4.2: Adaptation: Criojo Component Replacing YQL Service

component is composed of a Criojo agent, a wrapper for the Flickr service and a Picasa adapter, and an atom gateway that transforms the YQL request into Criojo format and produces a Json response from the agent response.

The Criojo agent is derived from the XML definition of the open data table. It provides a channel, photos_getCounts, and uses the channel getCount provided by a service wrapping the Flickr service. The resulting program consists of the following rules.

```
1 photos_getCounts(ret, s, from, to) -->
2     (getCount(s, counts, from, to)
```

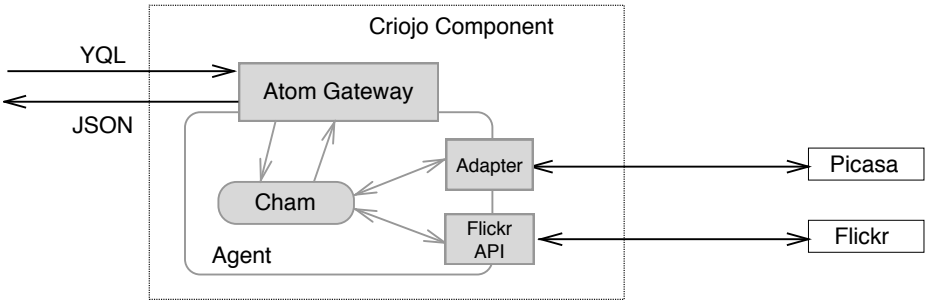

Figure 4.3: Detail of the `Criojo` Component

```
    & Session(ret, s)),
counts(s, n, from, to) & Session(ret, s) -->
    ret(s, n, from, to)
```

In this program, at reception of a request specified with a return channel (`ret`), a session identifier (`s`) and a range (`from` and `to`), the `Flickr` service `getCount` is called. The response over channel `counts` is finally forwarded over the return channel `ret`. By replacing the `Flickr` wrapper with an adapter, we can change the implementation of the service associated to the channel `getCount`, thus changing the provider.

The atom gateway located in the membrane surrounding the agent, corresponding to the implementation of the communication layer, is in charge of translating from a `HTTP` request into an incoming `Criojo` message, and from an outgoing `Criojo` message into a `Json` response. Thus, the previous `YQL` query is transformed into a native atom of the form `_yql(ret, query)`, where `ret` simulates a remote return channel and `query` is a representation of the query. When the agent tries to transmit a message over channel `ret`, the gateway transforms the message into a `Json` format and responds with it to the client.

The native relation `_yql` is associated to a function that treats the `YQL` query and produces the necessary atoms for calling the service. In our case it produces the atom `photos_getCounts(k, "1262307600", "1342132817")`, which triggers the execution of the `Criojo` program.

Communication with services like `Flickr` and `Picasa` is possible thanks to specialized API libraries, implemented using `Criojo` adapters [1] that wrap the services. Thanks to location transparency, we can use the li-

[1] For more on `Criojo` adapters, refer to Section 3.1.1.2.

braries as `Criojo` modules or as separate agents.

Finally, the `Picasa` adapter that simulates the service provided by `Flickr` is implemented as a set of rules that transform a call to a `Flickr` service into a call to a `Picasa` service. These rules can have different levels of granularity depending on whether they are expressed with pure `Criojo` or with impure `Criojo` by using native relations for the transformation. In any case the meaning of the resulting module is expressed by the following rules in pure `Criojo`.

```
1  getCount(s, ret, from, to) −−> (photoCloning(photo, end, s) &
2     Resp(s, ret, from, to, 0)),
3  (photo(s, id, date) & Resp(s, ret, from, to, n)) −−>
4     {date >= from & date < to} ?:
5     (Resp(s, ret, from, to, n+1) & Done(s, id)),
6  (photo(s, id, date) & Resp(s, ret, from, to, n)) −−>
7     {date < from || date >= to} ?:
8     (Resp(s, ret, from, to, n) & Done(s, id)),
9  end(s, id) −−> Todo(s, id),
10 (Todo(s, Succ(id)) & Done(s, Succ(id)) −−> Todo(s, id),
11 (Todo(s, 0) & Done(s, 0) & Resp(s, ret, from, to, n)) −−>
12    ret(s, n, from, to)
```

In this program we use the channel `photoCloning` provided by the `Picasa` API. As its name suggests it, the channel locally clones the information of each photo, taking as parameters a session identifier `s` and two response channels, one for coping the photo information `photo` and another one `end` to indicate the end of the cloning process. First, in line (1), when a `getCount` message is received, the `photoCloning` service is called and the vital information for the request is saved in relation `Resp`. Each response received from the `photoCloning` service through channel `photo` contains a photo identifier and the date in which the photo was taken, among other attributes that we ignore here due to the limited space. The rule in line (3) filters the photos in the date range and increases the count; the rule in line (6) for the photos not taken within the desired range let the count invariant. Each time a photo is processed, an atom `Done` is produced, registering the photo identifier. When the message `end` is received, the identifier of the last photo sent is received. It is then possible to check that all photos have effectively been processed (lines (9) and (10)). If it

is the case, the response is sent via the return channel stored in relation `Resp`: see line (11). It is interesting to remark that the algorithm should be simpler if the channels preserve the emission order.

4.2.2 Integration: the Facade Pattern

Whereas in the previous section we used the adapter pattern to convert the `Picasa` interface to match the requirements of the client, previously working with the `Flickr` service, we now want to work with both `Flickr` and `Picasa`, integrating them into a common service. However, despite the fact that both `Picasa` and `Flickr` services work with photos as resources, each application uses its own data model, whose differences may hinder their integration. To start, `Picasa` is album-centered: photos are accessed via the album they belong to and the same photo cannot be in more than one album at a time, while `Flickr` is photo-centered, allowing for one photo to belong into zero or more sets. Regarding their interfaces, the `Flickr` service provides several methods for obtaining information from photos. `Picasa`, on the other hand limits the methods it provides to a `Restful` style, more oriented towards a use via client libraries. We could use again the adapter pattern to adapt one model to the other; however since their interfaces are so different we decide to define a common model and interface. The resulting service implements the facade pattern, providing a simplified interface for `Picasa` and `Flickr`. Nevertheless, the new service also implements the adapter pattern, by adapting both services to the common model and interface. The communication with the client occurs as in the previous example: `HTTP GET` requests are translated into `Criojo` messages by the atom gateway, which also translates the response into a `HTTP` response, as you can see in Figure 4.4. Note how the architecture of the service resembles that of the adaptation use case.

Let us now show integration in action with the implementation of a service called `myPhotos` that returns a copy of the meta-data of every photo owned by a user. We want the service to build for each photo identified by an `URI id` the following attributes

```
(id, title, description, published_date)
```

and then to send their aggregation back. Here we have to deal with the differences in the information and the way it is obtained from both

services. The Picasa search service returns the following attributes:

```
(id, published_date,
title, summary, author, gid, albumid )
```

while the Flickr search service only returns the id, owner and title of the photo. If we want to obtain more information, we have to call another service called photoInfo for each photo, which returns the following attributes:

```
(id, owner, title, description,
posted_date, taken_date, url, ...  ).
```

In order to provide the required information we implement the following program that queries both services and formats the response according to the common data model. Both services return the search results as a list of photos, whose elements have to be transformed into the data model we have defined. First, the initial request is expanded into two requests to Picasa and Flickr respectively (line (1)) and the vital information for the request (return channel ret, session identifier s, user identifier uid common to two services) is saved in relation Resp.

```
1 myPhotos(ret, s, uid) −−> (Resp(ret, s, uid) &
2     pSearch(pResult, s, uid) & fSearch(fResult, s, uid)),
```

In the case of Picasa, the transformation is straightforward: in line (3) each element of the result from Picasa is transformed into a Photo atom with the desired format.

```
3 pResult(s, uid, (id, date, title, sum, _):: rest) −−>
4     (Photo(s, id, title, sum, date) & pResult(s, uid, rest)),
```

The Flickr result, on the other hand, requires a more elaborated processing, in two steps.

```
5 fResult(s, uid, (id, ow, title)::rest) −−>
6     (fPhotoInfo(s, id, fPhoto, uid) & fResult(s, uid, rest)
7     & Wait(s, id)),
8 (fPhoto(s, uid, id, ow, title, desc, pdate, tdate, url, _)
9 & Wait(s, id)) −−> Photo(s, url, title, sum, date),
```

In line (5) we first send a request for each element of the initial result, asking for more information. Then, in line (8) we transform each `fPhoto` message into a `Photo` atom. Relation `Wait` is used to ensure that all the individual responses have been received. Thus, it is possible to express the transition to the final step as follows.

```
10 pResult(s, uid, Nil) −−> PDone(s),
11 fResult(s, uid, Nil) −−> Abs(Wait(s, _)) ?: FDone(s),
12 (PDone(s) & FDone(s)) −−> Res(s, Nil),
```

The relation `Res` is used to store the result as a list, progressively updated by aggregating the individual photos, and finally sent through the return channel stored in relation `Resp` when there is no photo left.

```
13 (Res(s, plist) & Photo(s, id, title, sum, date)) −−>
14     Res(s, (id, title, sum, date)::plist),
15 (Resp(ret, s, uid) & Res(s, plist)) −−> Abs(Photo(s, _)) ?:
16     ret(s, uid, plist)
```

4.2.3 Coordination: the Mediator Pattern

In the last scenario about coordination, we show how by implementing the mediator pattern with `Criojo` we can combine two scripts written in different languages: one in `YQL` and the other one in a functional language like `Haskell`. Indeed, using `Criojo` as a pivot language allow us to compile the scripts into `Criojo` or integrate them via adapters. Then, with the aid of a mediator component, we can coordinate the resulting programs, thus taking advantage of the features provided by each language.

Concretely, in our example, we have a `YQL` query that returns the user-names of people appearing in the photos that meet a certain search criteria. The query is a join of two open data tables, `photos.search` and `photos.people`, resulting from the call of two services: one for searching the photos by some criteria, and the another one that returns the user-name of people appearing in a given photo. Here is the `YQL` query parametrized with some selection criteria.

```
1 select username, photo_id from photos.people
2 where photo_id in (
3   select photo_id from photos.search
```

```
4  where <some_criteria>) | sort (field="username")
```

The query returns a set of pairs `(username, photo_id)`, sorted by `username`, as indicated by the `sort` function in line (4).

Now, we would like to obtain a result grouped by `username`, in order to generate a `Json` response with the following structure:

```
{
"Bob" : ["photo_1", "photo_2"]
"Carl" : ["photo_3", "photo_2", "photo_4"]
"Edd" : ["photo_1", "photo_3"]
...
}
```

Since this is not possible in `YQL`, we could let the client application manipulate the data; but depending on the language used on the client side this could be more or less tedious. A better alternative is to implement a mediator that coordinates the existing query with another script written in a language where operations such as mapping and grouping are more natural, like a functional language. We can either compile the two scripts into `Criojo` or wrap the programs with `Criojo` adapters in order to use them within the agent. In this example we explore the second option, using a `REST` adapter for communicating with the `YQL` server and another adapter for communicating with the `Haskell` execution environment. Figure 4.5 shows the schema of the mediator pattern in the example.

The mediator takes the `YQL` selection criteria and once embedded in the complete query, forwards it to the `YQL` server via the `REST` adapter, which transforms the message into a `HTTP GET` request and then transforms the resulting `Json` document into an atom. The mediator consumes the result, containing a list of pairs `(username, photoid)` and passes it to the `Haskell` function `groupByKey` that we have created for grouping a set of key-value pairs by key. We can call `Haskell` function from the `Criojo` agent thanks to an adapter that integrates with the `Haskell` platform, by transforming `Criojo` atoms into function calls, and by transforming the values returned by the function into `Criojo` atoms, which the adapter then introduces in the solution. Upon receiving the result of the execution of the `Haskell` function, the mediator exports the answer back to the client via the atom gateway. As usual, the atom gateway acts as a proxy: it receives a `Criojo` message, formats it into a `Json` document

and sends it back to the client. The rules of the mediator component are contained in the following listing.

```
1  yql(ret, s, criteria) −−>
2      (_yql_query(people_photo, s, criteria) & Resp(ret, s)),
3  people_photo(s, list) −−>
4      _hs_fun(result, s, "groupByKey", list),
5  (result(s, list) & Resp(ret, s)) −−> ret(s, list)
```

The rule in line (1) corresponds to the reception of the criteria and its embedding into a `YQL` query sent to the `YQL` service. As usual, the vital information of the request (return channel `ret` and session identifier `s`) is stored in relation `Resp`. The result in the form of a `people_photo` message, containing a list of pairs `(username, photo_id)`, triggers the rule in line (2) that passes the list to the `Haskell` function `groupByKey`. On line (3) the last rule takes the result of the function and sends it back to the client through the return channel.

Note that the use of `Haskell` for the implementation of the `groupByKey` function is only for illustrative reasons, in order to show how it is possible to combine different languages with `Criojo`. In fact, operations such as mapping and grouping are also natural in `Criojo`, given its roots in term rewriting. The following snippet is the `Criojo` equivalent of the `groupByKey` function.

```
1  groupByKey(ret, s, list) −−> (Resp(ret, s) & ByKey(s, list)),
2  (ByKey(s, (key, value)::rest) & Pair(s, key, vlist)) −−>
3      ((Pair(s, key, value::vlist) & ByKey(s, rest)),
4  ByKey(s, (key, value)::rest) −−> {Abs(Pair(s, key, _)} ?:
5      ((Pair(s, key, value::Nil) & ByKey(s, rest)),
6  ByKey(s, Nil) −−> Res(s, Nil),
7  (Pair(s, key, vlist) & Res(s, list)) −−>
8      Res(s, (key, vlist)::list),
9  (Resp(ret, s) & Res(s, list)) −−> {Abs(Pair(s, _, _)} ?:
10     ret(s, list)
```

In line (1), the request is received, which produces an atom `Resp` for the future response, and an atom `ByKey` storing the list of pairs `(key, value)`. The list is recursively processed, populating relation `Pair`, which

assigns to each key a list of values: see lines (2) and (4). From the relation `Pair`, the result is computed (line (7)) and finally sent (line (9)).

To conclude, we have shown how to implement with `Criojo` three design patterns useful for interoperability: Adapter, Facade and Mediator. We were able to make interoperable not only data models but also protocols. Finally, the last example also shows another approach to service oriented computing, by using computations as resources. In fact, in this example the `YQL` script and the `Haskell` program become both resources that are manipulated by the mediator component. Related to this approach is the proposal of Erenkrantz, et al. [EGST07] who come up with an extension of the `REST` style called `CREST` (Computational `REST`), including the notion of mobile code. Thus, resources are not limited to content, but also include computations.

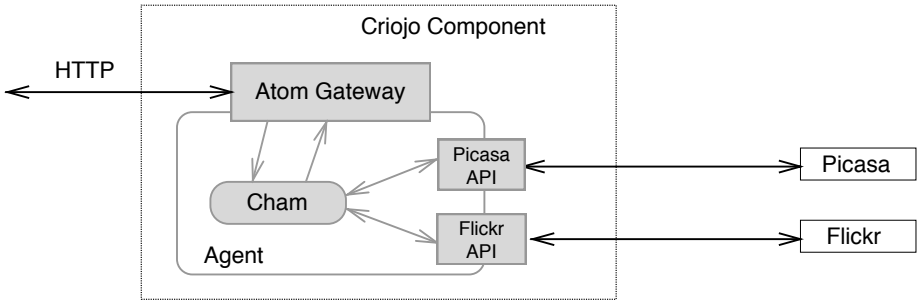

Figure 4.4: Integrating `Picasa` and `Flickr`.

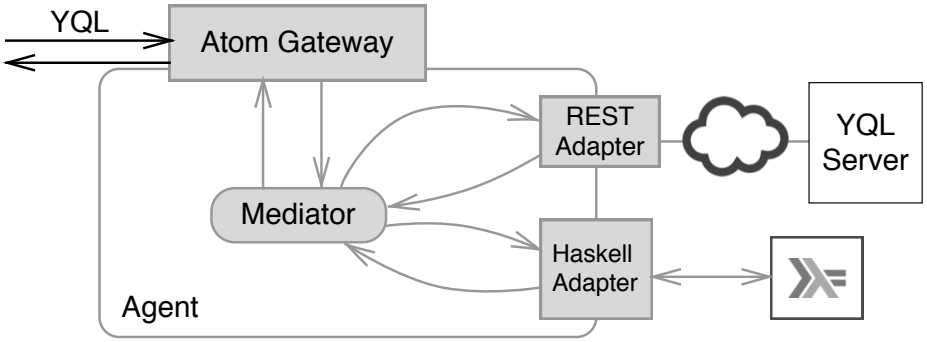

Figure 4.5: Coordinating a `YQL` script and a `Haskell` program with `Criojo`.

Chapter 5

Conclusion

The objective of this thesis has been

- to prove that the chemical programming paradigm is a good solution for the orchestration of services in network-based architectures;
- to propose a practical tool, in the form of a programming language, whose theoretical foundations rests on the Heta-calculus, a chemical abstract machine dedicated to service orchestration;
- to use this language as a pivot language in order to solve interoperability issues.

In order to determine what is needed in a language for orchestrating services, we studied the state of the art in Chapter 1, where we began by providing the basic concepts to understand distributed computing and service oriented computing. This chapter provides an analysis of the various approaches towards distributed programming, dealing with concerns such as (i) distribution models, with the opposition message-passing versus shared memory, (ii) communication, from synchrony to asynchrony, (iii) parallelism and concurrency and (iv) fault tolerance. The modern form of distributed computing turns out to be service-oriented computing, which derives from a trend towards an Internet-wide decentralization. For web services, we find two popular and often antagonistic models: one centered on processes, and another one centered on resources. As a consequence of the absence of a unified model for service-oriented computing, new interoperability problems appear in the orchestration of heterogeneous agents. Approaches to tackle these issues include solutions based on message oriented middlewares and model driven engineering techniques, which in general propose an intermediary protocol for the interoperability of agents. Since the solution always relies on the existence of a common resource model, we also study different approaches towards a universal model for representing and manipulating resources.

At this point, the chapter leads to the first contribution of the thesis: a set of requirements for an orchestration language, drawn from the state of the art. Here is a summary.

- Message-passing model with asynchronous communication
- Channel mobility
- Shared memory with locks and transactions for each agent
- Parallelism explicit globally, implicit locally
- Fail-safe fault tolerance
- Correlation between services
- Resource representation with a universal data model
- Computational completeness with respect to representations

With these requirements in mind, we presented in Chapter 2 the Heta-calculus, a calculus for service orchestration developed in the *Ascola* team. The syntax and the semantics of the Heta-calculus is defined via a distributed chemical machine: it describes collaborations between agents as well as the behavior of the agents themselves. The state of the agent is described in terms of a chemical solution that changes according to reaction rules, and agents communicate with each other by exchanging messages in the form of atoms. Contrary to the classical cham, the Heta-calculus has introspection, which extends its expressive power.

The second contribution of this thesis is the retrospective formulation of all the design decisions that have been made for the design of the Heta-calculus and its validation against requirements. See Table 5.1 for a complete check-list.

Nevertheless, the Heta-calculus is only a minimal formalism for describing service orchestrations. The third contribution of this thesis, presented in Chapter 3, is a programming language, `Criojo`, based on the Heta-calculus, for its syntax and its semantics. `Criojo` also concretizes some abstract features of the Heta-calculus, only useful for modeling. Thus, `Criojo` concretely implements the capacities for interfacing with external components, only defined as an abstract promise in the Heta-calculus, which greatly improves interoperability, particularly by satisfying the black-box principle. The first part of the chapter presents a

	Heta-calculus
Distributed Architecture	
Asynchronous message-passing	✓
Channel Library	-
Message Passing	
Channel Scope	✓
Channel mobility	✓
Scope Extrusion	✓
Agent Architecture	
Shared memory with locks	✓
Transactions	-
Parallelism	
Implicit Parallelism (locally)	✓
Explicit Parallelism (globally)	✓
Fault Tolerance	
Fail-safe fault tolerance	✓
Failure Detection/Notification	-
Logging	-
Services and Resources	
Correlation	✓
Interface	✓
Representation	✓
Typing	-
Completeness	-
Map/Reduce	✓

(Satisfied: ✓, Not Satisfied: ×, To be Completed: -)

Table 5.1: Heta-calculus – Validation against Requirements

tutorial explaining how to program with `Criojo`: the aim was to show that `Criojo` programs are concise, quite declarative, expressed at a high level of abstraction. The second part of the chapter details the implementation of `Criojo`, providing a set of guidelines for future implementations and extensions. Table 5.2 describes the state of the `Criojo` implementation. Although a lot of work must still be done, the language `Criojo` already turns out to be useful.

Indeed, Chapter 4 shows `Criojo` features applied in a real-world scenario: this is the fourth contribution of the thesis. We were able to specify a solution for the interoperability problems between `Flickr` and `Picasa`, two providers for photo services, thanks to a pivot architecture with `Criojo` as pivot language. The pivot architecture provides a framework for the solution of the adaptation, integration and coordination problems by implementing three well known design patterns: the adapter, facade and mediator patterns.

Finally, if the current implementation of `Criojo` proved to be a useful tool for service orchestration by extending the Heta-calculus without modifying its semantics, it remains a prototype and a lot of work still needs to be done. We now present the perspectives of this work.

5.1 Future work

We present some possible extensions that we consider as priorities. If most of them come from the requirements (see Table 5.2), we have also identified one of them for the development process of the language.

Causality and Synchronous communication. Certain algorithms require to preserve the order of events from emission to reception. This is called a causally ordered computation [CBMT96], which can be seen as a generalization of synchronous communication. Let us illustrate this by revisiting the `SVGPainter` example in Section 3.1.1.3. Assume in this case that we use an implementation of the `SVGPainter` as an independent agent to which our Sierpinski agent sends messages and that we add two operations, for opening and closing a file respectively. The algorithm can now be summarized as:

1. Send a message to open the file
2. Generate the Sierpinski triangles

	Criojo
Distributed Architecture	
Asynchronous message-passing	✓
Channel Library	×
Message Passing	
Channel Scope	-
Channel mobility	-
Scope Extrusion	×
Agent Architecture	
Shared memory with locks	✓
Transactions	-
Parallelism	
Implicit Parallelism (locally)	×
Explicit Parallelism (globally)	✓
Fault Tolerance	
Fail-safe fault tolerance	✓
Failure Detection/Notification	×
Logging	×
Services and Resources	
Correlation	✓
Interface	✓
Representation	✓
Typing	-
Completeness	-
Map/Reduce	-

(Implemented: ✓, Not Implemented: ×, Partially implemented: -)

Table 5.2: Criojo – Validation against Requirements

3. Send a `paintTriangle` message for each triangle
4. Close the file

Notice that if we proceed as indicated by the algorithm, there is no way of guaranteeing that the `SVGPainter` agent has received all the `paintTriangle` messages before closing the file due to the asynchronous nature of the communication and the latencies of the network. Of course it is possible to implement control mechanism for preserving causality in `Criojo`, in the form of acknowledgment messages; however, this can become cumbersome and result in boilerplate code. Thus a desirable extension of the language would be the introduction of richer channels, capable of supporting causally ordered messages above asynchronous communication or by directly using synchronous protocols in a native way.

Parallelization. With the emergence of multicore processors, distributed computing takes another dimension: computations needing multiple independent computers can be done within a single processor. An example of this trend is `Intel`'s experimental single-ship cloud processor that aims at simulating a scalable cluster of up to one hundred computers that communicate with each other via hardware supported message passing[1]. Traditional sequential programming is less adapted to this technology than parallel computing oriented languages like `Criojo`. However, currently, the implementation of `Criojo` does not resort to parallelism. One possible solution towards parallelization would be to add the capacity of deploying independent chams in the same computer, each within a dedicated core. Another solution would be to allow the parallel execution of rules inside a group. The problem can then be stated as a scheduling problem with conflicts [EHKR09]. If the problem has not been studied in the chemical model at the processor scale, it has been studied at the network scale, where algorithms have been proposed for the discovery and capture of atoms [BOT13].

Optimal Rule Execution. The current implementation of the reaction rules closely reflects the semantics expressed in the theory of the Heta-calculus, and guarantees the correct evaluation of rules. Nevertheless, automata are not the most effective implementation. Consider for instance the following program that eliminates duplicated atoms.

[1] See the web site

```
1 R(x) & R(x) -—> R(x)
```

The problem with implementing this rule as a state machine is that it leads to a Cartesian product on the total of atoms, resulting in a complexity of order $O(n^2)$, when it could be linear. The problem is even harder when we also consider the evaluation of guards. As said before, there is a balance to be found between the expressive power of guards, required for computational completeness, and the complexity of their evaluation, which should be not only reasonable but also predictable in an intuitive way.

Language Integration. Depending on what we want to do, there are two ways for the integration of `Criojo` with other languages, and extensions are required for each of these methods. First, adapters allow to reuse existing components written in different languages and to take advantage of the features of a given language. This is illustrated by our coordination example in Section 4.2.3, where `Criojo` is used for coordinating a `SQL` script with a functional language script. The next step is to define adapters for several different languages. The second way in which `Criojo` is integrated with other language, is through embedding: `Criojo` is implemented in a host language and can be used within that language, as exemplified by the current prototype. The difficulty lies in finding an effective way of using the native data types. Our current approach is based on a boxing/unboxing technique, where native types are wrapped inside `Criojo` terms. Implicit conversions allow to box/unbox values, thus enabling the use of native operations within expressions. However, this implies that wrappers and conversions have to be implemented for each native type we want to use within the rules. Moreover, boxing and unboxing is already done by languages like `Scala` and C#, which means that a double boxing and unboxing is performed each time. In consequence, we need a more practical and efficient way of integrating native types into the language.

To conclude, the language `Criojo` provides a basis for many future extensions, either in the current prototype, or in new prototypes using new host languages.

Appendix A

Résumé Étendu

A.1 Introduction

Avec l'émergence du "Cloud-computing" et des applications mobiles, il est possible de trouver un service web répondant à presque tout besoin. Un service est un programme informatique qui fournit un ensemble d'opérations accessibles à partir d'une adresse de réseau. Les programmes clients sur le web interagissent avec le service en utilisant des messages `HTTP`. Grace à la variété de services Web disponibles, les développeurs peuvent créer des applications complexes en combinant plusieurs services indépendants, dont la disposition et l'exécution peut être automatisé à l'aide de langages d'orchestration.

Cependant, la diversité des technologies et le manque de standardisation peuvent entraver la collaboration entre services. Imaginez par exemple que vous écrivez une application mobile de gestion et de partage de photos. Il existe différents services qui permettent de gérer des photos en ligne, dont `Flickr` est un des plus populaires. En utilisant l'interface de programmation (API) de `Flickr` vous implémentez une application qui communique avec ce service. Mais une fois que l'application gagne en popularité, de plus en plus d'utilisateurs demandent à pouvoir utiliser des services alternatifs comme `Picasa`. Néanmoins, `Flickr` et `Picasa` diffèrent non seulement dans la façon dont ils organisent les photos, mais aussi dans les services qu'ils fournissent par leurs interfaces. Par protocoles on entend la manière dont les messages sont organisés pour compléter des tâches communes, comme la recherche et l'édition : alors que `Flickr` fournit directement une variété des opérations plus ou moins complexes, `Picasa` s'appuie sur les bibliothèques clientes qui effectuent les mêmes tâches en combinant des méthodes `HTTP` de base `GET`, `POST`, etc. D'un point de vue technologique, bien que les deux services fournissent des interfaces `REST`, `Flickr` permet également aux clients d'utiliser du `SOAP` et du `XML-RPC`.

On se trouve donc face à des problèmes d'*interoperabilité* dus à l'hétéro-

génétité des différents services. A cet égard, nous avons identifié trois types de problèmes, adaptation, intégration et coordination, qui peuvent être décrits comme suit dans les termes de notre exemple :

Adaptation : l'application cliente qui orchestre les services de `Flickr` doit être adaptée pour orchestrer les services fournis par `Picasa`.

Intégration : l'application cliente doit orchestrer en même temps les services de `Picasa` et de `Flickr` en définissant un modèle de données commun et une interface commune aux deux services.

Coordination : du point de vue des langages d'orchestration de web services, les scripts existants écrits dans différents langages doivent être coordonnés pour coopérer dans l'orchestration des services utilisés.

Les infrastructures de type middleware sont généralement proposées pour résoudre les problèmes d'interopérabilité sous la forme d'architectures de bus avec un élément central qui traduit des messages. Néanmoins, une solution complète nécessite une représentation universelle des ressources. Notre approche, analogue à ces travaux consiste en une architecture pivot qui intègre différents langages d'orchestration avec des fournisseurs de services hétérogènes autour d'un langage pivot, permettant ainsi la mise en oeuvre de patrons de programmation courants : le patron adaptateur pour résoudre des problèmes d'adaptation, le patron façade pour résoudre des problèmes d'intégration, et le patron médiateur pour résoudre des problèmes de coordination. Le défi reste de trouver le langage d'orchestration adéquat qui puisse servir de langage pivot.

La thèse de cette dissertation est que le paradigme de programmation chimique peut fournir les fondations pour un langage d'orchestration. Concrètement,

- nous présentons un nouveau langage d'orchestration, appelé `Criojo`, qui met en oeuvre et étend un calcul original basé sur une machine chimique abstraite (cham) dédiée à la programmation orientée aux services,
- nous montrons comment le langage d'orchestration peut être utilisé pour définir une architecture pivot.

La conséquence à adopter cette approche serait une amélioration de l'interoperabilité des services et des langages d'orchestration, facilitant

ainsi le développement de services composés. Le haut niveau d'abstraction de Criojo pourrait permettre aux développeurs d'écrire des programmes très concis puisque les échanges de messages sont représentés de manière naturelle et intuitive. Ces programmes pourraient être utilisés non seulement comme orchestrations efficaces, remplaçant les orchestrations écrites dans des langues traditionnelles, mais aussi comme prototypes d'orchestrations, donnant une spécification claire pour les orchestrations concretes écrites dans des langues traditionnelles. En outre, le fondations formelles de `Criojo` fournissent une spécification du noyau d'un langage d'orchestration pour une architecture pivot, ce qui conduit à de nombreux avantages, non seulement pendant la phase de développement du langage, mais aussi pendant la description et les phases de validation des orchestrations écrites dans ce langage.

- La spécification formelle étant claire et concise facilite la mise en oeuvre du langage, tout en évitant les pièges souvent rencontrés dans les normes, comme est le cas du langage `BPEL`.

- La spécification formelle fournit les bases théoriques des outils utiles pour spécifier, tester et vérifier des orchestrations.

A.1.1 Contributions

Cette dissertation fait les contributions suivantes :

- une définition bien motivée d'un ensemble de conditions requises pour un langue d'orchestration,

- la formulation de toutes les décisions de conception qui ont été prises pour la conception du calcul chimique et de sa validation par rapport aux besoins, en plus de la présentation du calcul initial,

- la mise en oeuvre du prototype d'un langage d'orchestration appelé `Criojo`, basée sur les fondements théoriques données par le calcul chimique, décrit du point de vue d'un programmateur et d'un exécutant, respectivement,

- en particulier, un ensemble d'extensions utiles qui facilitent la programmation des vrai applications, comme la possibilité de s'interfacer avec des fonctions et des ressources externes,

- une méthode pour le développement de solutions différentes pour des problèmes d'interopérabilité, sous la forme d'une architecture de pivot utilisant Criojo en tant que langue de pivotement.

A.2 Vers un Langage pour l'Orchestration de Web Services

En vue de déterminer ce qui est nécessaire dans un langage pour orchestrer des services web, nous avons étudié l'état de l'art dans ce chapitre, où nous avons commencé par exposer les concepts de base permettant de comprendre la programmation distribuée et la programmation orientée service. Ce chapitre fournit une analyse des différentes approches dirigées vers la programmation distribuée, traitant de sujets comme (i) les modèles de distribution, avec l'opposition entre le modèle basé sur l'échange de messages et le modèle de memoire partagée, (ii) communication, en partant de la communication asynchrone vers la communication synchrone, (iii) parallélisme et concurrence et (iv) tolérance aux fautes. La forme moderne de la programmation distribuée s'avère être la programmation orientée service, qui découle d'une tendance vers une décentralisation à l'échelle d'Internet. Dans les services web nous trouvons deux modèles populaires et souvent antagonistes : l'un centré sur les processus et l'autre sur les ressources. Comme conséquence de l'absence d'un modèle unifié pour la programmation orientée service, de nouveaux problèmes d'interoperabilité apparaissent dans l'orchestration d'agents hétérogènes. Les approches pour adresser ces questions incluent des solutions basées sur des middle-wares et des techniques d'ingénierie dirigée par les modèles, qui proposent généralement un protocole intermédiaire pour l'interoperabilité des agents. Comme les solutions se reposent encore sur l'existence d'un modèle de ressources commun, nous étudions aussi différentes approches vers un modèle universel pour la représentation et la manipulation des ressources. A la fin du chapitre, nous listons un ensemble de besoins extraits de l'état de l'art, du point de vue de la programmation orientée service et du point de vue de la manipulation de ressources. Ces besoins fournissent la spécification suivante pour un langage pour l'orchestration de services web.

A.2.1 Exigences pour Orchestration de Service

Communication, Synchronisation et Parallélisme

Condition 1 (Architecture Distribuée - Echange des Messages)**.** Le langage doit permettre la définition des orchestrations réparties entre des agents. Il doit utiliser un modèle de passage de messages : les agents

échangent des messages sur des canaux.

Condition 2 (Architecture Distribuée - Canaux Asynchrones)**.** Le langage doit utiliser des canaux asynchrones, ce qui est naturel dans un contexte distribué.

Condition 3 (Architecture Distribuée - Bibliothèque de Canaux)**.** Le langage doit fournir une bibliothèque des canaux avec les conditions suivantes de synchronisation :

(i) *synchronie*,

(ii) *préservation de l'ordre causal*,

(iii) *diffusion*

et probablement d'autres.

Condition 4 (Passage des Messages - Portée de Canaux)**.** La portée d'un canal doit être contrôlée.

Condition 5 (Passage des Messages - Extrusion de la Portée)**.** La portée d'un canal transmis à un agent devrait être étendue à la réception l'agent.

Condition 6 (Architecture de l'Agent - Verrous)**.** Le langage doit fournir une primitive pour verrouiller les ressources.

Condition 7 (Architecture de l'Agent - Transactions)**.** La langage devrait permettre un mécanisme transactionnel à programmer pour chaque agent. Les transactions doivent satisfaire les conditions suivantes :

(i) atomicité,

(ii) isolation

Condition 8 (Parallélisme - Explicite Globalement, Implicite Localement)**.** La définition des agents distribués agissant en parallèle doit être explicitement indiqué. Pour chaque agent, le parallélisme entre les activités locales devrait être implicite.

Tolérance Aux Fautes Maintenant nous définissons les conditions requises pour la tolérance aux fautes. Nous nous limitons aux omissions et erreurs accidentelles. Ainsi, il est de la responsabilité du programmeur d'assurer un comportement correct en présence de fautes byzantines : cette tolérance peut être assurée par la sécurité de recourir à la cryptographie.

Condition 9 (Tolérance Aux Fautes - Fail-safe)**.** La langue doit appliquer une tolérance aux fautes dite *fail-safe* préservant les invariantes locales. Cela peut imposer une tolérance aux fautes plus forte.

Une invariante locale est une propriété satisfaite par un agent et conservée au cours de l'exécution. La première partie de l'exigence énonce une tolérance aux fautes minimale : en cas de perte d'un message ou d'un accident dans l'agent, chaque agent actif se comporte toujours correctement. Au-delà de ce seuil minimal, la tolérance aux fautes devient coûteuse : toute extension de la sécurité globale ou de la vivacité est donc facultative.

Condition 10 (Tolérance Aux Fautes - Détection et Notification)**.** Le langage doit fournir des mécanismes de détection et de notification des fautes d'omission et de crash.

La mise en œuvre de ces mécanismes dépend de la couche physique sous-jacente utilisée pour communiquer. Ainsi, l'exigence pourrait être impossible à satisfaire en raison du manque de fonctionnalités.

Condition 11 (Tolérance Aux Fautes - Enregistrement)**.** Le langage doit fournir des mécanismes pour enregistrer les événement ou actions.

Services et Ressources Dans cette partie, nous décrivons les conditions requises spécifiques pour les services et les ressources.

Condition 12 (Services - Corrélation)**.** Le langage doit fournir une primitive ou un mécanisme de corrélation des messages.

Condition 13 (Ressources - Interface)**.** Le langage doit fournir un mécanisme pour s'interfacer avec n'importe quelle ressource.

Pour les ressources internes, l'exigence correspond à la possibilité de nommer et représenter une ressource dans le language. Pour les ressources externes, l'exigence vise à améliorer l'interopérabilité.

Condition 14 (Ressources - Représentation)**.** Le langage doit fournir un modèle de données universel avec les propriétés suivantes:

(i) les données sont lisibles par l'homme

(ii) les données peuvent avoir un analysées syntaxique effectif

(iii) les données sont sérlialisables

Un modèle de données est universel s'il permet de représenter tout modèle de données, en particulier le modèle algébrique, le modèle relationnel et d'autres utilisés pour les données semi-structurées.

Condition 15 (Ressources - Types)**.** Si le langage est typé, son système de types peut fournir les opérations ensemblistes : l'union, l'intersection et la différence, et interpréter la relation de sous-typage comme une relation d'inclusion de sous-ensembles.

Cette exigence optionnelle s'appuie sur le succès rapporté par Benzaken et al. [BCNS13] lors de la formalisation d'un modèle de données pour données semi-structurées.

Condition 16 (Ressources - Exhaustivité Computationalle)**.** Le langage doit être complet par rapport au modèle de données.

En d'autres termes, toutes les fonctions calculables sur le modèle de données doit être exprimée dans le langage : c'est la thèse de Church appliquée au modèle de données universel. Concrètement, cela signifie que n'importe quel langage défini sur le modèle de données peut être traduit, ce qui peut être expérimenté avec des langages fonctionnels, logiques, et impératifs, par exemple.

Condition 17 (Services - Map/Reduce)**.** Le langage doit fournir un mécanisme pour l'implémentation des operations *Map/Reduce*.

A.2.1.1 Orchestration des services dans la pratique

Nous analysons les conditions requises par rapport aux deux solutions existantes possibles : des frameworks orientés aux objets pour l'implémentation de services `Restful` et `WS*`, comme `CXF` ; et des langages d'orchestration comme `BPEL`, qui est le standard de facto. Le résultat est résumé dans le Tableau A.1.

	Frameworks `Restful` / `WS*`	`BPEL`
Passage de messages asynchrone	✓	✓
Librairie de canneaux	-	-
Mobilité de canneaux	✓ / -	-
Mémoire partagée avec verrous	✓	✓
Transactions	-	✓
Parallélisme Explicite (local)	-	✓
Parallélisme Implicite (local)	×	×
Parallélisme Explicite (global)	×	×
Tolérance aux fautes *fail-safe*	✓	✓
Détection et Notification de Fautes	-	✓
Enregistrement	-	-
Corrélation	- / ✓	✓
Réssource Interface	✓	✓
Modèle des Données Universel	×	✓
Complétude	✓	✓
Map/Reduce	✓	×

(Yes: ✓, No: ×, Partly: -)

Table A.1: Satisfaction des Exigences en Pratique

A.3 Le Heta-calcul

Dans ce chapitre nous présentons l'Heta-calculus, en prenant en compte les besoins du chapitre precedent. L'Heta-calculus est un calcul pour la formalisation de la collaboration entre agents qui a été développé au sein de l'équipe *Ascola*. La syntaxe et la sémantique du Heta-calculus est celle d'une machine chimique distribuée, elle décrit aussi bien les collaborations entre agents que le comportement des agents eux mêmes. La définition formelle de la syntaxe du Heta-calculus est montrée dans le Tableau A.2.

Value Pattern	v	$::=$	$f\ v^*$	(term)
		$\mid$	V	(variable)
Atom Pattern	a	$::=$	$R(v)$	(atomic fact)
		$\mid$	c	(cell)
		$\mid$	A	(variable)
Cell Pattern	c	$::=$	$M[s]$	(membrane with solution)
Solution Pattern	s	$::=$	$\emptyset$	(empty solution)
		$\mid$	$a \,\&\, s$	(insertion)
		$\mid$	S	(variable)
Program	p	$::=$	$c\,\{r^*\}$	(initial cell { rules })
Rule	r	$::=$	$c \to g\,?\,c$	(head $\to$ guard? conclusion)
Guard	g	$::=$	$\top$	(true)
		$\mid$	$\bigwedge g^*$	(conjunction)
		$\mid$	$\neg(c \to g)$	(control guard)

Table A.2: Machine Chimique Abstraite Introspective – Syntaxe

L'état de l'agent est décrit en termes d'une solution chimique qui change suivant des règles de réaction, et les agents communiquent entre eux en échangeant des messages sous forme d'atomes. Les principales différences avec la machine classique sont

- La machine abstraite chimique est introspective, grâce aux gardes de contrôle. C'est le point majeur.

- Les règles sont plus effectives car il n'y a pas des règles réversibles
- Les solutions chimiques sont décrites par des patrons qui peuvent recourir à une solution variable, avec une occurrence unique, ce qui mène à une formulation plus générale tout en évitant un couplage complexe.

La sémantique du calcul est détaillé dans le Tableau A.3.

Value	ξ	$::=$	$f\,\xi^*$	(term)
Atom	α	$::=$	$R(\xi)$	(atomic fact)
		$\mid$	γ	(cell)
Cell	γ	$::=$	$M[\sigma]$	(membrane with solution)
Solution	σ	$::=$	$\emptyset$	(empty multiset)
		$\mid$	$\alpha \,\&\, \sigma$	(multiset insertion)

$$
\begin{array}{lllll}
\gamma & \models_\tau & \top & \stackrel{\text{def}}{\Leftrightarrow} & \top \\
\gamma & \models_\tau & \bigwedge_i g_i & \stackrel{\text{def}}{\Leftrightarrow} & \bigwedge_i (\gamma \models_\tau g_i) \\
\gamma & \models_\tau & \neg(c \to g) & \stackrel{\text{def}}{\Leftrightarrow} & \neg(\exists \tau'.(\gamma = c[\tau.\tau']) \wedge (\gamma \models_{\tau.\tau'} g))
\end{array}
$$

$$
\frac{(c_1 \to g \,?\, c_2 \in p) \quad (c_1[\tau] \models_\tau g)}{c_1[\tau] \Rightarrow c_2[\tau]} \quad [\texttt{REACTION CHIMIQUE}]
$$

$$
\frac{\gamma_1 \Rightarrow \gamma_2}{M[\gamma_1 \,\&\, \sigma] \Rightarrow M[\gamma_2 \,\&\, \sigma]} \quad [\texttt{MEMBRANE}]
$$

Table A.3: Machine Chimique Abstraite Introspective – Sémantique

Le chapitre conclut avec la deuxième contribution de cette thèse qu'est la formulation rétrospective de toutes les décisions de conception qui ont été faites pour la conception du 'Heta-calculus et sa validation par rapport aux besoins. Le Tableau A.4 donne un résumé de cette validation.

	Heta-calculus
Architecture Distribuée	
Passage de Messages Asynchrone	✓
Bibliothèque de canaux	-
Passage de Messages	
Portée de Canaux	✓
Mobilité de Canaux	✓
Extrusion de la Portée	✓
Architecture des Agents	
Mémoire partagée avec verrous	✓
Transactions	-
Parallélisme	
Parallélisme Implicite (local)	✓
Parallélisme Explicite (global)	✓
Tolérance aux Fautes	
Tolérance aux fautes Fail-safe	✓
Détection de failles / Notification	-
Logging	-
Services et Ressources	
Corrélation	✓
Interface	✓
Représentation	✓
Typage	-
Completude	-
Map/Reduce	✓

(Satisfait: ✓, Non Satisfait: ×, A être complété: -)

Table A.4: Heta-calculus – Validation contre les Besoins

A.4 Criojo Pratique

Criojo permet la définition des agents par un ensemble de règles, en suivant un schema général. Dans ce schema, nous partons d'une description donnée pour générer les règles. Ensuite, les règles générées sont exécutées sur une machine chimique. Il existe plusieurs possibilités pour la mise en oeuvre du langage : soit via un compilateur ou une interprétation sur une machine virtuelle, ou les deux, en fonction du niveau d'abstraction de la machine virtuelle. Notre choix qui repose sur Scala, a été de décrire les règles directement dans le langage hôte sous la forme d'un langage interne dédié (DSL), ce qui était la meilleure option pour un prototype, et ensuite d'implémenter la machine chimique comme un interprèteur de règles. Dans cette section nous présentons le langage du point de vue du dévelopeur et du réalisateur.

A.4.1 Programmation avec Criojo en Scala

Criojo est implémenté comme une API Scala avec un langage interne dédié (DSL interne) imbriqué dans un langage hôte (Scala). Criojo utilise un sous-ensemble de la grammaire de Scala, et ajoute de nouvelles fonctionnalités sans réellement modifier le langage hôte. Le principal avantage d'un DSL interne est qu'il n'a pas besoin d'un compilateur, donc l'implémenteur peut se concentrer sur la mise en œuvre des fonctionnalités sémantiques du langage intégré sans se préoccuper de la syntaxe. Cependant, les DSL internes sont en quelque sorte limités par le modèle du langage hôte : le système de types et les constructions syntaxiques disponibles.

A.4.1.1 Définition des Agents

Un agent est une unité de calcul indépendant, dont l'état est représenté par une solution chimique et dont le comportement est défini comme un ensemble de règles de réaction. Les ressources de l'agent sont représentées en termes de structures relationnelles : un prédicat appliqué à des termes exprime un fait atomique et définit une relation, considéré comme un multi-ensemble. Le multi-ensemble d'atomes dans l'agent constitue son état sous la forme d'une solution chimique. L'état de l'agent est modifié par des règles de réaction qui génèrent de nouveaux atomes en consommant des atomes existants de la solution. Certains des nouveaux atomes

restent dans la solution locale, tandis que d'autres sont exportées, selon le type de relation qui les définit. Il existe deux types de relations dans un agent : relations locales, qui sont utilisées seulement en interne au sein de l'agent ; et canaux, qui permettent la communication avec d'autres agents, en transportant des messages (atomes) d'un agent à l'autre. Une explication plus détaillé de la syntaxe et des examples se trouvent dans la version longue en anglais de cette thèse.

`Criojo` comporte trois caractéristiques fondamentales : l'introspection, l'adaptation et la modularité.

Introspection Grace aux gardes, les agents sont capables d'introspecter sur leur propre état. Donc, une règle ne peux s'executer que si sa garde est satisfaite. La syntaxe de gardes en `Criojo` est la suivante:

Guard	g	::=	`True`		
		|	g && g	(et)	
		|	$g \parallel g$	(ou)	
		|	`Not`$(s \to g)$	(control)	
		|	`Abs`(s)	(absence)	| x `op` y (garde native)

`Criojo` étend l'Heta-calculus avec des gardes natives qui sont basées sur des tests natIfs réalisés dans le langage hôte.

Adaptation Les adaptateurs en `Criojo` permettent la collaboration entre les agents et des composants externes. Un adaptateur encapsule un composant externe, en fournissant une abstraction en termes bien adaptés à la machine chimique. Concrètement, un adaptateur simule un ensemble de règles qui génèrent les atomes correspondant aux ressources fournies par le composant enveloppé, et cet ensemble peut être infini. Les adaptateurs en `Criojo` sont définis comme des types spéciaux des relations appelées *Relations Natives* Une relation native est associée à une fonction native qui transforme un ensemble de termes en une molecule, c'est à dire, un ensemble d'atomes.

Modularité La modularité et la séparation des préoccupations peuvent être atteints dans `Criojo` grâce à la collaboration entre agents. Une autre façon de garantir le principe de la responsabilité unique est en

factorisant le comportement dans des modules indépendants qui peuvent être combinés plus tard dans un seul agent.

A.4.2 Vers une implementation efficace

La hiérarchie des agents communicants (Fig. A.1) est définie par les composants ci-dessous:

Orchestrateur : implementation d'un orchestrateur qui contient une solution chimique et un ensemble de règles de réduction.

Gateway : un firewall qui fait part d'un orchestrateur. Sert à envoyer et recevoir des messages.

Firewall : l'implementation d'un firewall pure utilisé dans la transmission de messages.

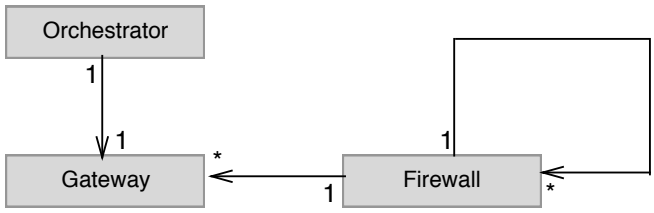

Figure A.1: Diagramme de Classes – Composants de l'Architecture.

L'execution des orchestrateurs sont organisés par rounds. Un orchestrator produit et consomme des atom à chaque round. Au début d'un round, il met à jour sa solution chimique avec les messages entrants, calcule les multi-ensembles des atomes consommées et produites par le déclenchement d'une ou plusieurs règles de réduction, met à jour enfin sa solution chimique et le multi-ensemble de messages sortants. À la fin du round, les messages produits par un orchestrateur sont envoyés à son gateway. Les messages sortants sont transmis par le gateway à son firewall parent, les messages internes sont re-envoyés à l'orchestrateur. Au lieu de calculer toute la solution chimique à chaque tour, nous avons recours à une stratégie efficace : *l'incrementalisation* [Liu00]. Ainsi, à chaque exécution nous calculons la difference entre l'état precedent et le nouvel état.

Pour calculer l'ensemble des évaluations de molecules candidates lors d'un round, nous avons besoin de trouver une correspondence entre une

séquence d'atomes dans la solution et le patron de la tête de la règle. Pour l'incrémentalisation, nous memoïsons des solutions partielles au cours d'un round. Ainsi, l'état de correspondences pour une règle est représenté par une machine à état. Une solution similaire a déjà été étudiée dans le système `jocaml` [FM98], l'une des implémentations du join-calculus, où des patrons de jointure sont compilés dans des machines à états finis.

A fin d'expliquer l'execution d'une règle avec une machine à état, prenons comme example la règle:

$$A(x) \,\&\, B(x,y) \to C(x,y)$$

Cette règle est représentée par la machine d'état de la Fig. A.2. Au départ, la solution et la machine d'état sont vides. Tout d'abord, l'atome $B^1(a,b)$ arrive et la valuation $\{x = a, y = b\}$ est ajoutée à l'état (01). Ensuite, l'atome $A^2(c)$ arrive et la valuation $\{x = c\}$ est ajoutée à l'état (10); mais, puisqu'il n'y a pas de correspondance avec $A(x)[x = a; y = b]$, aucune valuation est ajoutée à l'état (11). Enfin, l'atome $A^3(a)$ arrive et la valuation $\{x = a\}$ est ajoutée à l'état (10). Parce que cette fois il y a une correspondance, la valuation $\{x = a, y = b\}$ est ajoutée à l'état (11). Par ailleurs, chaque valuation est couplée avec une séquence des identificateurs des atomes qui les produisent. L'évaluation de gardes s'avère moins triviale. L'incrementalisation efficace de cette évaluation est difficile en raison de la liaison de variables entre les différents niveaux dans la règle, puisque toute les têtes ont des variables liées dans ces gardes. L'implémentation actuelle ne recourt pas à l'incrementalisation pour l'évaluation des gardes. Cependant, l'impact n'est pas trop fort parce que la structure de pré-règles est plutôt plate, avec généralement zéro ou un niveau de gardes.

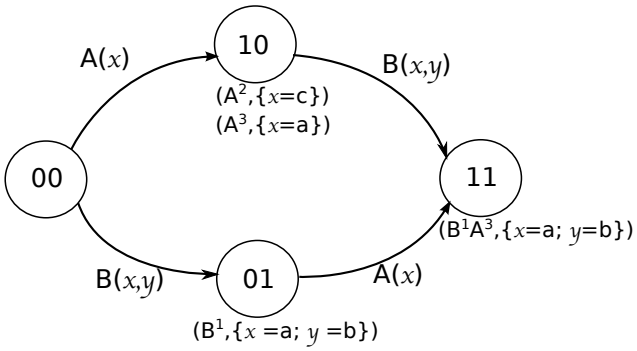

Figure A.2: La machine d'états correspondant à la règle $A(x) \,\&\, B(x,y) \rightarrow C(x,y)$ après avoir reçu les atomes $B^1(a,b), A^2(c)$, et $A^3(a)$.

A.5 Une Solution Pivot pour les Problèmes d'Interopérabilité

L'interopérabilité, dans le contexte de la programmation orientée services n'est pas toujours simple. Supposons, par exemple, que vous souhaitez automatiser l'organisation de vos photos, qui sont gérées par deux systèmes de gestion de photos différentes, comme Picasa et Flickr. Vous pouvez rapidement faire face à des problèmes d'interopérabilité, à savoir l'adaptation, l'intégration et les problèmes de coordination. En effet interfaces Picasa et Flickr diffèrent du point de vue fonctionnel : les deux interfaces utilisent des modèles de ressources distinctes pour organiser les photos; et communicationnelle : `Flickr` fournit à la fois des services `REST` et `WS*`, tandis que `Picasa` ne fournit des services `REST`. Par conséquent, une adaptation est nécessaire quand une application client qui orchestre les services `Picasa` doit évoluer pour orchestrer des services fournis par `Flickr`, ou inversement; ou même quand elle doit évoluer d'une interface `Restful` à une interface `WS*`, dans le cas de `Flickr`. Une intégration est nécessaire lorsque l'application cliente doit orchestrer la fois des services `Picasa` et `Flickr`. Une coordination est nécessaire lorsque deux scripts, éventuellement écrits dans des langages distincts, doivent coopérer pour orchestrer des services fournis par un seul système.

Typiquement, pour résoudre les problèmes d'interopérabilité, les développeurs mettent en œuvre des patrons de conception, ou une de leurs variations. Le problème de l'adaptation peut être résolu avec le patron d'adaptateur. Le problème de l'intégration peut être résolu avec le patron Façade. Enfin, le problème de coordination peut être résolu par le patron Médiateur. Cependant, les trois solutions s'appuient sur une architecture avec un cadre commun entre les langues et les interfaces d'orchestration.

Notre proposition est une architecture pivot où les scripts écrits dans différentes langues sont traduits dans une langue pivot puis exécutés sur différentes interfaces et modèles de données. Comme `Criojo` est un langue (probablement) universelle pour interfacer des ressources et orchestrer des services, nous l'utilisons comme la langue au centre de l'architecture de pivot. Dans cette partie nous expliquons l'architecture pivot et nous indiquons les caractéristiques requises dans une langue pivot. Ensuite, nous décrivons les trois problèmes d'interopérabilité et la solution proposée basée sur l'architecture de pivot avec `Criojo` comme langue pivot.

A.5.1 L'architecture Pivot

Pour être efficace, l'architecture pivot repose sur trois hypothèses pour la langue de pivot: (i) que n'importe quel langage d'orchestration peut être compilé dans cette langue, (ii) que le langage pivot peut interagir avec des interfaces différentes, et (iii) que les patrons de conception utilisés pour résoudre les problèmes d'interopérabilité peuvent être encodés dans cette langue. Nous tournons ces hypothèses en trois exigences relatives à la langue pivot.

Universalité pour la compilation Pour compiler des scripts écrits dans différentes langues d'orchestration dans la langue de pivot, nous avons besoin d'une langue multi-paradigme, qui supporte la compilation des langages impératifs comme `Java`, langages fonctionnels comme `XQuery`, des langues concurrents comme `BPEL` et des langues logiques comme `SQL` ou `YQL`.

Universalité pour interfacer Les interfaces de services diffèrent non seulement du point de vue fonctionnel, mais aussi communicationnel, comme l'illustre le cas de Flickr et Picasa. Un langage de représentation de ressources universel est donc nécessaire, ainsi que une couche logicielle intermédiaire capable de s'interfacer avec différentes sources.

Expressivité Le langage pivot doit être suffisamment expressif pour permettre l'encodage des différents patrons de conception. Nous considérons que cette dernière exigence découle de la première, car un certain niveau d'expressivité est nécessaire pour permettre la compilation des paradigmes de programmation.

A.5.2 Implémentation avec `Criojo`

Cette section montre comment `Criojo` est utilisé comme langage pivot dans l'architecture pivot, et comment il peut être utilisé pour résoudre les problèmes d'interopérabilité. Nous présentons trois scénarios différents exposant l'adaptation, l'intégration et les problèmes de coordination, respectivement, et proposons une solution basée sur l'architecture pivot. Notre proposition est basée sur des patrons de conception. Concrètement, nous mettons en oeuvre le patron *Adaptateur* pour résoudre le problème d'adaptation, le patron *Façade* pour résoudre le problème d'intégration et le patron *Médiateur* pour résoudre le problème de coordination.

La mise en oeuvre du patron Façade étant similaire à celui du patron Adaptateur nous présentons dans ce resume seulement celui de l'adaptation. Les examples complets se trouvent en la version complete en anglais.

A.5.2.1 Adaptation : le Patron Adaptateur

Il est possible qu'une application client modifie le fournisseur d'un service, soit parce que l'utilisateur aime la nouveauté ou le service actuel n'est plus disponibles, etc. Dans tous les cas, une adaptation est requise. Une solution pour ce problème est l'implémentation du patron adaptateur, en transformant l'interface d'un service dans une autre interface, celle attendue par le client. Les Figures A.3, A.4 et A.5 montrent l'example d'une application client connectée à `Flickr` qui doit être adaptée pour se connecter à `Picasa`.

Initialement le client utilise le langage de requête `YQL` pour requêter sur les photos.

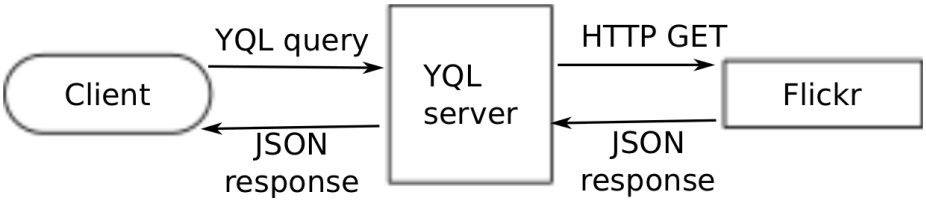

Figure A.3: `YQL` Execution de la Requête

Nous proposons de remplacer le serveur `YQL` avec un composant `Criojo`.

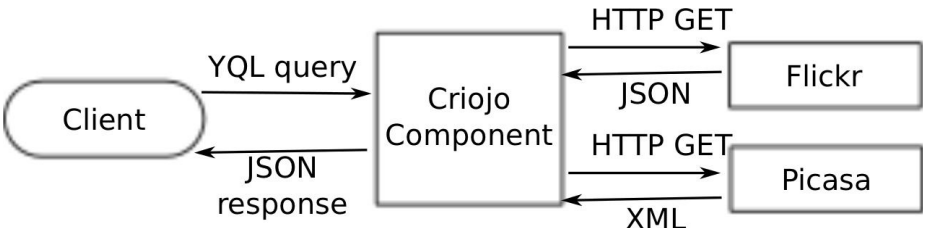

Figure A.4: Adaptation: Le Composant `Criojo` Remplace le Service `YQL`

Le composant `Criojo` implemente le patron adaptateur, ce qui lui permet de communiquer avec `Flickr` ou `Picasa`.

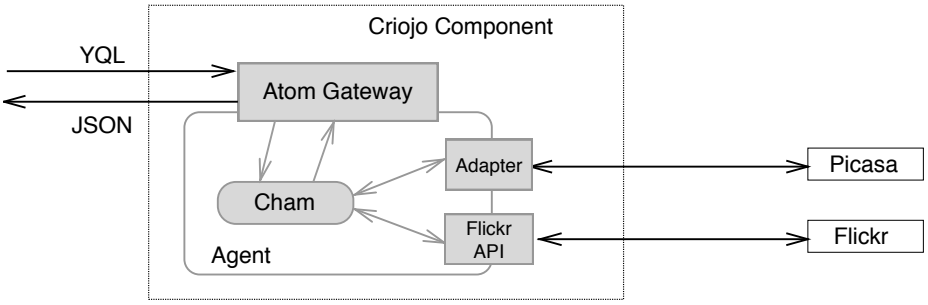

Figure A.5: Détail du Composant `Criojo`

A.5.2.2 Coordination : le Patron Mediateur

Dans ce dernier scenario nous montrons comment le patron mediateur permet de combiner deux scripts écrits dans des langages différents : l'un écrit en `YQL` et l'autre écrit en un langage fonctionnel comme `Haskell`. En effet, en utilisant `Criojo` comme langue pivot nous pouvons compiler les scripts dans `Criojo` ou les intégrer via des adaptateurs. Puis, avec l'aide d'un composant médiateur, nous pouvons coordonner les programmes qui en résultent, profitant ainsi des fonctionnalités offertes par chaque langage. Dans cette exemple nous avons une requête `YQL` que retourne les noms d'utilisateurs apparaissant dans certaines photos. La requête renvoie un ensemble de paires `(usename, photo_id)`, triées par nom d'utilisateur. Pour obtenir un résultat groupé par `username` nous avons recours à un médiateur qui coordonne la requête existante avec un autre script écrit dans un langage où ce genre d'operations sont exprimées de façon plus naturelle. La Figure A.6 montre le schema du patron mediateur, où nous utilisons un adaptateur `REST` pour communiquer avec le serveur `YQL` et un autre adaptateur pour communiquer avec l'environnement d'execution `Haskell`.

Le médiateur prend les critères de sélection `YQL` et une fois intégré dans la requête complète, les transmet au serveur `YQL` via l'adaptateur `REST` qui transforme le message en une requête `HTTP GET`, puis transforme le document `Json` résultant en un atome.

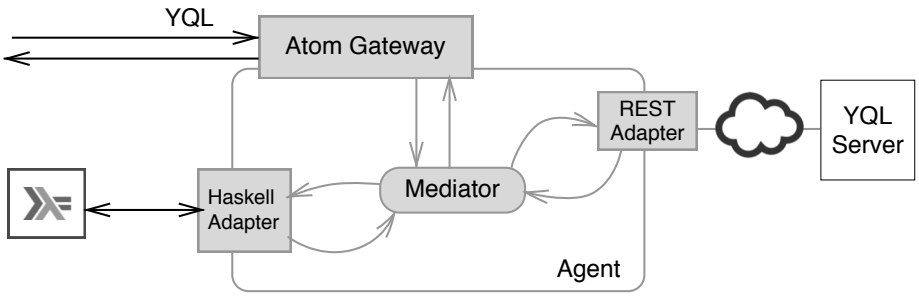

Figure A.6: Detail of the `Criojo` Component

A.6 Les Travaux à Venir

Si la mise en oeuvre actuelle de `Criojo` est avérée être un outil utile pour l'orchestration de services en étendant la Heta-calculus sans modifier sa sémantique, il reste un prototype et beaucoup de travail doit encore être fait. Nous présentons dans cette section les perspectives de ce travail :

A.6.1 La causalité et la communication synchrone

Certains algorithmes nécessitent de préserver l'ordre des événements de l'émission à la réception. Cela s'appelle un calcul causalement ordonné [CBMT96]. Donc, une extension souhaitable du langage serait l'introduction de canaux plus riches, capables de supporter des messages ordonnés par causalité, soit en étendant la communication asynchrone ou en utilisant directement les protocoles synchrones de manière native.

A.6.2 Parallélisation

Avec l'émergence des processeurs multicœurs, la programmation distribuée prend une autre dimension : les calculs nécessitant plusieurs ordinateurs indépendants peuvent se faire dans un seul processeur. La programmation séquentielle traditionnelle est moins adaptée à cette technologie contrairement aux langages orientés au parallélisme comme `Criojo`. Cependant, l'implémentation actuelle de `Criojo` ne recourt pas au parallélisme. Une solution possible vers la parallélisation serait d'ajouter la capacité de déployer les différentes chams dans le même ordinateur, chacun dans un noyau dédié. Une autre solution serait de permettre l'exécution parallèle de règles à l'intérieur d'un groupe. Le problème peut alors être déclaré comme un problème d'ordonnancement avec conflits [EHKR09].

A.6.3 Exécution de règles optimales

L'implémentation actuelle des règles de réaction reflète étroitement la sémantique exprimée dans la théorie de l'Heta-calculus, et garantit l'évaluation correcte des règles. Néanmoins, les automates ne sont pas la solution la plus efficace. Dans certain cas, cela peut conduire à un produit cartésien sur le total d'atomes. Il y a un équilibre à trouver entre la puissance expressive de gardes, requis pour l'exhaustivité du calcul ; et la complexité de leur évaluation, qui doit être non seulement raisonnable, mais également prévisible de façon intuitive.

A.6.4 Intégration

Selon ce que nous voulons faire, il y a deux façons pour l'intégration de `Criojo` avec d'autres langues, et des extensions sont nécessaires pour chacune de ces méthodes. Tout d'abord, les adaptateurs permettent de réutiliser des composants existants écrits dans différents langages et de tirer parti des caractéristiques d'une langue donnée. La deuxième façon est d'embarquer `Criojo`, comme le montre le prototype actuel.

Bibliography

[ACC+10] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys 2010*, pages 223–236, 2010. (Cited on page 48.)

[ACG86] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(8):26–34, 1986. (Cited on page 12.)

[ADG+12] Diana Allam, Rémi Douence, Hervé Grall, Jean-Claude Royer, and Mario Südholt. A message-passing model for service oriented computing. In Karl-Heinz Krempels and José Cordeiro, editors, *WEBIST*, pages 136–142. SciTePress, 2012. (Cited on page 21.)

[All14] Diana Allam. *Loose Coupling and Substitution Principle in Object-Oriented Frameworks for Web Services*. PhD thesis, Mines de Nantes, 2014. (Cited on page 21.)

[AO93] Gregory R. Andrews and Ronald A. Olsson. *The SR Programming Language: Concurrency in Practice*. Benjamin/Cummings, 1993. (Cited on page 16.)

[AS85] Bowen Alpern and Fred Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985. (Cited on page 8.)

[Bae05] J. C. M. Baeten. A brief history of process algebra. *Theor. Comput. Sci.*, 335(2-3):131–146, May 2005. (Cited on page 55.)

[Bal90] Henri Bal. *Programming Distributed Systems*. Prentice Hall, 1990. (Cited on page 9.)

[BB90] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming lan-*

guages, POPL '90, pages 81–94, New York, NY, USA, 1990. ACM. (Cited on pages 58, 70 and 73.)

[BBC+06] M. Boreale, R. Bruni, L. Caires, R. De Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. Vasconcelos, and G. Zavattaro. Scc: a service centered calculus. In *Proceedings of the Third international conference on Web Services and Formal Methods*, WS-FM'06, pages 38–57, Berlin, Heidelberg, 2006. Springer-Verlag. (Cited on pages 55 and 56.)

[BBG+06] Steffen Becker, Antonio Brogi, Ian Gorton, Sven Overhage, Alexander Romanovsky, and Massimo Tivoli. Towards an engineering approach to component adaptation. In *Proceedings of the 2004 International Conference on Architecting Systems with Trustworthy Components*, pages 193–215. Springer-Verlag, 2006. (Cited on pages 23 and 25.)

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-Centric General-Purpose Language. *SIGPLAN Notices*, 38(9):51–63, 2003. (Cited on page 29.)

[BCNS13] Véronique Benzaken, Giuseppe Castagna, Kim Nguy, and Jérôme Siméon. Static and Dynamic Semantics of NoSQL Languages. In *POPL '13: Proceedings of the 40h annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2013. (Cited on pages 34, 38, 45, 67 and 162.)

[BCP07] Antonio Brogi, Carlos Canal, and Ernesto Pimentel. Behavioural types for service integration: Achievements and challenges. *Electronic Notes in Theoretical Computer Science*, 180(2):41–54, 2007. (Cited on page 23.)

[BEZ92] Geoff Barrett and Steven Ericsson-Zenith. *occam 3 reference manual*. INMOS, 1992. (Cited on page 13.)

[BFR06] J.-P. Banâtre, P. Fradet, and Y. Radenac. Generalised multisets for chemical programming. *Mathematical. Structures in Computer Science*, 16(4):557–580, 2006. (Cited on page 59.)

[BGB+11] Kevin S Beyer, Rainer Gemulla, Andrey Balmin, Eugene J Shekita, Carl-christian Kanne, and Fatma Ozcan. Jaql : A Scripting Language for Large Scale Semistructured Data Analysis. In *PVLDB*, pages 1272–1283, 2011. (Cited on page 37.)

[BGR11] Yerom-David Bromberg, Paul Grace, and Laurent Réveillère. Starlink: Runtime interoperability between heterogeneous middleware protocols. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems*, ICDCS '11, pages 446–455, Washington, DC, USA, 2011. IEEE Computer Society. (Cited on page 23.)

[BGRB11] Yérom-David Bromberg, Paul Grace, Laurent Réveillère, and Gordon S. Blair. Bridging the interoperability gap: Overcoming combined application and middleware heterogeneity. In *Middleware'2011*, volume 7049 of *Lecture Notes in Computer Science*, pages 390–409, 2011. (Cited on page 23.)

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. (Cited on page 11.)

[BK09] Michael Benedikt and Christoph Koch. Xpath leashed. *ACM Comput. Surv.*, 41(1):3:1–3:54, January 2009. (Cited on page 35.)

[BM93] Jean-Pierre Banâtre and Daniel Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1), 1993. (Cited on page 59.)

[BOT13] Marin Bertier, Marko Obrovac, and Cédric Tedeschi. Adaptive atomic capture of multiple molecules. *Journal of Parallel and Distributed Computing*, 73(9):1251–1266, 2013. (Cited on page 152.)

[BP09] Jean-Pierre Banâtre and Thierry Priol. Chemical programming of future service-oriented architectures. *Journal of Software*, 4(7), 2009. (Cited on page xiv.)

[BPZ11] Lina Bentakouk, Pascal Poizat, and Fatiha Zaïdi. Checking the behavioral conformance of web services with symbolic testing and an smt solver. In *TAP*, pages 33–50, 2011. (Cited on page 53.)

[Bre12] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012. (Cited on page 39.)

[BRF] Hariolf Betz, Frank Raiser, and Thom FrÜhwirth. A complete and terminating execution model for constraint handling rules. *Theory Pract. Log. Program.*, 10(4-6):597–610. (Cited on page 60.)

[Bun97] Peter Buneman. Semistructured data. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '97, pages 117–121, New York, NY, USA, 1997. ACM. (Cited on pages 33 and 35.)

[CBMT96] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous, and causally ordered communication. *Distrib. Comput.*, 9(4):173–191, 1996. (Cited on pages 17, 150 and 177.)

[CBPS10] Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010. (Cited on page 9.)

[CDNP+11] Luís Caires, Rocco De Nicola, Rosario Pugliese, VascoT. Vasconcelos, and Gianluigi Zavattaro. Core calculi for service-oriented computing. In Martin Wirsing and Matthias Hölzl, editors, *Rigorous Software Engineering for Service-Oriented Systems*, volume 6582 of *Lecture Notes in Computer Science*, pages 153–188. Springer, 2011. (Cited on page 56.)

[CGR88] R.F. Cmelik, N.H. Gehani, and W.D. Roome. Fault tolerant concurrent c: a tool for writing fault tolerant distributed programs. In *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on*, pages 56 –61, jun 1988. (Cited on page 16.)

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166, 1989. (Cited on page 31.)

[CHY12] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2), 2012. (Cited on page 52.)

[CMP06] Carlos Canal, Juan Manuel Murillo, and Pascal Poizat. Software adaptation. *L'OBJET*, 12(1):9–31, 2006. (Cited on page 23.)

[Cod70] Edgar Codd. A relational model of data for large shared data banks. *Communations of the ACM*, 13(6):377–387, 1970. (Cited on page 30.)

[Cod71] Edgar Codd. Further normalization of the data base relational model. *IBM Research Report, San Jose, California*, RJ909, 1971. (Cited on page 30.)

[Cur04] Edward Curry. *Middleware for Communication*, chapter Message-Oriented Middleware, pages 1–28. Number 1. John Wiley and Sons, Ltd, 2004. (Cited on page 129.)

[DCd10] Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. Sessions and session types: An overview. In Cosimo Laneve and Jianwen Su, editors, *WS-FM*, volume 6194 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2010. (Cited on page 23.)

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150. USENIX Association, 2004. (Cited on page 39.)

[DKSD07] Leslie De Koninck, Tom Schrijvers, and Bart Demoen. User-definable rule priorities for chr. In *Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, PPDP '07, pages 25–36, New York, NY, USA, 2007. ACM. (Cited on page 60.)

[EBJ11] Jeff Epstein, Andrew P. Black, and Simon L. Peyton Jones. Towards haskell in the cloud. In *Haskell Symposium 2011*, pages 118–129. ACM, 2011. (Cited on page 29.)

[EGST07] Justin R. Erenkrantz, Michael Gorlick, Girish Suryanarayana, and Richard N. Taylor. From representations to computations: the evolution of web architectures. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 255–264, New York, NY, USA, 2007. ACM. (Cited on page 145.)

[EHKR09] Guy Even, Magnús M. Halldórsson, Lotem Kaplan, and Dana Ron. Scheduling with conflicts: online and offline algorithms. *J. Scheduling*, 12(2):199–224, 2009. (Cited on pages 152 and 177.)

[Ein11] Oren Eini. The pain of implementing linq providers. *Queue*, 9(7):10:10–10:22, July 2011. (Cited on page 37.)

[Erl09] Thomas Erl. *SOA Design Patterns*. Prentice Hall, 2009. (Cited on page 24.)

[FBS04] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of interacting bpel web services. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 621–630, New York, NY, USA, 2004. ACM. (Cited on page 53.)

[FCP+12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, January 2012. (Cited on page 33.)

[Fer04] Andrea Ferrara. Web services: a process algebra approach. In *Proceedings of the 2nd international conference on Service oriented computing*, ICSOC '04, pages 242–251, New York, NY, USA, 2004. ACM. (Cited on page 55.)

[FG96] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 372–385, New York, NY, USA, 1996. ACM. (Cited on pages 59 and 72.)

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000. AAI9980887. (Cited on pages 19 and 20.)

[FM77] Charles Forgy and John P. McDermott. Ops, a domain-independent production system language. In R. Reddy, editor, *IJCAI*, pages 933–939. William Kaufmann, 1977. (Cited on page 59.)

[FM98] Fabrice Le Fessant and Luc Maranget. Compiling join-patterns. *Electronic Notes in Theoretical Computer Science*, 16(3):205 – 224, 1998. (Cited on pages 120 and 170.)

[Frü08] Thom W. Frühwirth. Welcome to constraint handling rules. In *Constraint Handling Rules*, pages 1–15. 2008. (Cited on page 59.)

[G̈99] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, March 1999. (Cited on pages 7, 9 and 64.)

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96–107, 1992. (Cited on pages 22 and 52.)

[Geh84] Narain H. Gehani. Broadcasting sequential processes (bsp). *Software Engineering, IEEE Transactions on*, SE-10(4):343 –351, july 1984. (Cited on page 11.)

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, illustrated edition edition, 1994. (Cited on page 135.)

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1 – 101, 1987. (Cited on page 60.)

[GL12] Seth Gilbert and Nancy A. Lynch. Perspectives on the CAP Theorem. *IEEE Computer*, 45(2):30–36, 2012. (Cited on pages 9, 38 and 39.)

[GLG+06] Claudio Guidi, Roberto Lucchi, Roberto Gorrieri, Nadia Busi, and Gianluigi Zavattaro. SOCK: A calculus for service oriented computing. In Asit Dan and Winfried Lamersdorf, editors, *ICSOC*, volume 4294 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2006. (Cited on page 56.)

[GLP04] Deepak Garg, Akash Lal, and Sanjiva Prasad. Effective chemistry for synchrony and asynchrony. In Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell, editors, *IFIP TCS*, pages 479–492. Kluwer, 2004. (Cited on page 58.)

[GM02] Harald Ganzinger and David McAllester. Logical algorithms. In Peter Stuckey, editor, *Logic Programming*, volume 2401 of *Lecture Notes in Computer Science*, pages 31–42. Springer Berlin / Heidelberg, 2002. (Cited on pages 31 and 67.)

[Gog94] Formal semantics of sql. In Martin Gogolla, editor, *An Extended Entity-Relationship Model*, volume 767 of *Lecture Notes in Computer Science*, pages 99–120. Springer Berlin / Heidelberg, 1994. (Cited on page 32.)

[GP09] Dimitrios Georgakopoulos and Michael Papazoglou, editors. *Service-Oriented Computing*. MIT Press, 2009. (Cited on pages ix, 1 and 25.)

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983. (Cited on page 14.)

[Gra81] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *VLDB*, pages 144–154. IEEE Computer Society, 1981. (Cited on page 32.)

[GZ01] Sergio Greco and Carlo Zaniolo. Greedy algorithms in datalog. *Theory Pract. Log. Program.*, 1(4):381–407, July 2001. (Cited on page 31.)

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. (Cited on pages 11 and 29.)

[Her90] Maurice Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, 1990. (Cited on page 11.)

[HGL11] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1213–1216, New York, NY, USA, 2011. ACM. (Cited on page 31.)

[HHH10] Tim Hallwyl, Fritz Henglein, and Thomas T. Hildebrandt. A standard-driven implementaion of ws-bpel 2.0. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC*, pages 2472–2476. ACM, 2010. (Cited on page xv.)

[HHPJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM. (Cited on page 28.)

[HO06] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In DavidE. Lightfoot and Clemens Szyperski, editors, *Modular Programming Languages*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer Berlin Heidelberg, 2006. (Cited on page 124.)

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice/Hall International, Englewood Cliffs, N.J, 1985. (Cited on page 55.)

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley, 2003. (Cited on page 53.)

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983. (Cited on page 32.)

[HS05] M.N. Huhns and M.P. Singh. Service-oriented computing: key concepts and principles. *Internet Computing, IEEE*, 9(1):75 – 81, jan-feb 2005. (Cited on page 19.)

[HSS05] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming bpel to petri nets. In Wil van der Aalst, Boualem Benatallah, Fabio Casati, and Francisco Curbera, editors, *Business Process Management*, volume 3649 of *Lecture Notes in Computer Science*, pages 220–235. Springer Berlin / Heidelberg, 2005. (Cited on page 54.)

[HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. (Cited on page 24.)

[ISO89] ISO/IEC. *Information Processing Systems – Open Systems Interconnection: LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, 1989. (Cited on page 55.)

[Jac95] Bart Jacobs. Objects and Classes, Co-Algebraically. In *Object Orientation with Parallelism and Persistence*, pages 83–103. 1995. (Cited on page 29.)

[JS08] Simon L. Peyton Jones and Satnam Singh. A tutorial on parallel and concurrent programming in haskell. In *Advanced Functional Programming Summer School*, volume

5832 of *Lecture Notes in Computer Science*, pages 267–305. Springer, 2008. (Cited on pages 13 and 30.)

[KCM06] David Kitchin, William R. Cook, and Jayadev Misra. A language for task orchestration and its semantic properties. In *Proceedings of the 17th international conference on Concurrency Theory*, CONCUR'06, pages 477–491, Berlin, Heidelberg, 2006. Springer-Verlag. (Cited on page 56.)

[Klu82] Anthony C. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982. (Cited on page 32.)

[KvB04] Mariya Koshkina and Franck van Breugel. Modelling and verifying web service orchestration by means of the concurrency workbench. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, September 2004. (Cited on page 57.)

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, 1977. (Cited on page 8.)

[LGL10] Mayleen Lacouture, Hervé Grall, and Thomas Ledoux. CREOLE: a Universal Language for Creating, Requesting,Updating and Deleting Resources. In *Proceedings of the 9th International Workshop on the Foundations of Coordination Languages and Software Architectures-A (FOCLASA2010)*, 09 2010. (Cited on pages 25 and 134.)

[Lis88] Barbara Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, March 1988. (Cited on page 16.)

[Liu00] Yanhong A. Liu. Efficiency by incrementalization: An introduction. *Higher-Order and Symbolic Computation*, 13(4):289–313, 2000. (Cited on pages 118 and 169.)

[LL90] Leslie Lamport and Nancy Lynch. Handbook of theoretical computer science (vol. b). chapter Distributed computing: models and methods, pages 1157–1199. MIT Press, Cambridge, MA, USA, 1990. (Cited on pages 4 and 7.)

[LM07a] Ralf Lämmel and Erik Meijer. Revealing the X/O Impedance Mismatch: Changing Lead into Gold. In *SSDGP'06*, volume 4719 of *LNCS*, pages 285–367, 2007. (Cited on pages 29 and 47.)

[LM07b] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for ws-bpel. *Journal of Logic and Algebraic Programming*, 70(1):96 – 118, 2007. ¡ce:title¿Web Services and Formal Methods¡/ce:title¿. (Cited on page 57.)

[LMSW06] Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. Analyzing interacting bpel processes. In Schahram Dustdar, José Fiadeiro, and Amit Sheth, editors, *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 17–32. Springer Berlin / Heidelberg, 2006. (Cited on page 54.)

[LPT07] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A calculus for orchestration of web services. In *Proceedings of the 16th European conference on Programming*, ESOP'07, pages 33–47, Berlin, Heidelberg, 2007. Springer-Verlag. (Cited on pages 56 and 57.)

[LPT12] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. Using formal methods to develop ws-bpel applications. *Science of Computer Programming*, 77(3):189 – 213, 2012. (Cited on page 57.)

[LR06] James R. Larus and Ravi Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2006. (Cited on page 12.)

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ : Reconciling objects , relations and XML in the . NET framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006. (Cited on page 37.)

[MC94] Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26:87–119, 1994. (Cited on page 22.)

[McL02] Brett McLaughlin. *Java and XML Data Binding*. O'Reilly, 2002. (Cited on page 27.)

[Mei11] Erik Meijer. The World According to LINQ Big data is about more than size , and LINQ is more than up to the task . *Queue*, 9(8):60:60—-60:72, 2011. (Cited on page 37.)

[MF95] Friedemann Mattern and Stefan Fünfrocken. A non-blocking lightweight implementation of causal order message delivery. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, volume 938 of *LNCS*, pages 197–213. SpringVerl, 1995. (Cited on pages 6 and 62.)

[Mon13] Fabrizio Montesi. *Choreographic Programming*. PhD thesis, Faculty of the IT University of Copenhagen, 2013. (Cited on page 52.)

[MPW92a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992. (Cited on pages 4, 5 and 55.)

[MPW92b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, September 1992. (Cited on page 57.)

[MPW92c] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Information and Computation*, 100(1):41–77, 1992. (Cited on pages 4 and 55.)

[MSSW08] Peter Massuthe, Alexander Serebrenik, Natalia Sidorova, and Karsten Wolf. Can i find a partner? undecidability of partner existence for open nets. *Information Processing Letters*, 108(6):374 – 378, 2008. (Cited on page 54.)

[NPS91] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of sql queries. *ACM Trans. Database Syst.*, 16(3):513–534, September 1991. (Cited on page 32.)

[ÖV11] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Third Edition*. Springer, 2011. (Cited on page 39.)

[OVvdA+07] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Sci. Comput. Program.*, 67(2-3):162–198, July 2007. (Cited on page 54.)

[Pet62] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962. (Cited on page 54.)

[Pet77] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977. (Cited on page 54.)

[Pie97] Benjamin C. Pierce. Foundational calculi for programming languages. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, pages 2190–2207. CRC Press, 1997. (Cited on page 55.)

[Pro14] Betty Project. State of the Art Report, Project Betty (Behavioural Types for Reliable Large-Scale Software Systems). Project report, 2014. (Cited on page 23.)

[PZL08] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *Proceedings of the 17th International World Wide Web Conference (WWW 2008)*, pages 805–814, 2008. (Cited on page 19.)

[PZQ+06] Geguang Pu, Huibiao Zhu, Zongyan Qiu, Shuling Wang, Xiangpeng Zhao, and Jifeng He. Theoretical foundations of scope-based compensable flow language for web service. In *Proceedings of the 8th IFIP WG 6.1 international conference on Formal Methods for Open Object-Based Distributed Systems*, FMOODS'06, pages 251–266, Berlin, Heidelberg, 2006. Springer-Verlag. (Cited on page 57.)

[PZWQ06] Geguang Pu, Xiangpeng Zhao, Shuling Wang, and Zongyan Qiu. Towards the semantics and verification of bpel4ws. *Electronic Notes in Theoretical Computer Science*, 151(2):33 – 52, 2006. (Cited on page 57.)

[Ros10] Jonathan Rosenberg. Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. 2010. (Cited on page 18.)

[RWHM03] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. Stun - simple traversal of user datagram protocol (udp) through network address translators (nats), 2003. (Cited on page 18.)

[SB12] Manuel Serrano and Gérard Berry. Multitier programming in hop. *Commununications of the ACM*, 55(8):53–59, 2012. (Cited on page 43.)

[SBS04] Gwen Salaün, Lucas Bordeaux, and Marco Schaerf. Describing and reasoning on web services using process algebra. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '04, pages 43–, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on page 55.)

[SF12] Pramod J. Sadalage and Martin Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison Wesley, 2012. (Cited on page 33.)

[Sha86] Ehud Shapiro. Concurrent prolog: A progress report. *IEEE JOURNALS AND MAGAZINES*, 19, 1986. (Cited on page 12.)

[SP08] Robert Simmons and Frank Pfenning. Linear logical algorithms. In Luca Aceto, Ivan Damgård, Leslie Goldberg, Magnús Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, volume 5126 of *Lecture Notes in Computer Science*, pages 336–347. Springer Berlin / Heidelberg, 2008. (Cited on page 60.)

[SVWSD07] Jon Sneyers, Peter Van Weert, Tom Schrijvers, and Bart Demoen. Aggregates in constraint handling rules. In *Proceedings of the 23rd international conference on Logic programming*, ICLP'07, pages 446–448, Berlin, Heidelberg, 2007. Springer-Verlag. (Cited on page 60.)

[SW01] Davide Sangiorgi and David Walker. *PI-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2001. (Cited on page 104.)

[tBBG07] M. ter Beek, A. Bucchiarone, and S. Gnesi. Formal methods for service composition. *Annals of Mathematics, Computing & Teleinformatics*, 5(1):1 – 10, 2007. (Cited on page 53.)

[TS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. (Cited on page 3.)

[Vin97] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997. (Cited on page 27.)

[Vog03] Werner Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6):59–66, November 2003. (Cited on page 19.)

[WDG+07] Martin Wirsing, Rocco De Nicola, Stephen Gilmore, Matthias Hölzl, Roberto Lucchi, Mirco Tribastone, and Gianluigi Zavattaro. SENSORIA Process Calculi for Service-Oriented Computing. In Don Sanella and Ugo Montanari, editors, *Trustworthy Global Computing, Second Symposium, TGC 2006, Revised Selected Papers.*, volume 4661 of *Lecture Notes in Computer Science*, pages 30–50, Italy, 2007. Springer. (Cited on pages 55 and 56.)

[Wol09] Karsten Wolf. Does my service have partners? In Kurt Jensen and WilM.P. van der Aalst, editors, *Transactions on Petri Nets and Other Models of Concurrency II*, volume 5460 of *Lecture Notes in Computer Science*, pages 152–171. Springer Berlin Heidelberg, 2009. (Cited on page 54.)

[WP13] Chen Wang and J. Pazat. A chemistry-inspired middleware for self-adaptive service orchestration and choreography. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 426–433, May 2013. (Cited on page 24.)

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. *Computing Systems*, 9(4):265–290, 1996. (Cited on pages 10 and 27.)

[WSSD06] Peter Van Weert, Jon Sneyers, Tom Schrijvers, and Bart Demoen. Extending CHR with negation as absence. In *Proceedings of the Third Workshop on Constraint Handling Rules*, pages 125–139, 2006. (Cited on page 60.)

[WWWK96] Jim Waldo, Geoff Wyant, Ann Wollrath, and Samuel C. Kendall. A note on distributed computing. In Jan Vitek and Christian F. Tschudin, editors, *Mobile Object Systems*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer, 1996. (Cited on page 47.)

[WZ00] Haixun Wang and Carlo Zaniolo. User-defined aggregates in database languages. In *7th International Workshop on Database Programming Languages, DBPL'99*, volume 1949 of *LNCS*, pages 43–60. SpringVerl, 2000. (Cited on page 31.)

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997. (Cited on pages 22 and 23.)

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

MINES Nantes
Institut Mines-Télécom

Thèse de Doctorat

Mayleen LACOUTURE

Un langage de Programmation Chimique pour l'Orchestration des Services
Application aux problèmes d'interopérabilité

A Chemical Programming Language for Orchestrating Services
Application to Interoperability Problems

Résumé
Avec l'émergence du "Cloud-computing" et des applications mobiles, il est possible de trouver un service web répondant à presque tout besoin. De plus, les développeurs peuvent créer des applications complexes en combinant différents services indépendants, dont l'agencement et l'exécution peuvent être automatisés avec l'aide de langages d'orchestration. Cependant, la diversité des technologies et le manque de standardisation peuvent entraver la collaboration entre services. Un exemple de cette limitation est le cas de la gestion des photos avec des services tels que Flickr et Picasa, qui diffèrent non seulement sur la façon dont les photos sont organisées mais aussi sur les services qu'ils fournissent. L'hétérogénéité des ces deux services conduit à des problèmes d'interopérabilité, à savoir dans l'adaptation, l'intégration et la coordination. Nous proposons un framework pour aider à la résolution de ces problèmes, sous la forme d'une architecture qui intègre différents langages d'orchestration avec des fournisseurs de services hétérogènes autour d'un langage pivot. Comme langage pivot, nous proposons le langage d'orchestration Criojo qui implémente et étend le Heta-calcul, un calcul original associé à une machine chimique abstraite dédié à l'orchestration de services. En adoptant cette approche l'interopérabilité entre les services et les langages d'orchestration sera améliorée, facilitant ainsi le développement des services composites. Le haut niveau d'abstraction de Criojo pourrait permettre aux développeurs d'écrire des orchestrations très concises puisque les échanges de messages sont représentés d'une manière naturelle et intuitive.

Mots clés
Interopérabilité, Services, Programmation Chimique.

Abstract
With the emergence of cloud computing and mobile applications, it is possible to find a web service for almost everything. Moreover, developers can create complex applications by combining several independent services, whose arrangement and execution can be automated with the aid of orchestration languages. Nevertheless, the diversity of technologies and the lack of standardization can hinder the collaboration between services. An example of this limitation is the case of photo management with services such as Flickr and Picasa, which not only differ on the way photos are organized, but also in the services they provide. The heterogeneity of the two services leads to interoperability problems, namely adaptation, integration and coordination problems. We propose a framework for helping at the resolution of these issues, in the form of an architecture that integrates different orchestration languages with heterogeneous service providers around a pivot language. As a pivot language we propose an orchestration language based on the chemical programming paradigm. Concretely, this dissertation presents the language Criojo that implements and extends the Heta-calculus, an original calculus associated to a chemical abstract machine dedicated to service-oriented computing. The consequence of adopting this approach would be an improvement in the interoperability of services and orchestration languages, thus easing the development of composite services. The high level of abstraction of Criojo could allow developers to write very concise orchestrations since message exchanges are represented in a natural and intuitive way.

Key Words
Interoperability, SOC, Chemical Programming.