博士論文

A Contract-based Programming Model
for Distributed Computing
（分散コンピューティングのための
契約に基づくプログラミングモデル）

学籍番号　37-087091
氏名　　マツイ, アウレリオ　アキラ　メーロ

指導教員　相田　仁教授

平成 26 年 2 月 14 日

Contents

List of Tables

List of Figures

List of Acronyms

ADPI Asynchronous Distributed π-Calculus

AOP Aspect-Oriented Programming

API Application Programming Interface

APT Annotation Processing Tool

AST Abstract Syntax Tree

CLR Common Language Runtime

DbC Design by Contract

DSL Domain-Specific Language

DSO Distributed Service Objects

DTO Data Transfer Object

EBNF Extended Backus-Naur Form

ECS Execution Control Subtree

EST Expected State Transition

FSM Finite State Machine

GGF Global Grid Forum

IDE Integrated Development Environment

IRC Internet Relay Chat

JAR Java Archive

JDK Java Development Kit

JML Java Modeling Language

JMS Java Message Service

JMX Java Management Extensions

JVM Java Virtual Machine

LTM Linear Transformation Matrix

MPI Message Passing Interface

OASIS Organization for the Advancement of Structured Information Standards

OGSA Open Grid Service Architecture

OGSI Open Grid Services Infrastructure

OOP Object-Oriented Programming

PID Process Identifier

PROMELA Process Meta Language

REPL Read-eval-print loop

SOA Service-Oriented Architecture

SPI Service Provider Interface

SSDL SOAP Service Description Language

SSH Secure Shell

UDDI Universal Description, Discovery, and Integration

UUID Universally Unique Identifier

WS-BA Web Service Business Activity

WSDL Web Service Description Language

WSRF Web Services Resource Framework

XML Extensible Markup Language

Preface

0.1 Acknowledgments

I would like to thank first and foremost Professor Hitoshi Aida for all his dedication during those years. I also want to thank all my laboratory mates: Sayuri Nakayama, Shingo Chiba, Shinya Kambori, Saeki Yoshiasu, Qu Shi, Mohamad Samir A. Eid, Amani, Tanaka, Kim Narae, and all other people I worked with in the Aida Laboratory and who I failed to mention here. I also would like to thank the many department students who I interacted with durin all these years of research and who provided me insights and ideas for my research. I would also like to thank professor Nakayama for his time.

Janaina de Souza Simões for all her support and patience. For all knowledge and everything I learned from you.

Maria Lúcia de Souza, Baasanjav Tserendagva, Anna Guseva, Natalia Poiata, Yoko Hisano, Atsushi Ota, Qu Shi, Yoshi, Carolina Ramirez, Felipe Hurtado, Amos, Valerio Salvucci, Thiago Minami, Mariane and Adriene Wada, Abrar.

The MEXT. The Japanese people in general, who funded this research.

Last, but not least, I would like to thank the Free Software community for all software and knowledge I gained from it. Especially, and as a personification of the Free Software community, I would like to thank Richard Stallman for his tireless work to promote the fundamental freedoms[1] related to software and culture in general.

To all the giants on whose shoulders I try to stand. To all those giants whose names can be found on the references, and to those who, although did not contributed directly enough to my research to appear in the references, were not less important during this journey.

0.2 Software used

This research was made possible thanks to the free availability of a miriad of pieces of free software. The least I can do in order to show my profound respect and admiration for the thousands of volunteers who made them possible is to write down this (admittedly) incomplete list of software I used, in no particular order: Gnu-Linux[2] (the kernel and all thousands of tools, including gcc[3], ls), Bash, Ubuntu[4], Emacs[5], Emacs org-mode[6], Vim[7], LaTeX, Java, Eclipse IDE for Java, Apache ActiveMQ[8], Apache Maven[9], Apache Log4J[10], Apache Commons[11], JUnit, Inkscape Vector Graphics Editor[12], GIMP Image Editor, Python (Jython[13]), Gnome[14], Evince[15], MySQL[16], among others.

[1] https://www.gnu.org/philosophy/free-sw.html
[2] http://www.linux.org/
[3] http://gcc.gnu.org/
[4] http://www.ubuntu.org
[5] http://www.gnu.org/s/emacs/
[6] http://orgmode.org
[7] http://www.vim.org/
[8] http://activemq.apache.org
[9] http://activemq.apache.org
[10] http://logging.apache.org/log4j/1.2/
[11] http://commons.apache.org/
[12] http://www.inkscape.org
[13] http://www.jython.org/
[14] https://www.gnome.org/
[15] https://wiki.gnome.org/Apps/Evince
[16] http://www.mysql.com/

Introduction

There seems to exist a consensus over a shift, along the last decades, from expensive computers and relatively cheap costs of programming to cheap computers and expensive developers.

This change was fueled by the reduction of the prices of hardware. More hardware performance at lower costs fueled the popularization of the programming tools that required less effort from developers, even at the expense of lowering the performance of programs. Usually optimization goes until a reasonable extend in therms of cost effectiveness, therefore faster and cheaper hardware is capable of indirectly change which are the challenges programmers will face.

Currently, each hour of a programmer salary is very expensive, comparing with the costs of hardware that will execute the software he produces. The result is that programmers are becoming more and more specialized on translating business logic into source codes of high level programming languages. In other words, the recent trend is to insulate programmers from concerns unrelated to business logic.

Enabling this business logic to be executed in many scenarios is becoming more and more a responsibility of programming language tools, pre-processors and compilers.

The same trend is also followed in the field of computation distribution, high performance computing, and parallelism. For example, OpenMP presents a programming model in which programmers are encouraged to leave thread management to the OpenMP libraries.

Clearly, a hand made management of threads may allow for a more flexible, and therefore potentially more fit to the problem, software. For example, OpenMP is quite adequate to problems in which parallelism may be obtained by splitting a task in similar pieces which are executed in parallel by many threads.

In general, those tools make development easier by introducing abstraction layers. The effect of those layers of abstractions would be the reduction of final performance, as optimizations are harder whenever the abstraction used is higher.

The same concern isolation for programming models observed in multi-core programming or local high performance computing can also be seen for distributed computing.

Now an important trend for next generation systems is to create systems that are the result of a composition of services that may be made available in disjoint administrative domains that may be separated by long distances.

In this research we aim to develop a development tool or style to address the problem to development of service-based systems while keeping the development simple (with a high level of abstraction). We investigate the possible uses of the methods enumerated above to those sorts of systems.

0.3 Goal

Our goal is to simply development of distributed applications based on services. A service, in our definition, is some sort of procedure exposed utilizing an abstract interface. We do not want to create a development environment adequate to a specific platform, but an idea that can adapted to a number of platforms. Here we use Java simply as an illustration and because most of the more complex components of Java are freely available and already mature enough for our needs.

0.4 Outline: Proposed components

In this section we outline the four components of our proposed environment.

0.4.1 DSL - programming abstraction

We propose a new Domain-Specific Language (DSL) that provides an abstraction around which programmers can easily write complex distributed systems based on services. By using the DSL programmers will at the same time have to care very little with the actual computation distribution details (such as network, message exchange mechanisms and protocols), and provide the means for an automated tool to check the adherence of generated programs with a previously defined contract. We will present what we mean by contract bellow.

0.4.2 Message domains - separation of concerns

A message domain is a partition that resembles the message exchange model of Java Message Service (JMS), but that can have its structure dynamically defined by some of it participants. This allows us to create at run time a new connection topology, which we will exploit to react to changes in the set of connections between software agents. This need will become more clear when we introduce the process calculus we use to define contracts.

0.4.3 Distributed Middleware - support for prototyping

In order to check the theoretical model we propose, we have developed a prototype middleware. We explain the structure of this middleware in details on Chapter8.

0.4.4 Versioning model - support for service definition changes

Here we call a contract an abstract specification of how systems based on services should be composed and behave. A confusion can be made between this concept of contract and the concept in Design by Contract (DbC). While here a contract is a way to specify a behavior in terms of message exchange between processes (which for us is roughly equivalent to software agents), in DbC, a contract is a set of restrictions that aim at establishing rules used to validate data and this way avoid defensive programming, improving source code readability, among other benefits for the daily routine of programmers. In contrast, our approach for contract is one in which we provide a tool to check software correctness, which aims at allowing a number of pieces of software created by different developers to interact.

Our approach for formal verifications also aims at allowing client agents to partially implement contracts. This feature is important to allow for the same software client to interact with more than one server protocol version.

0.5 Assumptions and simplifications

Here we did not try to build a complete implementation of the stack necessary to the execution of an actual distributed service environment. Instead, we focused on the four aspects introduced in the previous section. Therefore, we had to make a number of assumptions regarding the rest of the infrastructure and to apply some simplifications.

0.5.1 Message Based

We assumed that all communication is message based. No direct connection between peers is implied[17]. A peer could send a message and no guarantee of message delivery is given a priory. Therefore we implemented a simple handshake when needed. In our model, such implementation details are not part of the Object-Oriented Programming (OOP) abstraction layer, but part of the specific driver for the message exchange mechanism.

There is a number of ways to implement message exchange. We could have designed a Web Service to implement message exchange but utilizing JMS had the fortunate advantage of providing a more scalable model. The tradeoff was interoperation, which is key to open systems such as Cloud Computing. Nevertheless, some message exchange services do not implement JMS as the sole interface to programming languages. Instead, JMS is only the interface for Java and other programming languages may utilize other Application Programming Interfaces (APIs) to communicate with the message exchange service.

The natural question that may rise is about the format of the messages. Can it be read by other programming languages? In the case of our research we tried to circumvent this problem by using JSON,

[17] Although, as we will see on Chapter8, it is also possible that a JMS implementation uses direct peer-to-peer connection for nodes to exchange messages.

but we were not successful [18]. In order to simplify development, we utilized serialized Java objects instead, as we will introduce briefly in the following section.

0.5.2 Protocol

Our implementation aims at developing a translation method that takes OOP description of procedures and makes it capable of being executed by the infrastructure without the need for the programmer to worry about specific protocol details. For example, if a certain protocol requires long messages to be spluit into more than one packet, the OOP abstraction will simply try to send a message, while the actual communication, implemented by the messaging driver, will be the exchange of several packets.

Protocol independence was our motivation to utilize JMS, since JMS does not force us to utilize any specific protocol, but can provide a pure OOP abstraction via the exchange of Object messages. Exchanging Java objects as message contents made implementation of the middleware and DSL simpler, but limited the utilization of the middleware for Java Virtual Machine (JVM) programming languages (such as Groovy and Scala) or scripting that can be executed on top of Java (such as Jython and JRuby).

[18] To see more details about why JSON could not be used, please refer to the Chapter8.

Part I

OOP and Distributed Computing

Chapter 1

Distributed OOP

Abstract

Recent studies in programming tools for grid computing have proposed the use of Aspect-Oriented Programming (AOP) as a way to separate the business logic concerns from the grid processing distribution concerns. We found that limitations in the design of current AOP languages restricted the capacity of the programming tools to manipulate the programmer's source code. Therefore, this paper proposes an extension of this concept in which both AOP and OOP source codes are generated at compiling time to increase the lexicon available for programmers to express advanced computation distribution. We also tested our proposal using a distributed Apriori algorithm.

Introduction

With the popularization of computational grids execute scientific applications, scientists who were not used to distributed computing programming are now having to deal with the complexities of the grids. Recent developments, as in [52] and [53], propose a development model for grid systems that applies source code transformation to make the grid complexities transparent to programmers who are not specialists in distributed programming.

Traditionally, source codes of distributed applications created with the client-server paradigm will normally have calls to grid functions side by side with the actual implementation of the main algorithms.

In the new approach, on the other hand, programmers are supposed to create applications that are started locally and that have some of their parts executed by the CPUs available in the grid. In this approach, AOP is used to create a tool that receives the source code written by the programmer and performs transformations in it to create a distributed application and deploy it at runtime to a set of remote servers on demand.

The use of AOP can separate these concerns in several disjoint entities that can be managed separately, increasing the manageability of the system as a whole and allowing for source code to be adapted to situations they were not designed for (obliviousness as defined in [27]).

A complex distributed infrastructure like a grid requires the calling system to deal with concerns such as resource or service location and authentication. In grid computing, AOP can be used to automate the repetitive coding necessary to create communication with the grid, as proposed in [38].

The mechanisms created so far are limited to the creation of bag-of-tasks [28], a class of distributed programs in which the tasks do not have any relationship between each other during their execution.

In this paper we propose an extension of the AOP programming tool for grid computing in which we apply a dynamic aspect and class generator to extend an object-oriented programming language, increasing the vocabulary that a programmer can use to express his computation distribution strategy.

To illustrate our method, we implemented two programming constructs: (1) a mechanism for remote procedures to exchange data with the process initiator node during the execution of these remote procedures, and (2) an amelioration in the mechanism to instantiation of objects in remote nodes. The effectiveness of our method was verified using a distributed data mining algorithm.

The rest of this chapter is organized as follows. The following section is aimed to present the related works. Next, in section 3, we briefly discuss the limitations in previous proposals. The following section presents our proposed amelioration. Section 5 presents a comparison between the expressiveness of our mechanism and the previous implementations of code transformation tools for grids. In the last section we present our conclusions.

1.1 Related Work

Although several alternatives to execution of methods in remote objects already exist, such as Java RMI (Remote Method Invocation) [6] for instance, the problem the research in AOP to program grids tries to solve is not how to make remote method calls possible. Instead, research in this field aims to outline a programming environment that allows for programmers to express computation distribution using a method that requires minimum creation or alteration of OOP source code while using programming models similar to standard non-distributed OOP.

Previous researches proposed different methods to automatically embed concerns about grid computing.

OurGrid [1] proposes that all run() methods in Java threads should be transformed in remote methods. The approach is simple, but interferes with the normal operation of threads.

GridGain [2] is an open source project that proposes the use of marker annotations to add semantic to methods that are executed remotely. In addition, it's also possible to plugging schedulers or MapReduce [22] processors.

Both implementations are similar, since both use AspectJ [3] as the base AOP language, both map OOP source code to a respective grid implementations, and both are specially designed for Bag-of-Task creation.

1.2 Limitations in current AOP approaches

However, AspectJ cannot be used efficiently to create changes that are based on reflection over any source code. This limitation is caused by the static nature of its aspect locators (join points) and inter-type declarations. For example, using AspectJ it's not possible, based on a method called getX, to create another method called getXAndUpdateUI whatever is the value of the string X. What AspectJ can do is to trap all the methods called getX, whatever the value of X, and add programming statements, before, after, around, or instead of calls to getX. AOP locator signatures for this purpose are get*.

This limitation also has consequences in the extent until which the tools to transform OOP source code to grids can contribute to class transformations. These tools are committed to offer an alternative to the simple use o APIs to solve the distribution concern, but the static nature of AspectJ limits the transformations to allow only remote method invocations in which each method call is equivalent to the remote execution of a batch procedure and no optimizations regarding data movement or asynchronous communication between nodes can be easily introduced.

This research is aimed precisely on exploring ways to minimize these limitations and on proposing new idioms that can be used to enhance the expressiveness of grid client software without affecting source code manageability.

1.3 Enhancements proposed

As already argued, AspectJ can basically trap calls and alter them arbitrarily. But AspectJ cannot be used to generate new method signatures, which can be used to create a seamless way to do inversion of control, or dependency injection into the source code, for example.

LogicAJ [70] is an AOP language that extends AspectJ to include the dynamic constructs missing in AspectJ. Although LogicAJ could ease the construction of our transformations, we did not use LogicAJ since our strategy is to restrict the application of AOP to source code transformation and to leave the actual declaration of new class structures to a processor we wrote. In short, we chose to express the structures of new classes and methods using an actual algorithm in Java instead of using a syntax provided by an AOP language.

Our proposed mechanism for source code transformation is depicted in Figure 1.1.

The byte code creation process starts with a source code generation tool called Annotation Processing Tool (APT) . This tool is part of Java since version 1.5 and can be plugged into the Eclipse Integrated Development Environment (IDE) [4] to be activated every time any class is changed by the programmer.

The APT is designed not to change any source code provided by the programmer, but can generate additional source code if necessary, which is the feature we used.

We defined a set of Java marker annotations that, when present in programmer's source code, caused the source code to be parsed by our annotation processor.

[1] OurGrid, http://www.ourgrid.org
[2] Grid Gain, http://www.gridgain.org
[3] AspectJ: http://www.eclipse.org/aspectj/
[4] Eclipse IDE, http://www.eclipse.org

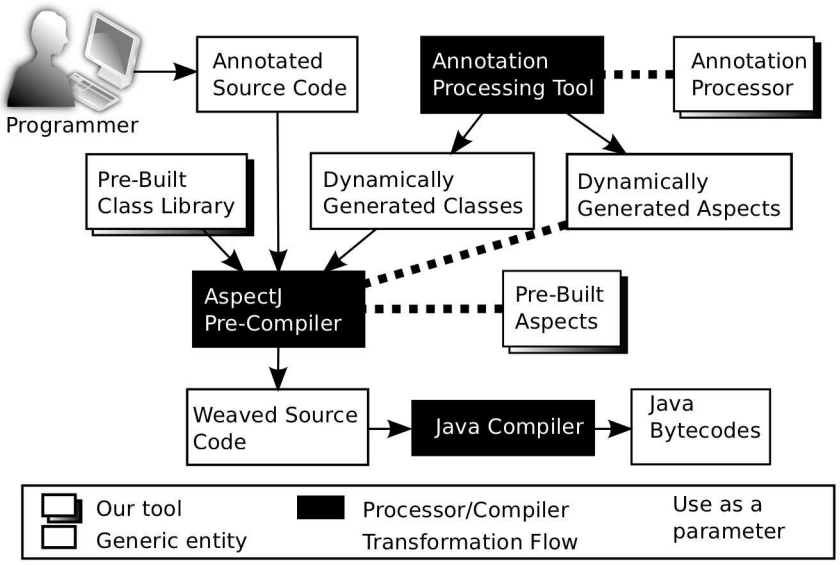

Figure 1.1: Source Code Transformation Workflow

This annotation processor is in charge of detecting which sorts of transformations the programmer wants in his source code and generating supporting classes and supporting aspects that will perform these changes.

As we already stated, annotation processors cannot change the source code written by the programmer. So the aspects generated by our pre-processor, in conjunction with the pre-built aspects will be in charge of altering the functioning of programmers' classes on behalf of the annotation processors.

In the next step, the AspectJ pre-compiler will read all available aspects and apply them to all available classes. Available aspects comprise both pre-built aspects and dynamically generated ones. And the classes that are subject to change comprise the dynamically generated classes, the annotated source code written by the programmer, and the pre-built class library.

Finally, the programmer's source code weaved with additional source code is then passed to the Java compiler which generates the bytecodes that can be then executed directly by any Java Virtual Machine.

Although the compilation process adds several steps to the standard compilation of Java classes, all the compilation is incremental. Which means whenever a class is changed, only the class changed by the programmer, and the dynamic entities related are affected.

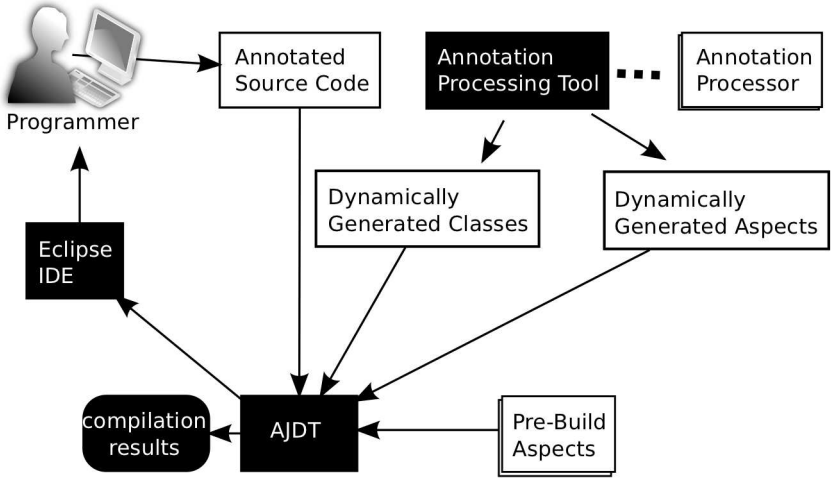

Figure 1.2: Flow of products during compilation step

```
1 public interface RemoteContextReceiver {
2   void setRemoteContext(RemoteContext remoteContext);
3 }
```

Figure 1.3: A common approach to implement dependency injection

1.3.1 Feedback mechanism

As already stated, the AOP approach proposed in previous studies aims to distribute methods and respective context to remote machines. Comparing it with the explicit way to distribute computation (by referencing elements in a API, as is the approach of the Java Cog Kit [78] [43]), the AOP approach lacks of a certain control over what is happening to tasks during their execution.

No information is given to the originator node about where and under which conditions a specific execution of a method is taking place. This separation was created in previous studies to keep the programming model consistent, so no elements of the distribution itself would mix with the source code that implements the algorithm. Nevertheless, there are situations in which programmers need to have more control (or at least more knowledge) over tasks during their execution in order to decide, for example, whether the task should continue or whether it should be aborted.

Also, simple exportation of methods to the grid does not allow these methods to interact with any other object residing in the initiator node or other methods executing in other remote nodes. So in previous proposals, each method execution was necessarily isolated, and the only way to share data with a method was through the method parameters and values returned. In other words, interaction with a procedure executed in a remote node was only possible during procedure initialization or termination.

So, to provide this more refined control, while keeping the same programming model, we introduced a new marking annotation called @GLocal. This annotation, when applied to a method, causes the method to be always executed in the initiator node N0. When a method marked with @GLocal (which we will call from now on a GLocal method) is called from a method executing in N0, the annotation has no effect. But when a GLocal method is called from a method executing in a remote node, the presence of the tag causes the remote execution to halt and the GLocal method to be executed in N0.

To provide information about the remote conditions of the task execution, a remote context object is supplied via dependency injection. The remote context object contains information about the amount of memory available, the number of CPUs, the elapsed time since the start of the remote method and a unique identification number automatically generated. This number can be used in N0 for several purposes, such as controlling which remote node has which partition of data to be processed.

We avoided using a more invasive way to implement dependency injection, such as forcing a class to implement a certain interface as illustrated in Figure 1.3. Implementing this interface forces the class to have a setter method that will receive the injected object.

We chose not to follow this approach because we wanted to keep the injection of remote context objects limited only to the GLocal methods. If we used the dependency injection as above, the setter method would need to be backed up by a field that would be available to all the methods of the class. Since GLocal methods are those methods that bring data from remote nodes to N0, we should avoid other parts of the class to have access to remote context objects.

We created two different approaches to allow methods marked as GLocal to receive remote context objects. The use of this injection is optional. So the programmer can choose not to change his original source code if he doesn't need to get detailed information about the remote context.

The first method to receive remote context objects is by adding a parameter to the GLocal method. This extra parameter will receive an object of the class RemoteContext. So the GLocal method can explicitly read the contents of the remote context object. This strategy allows for a very refined control over the execution of the remote method, but also mix different responsibilities in the same class, because typically GLocal methods will be placed in a class that has also remote methods. So the same class is resolving concerns related to code distribution and the business logic. An example of the first injection approach is the method progressReport in Figure 1.4.

In the second injection method the programmer should add a GContextHandler annotation to a GLocal method. The GContextHandler annotation must have one parameter, which is a class that implements the interface RemoteContextHandler. In Figure 1.4 we can see an example of this approach in the method partialResult.

In this example, the pre-built classes will create an instance of the class MyRemoteContextReceiver and

```
1  public void remoteMethod(int id) throws Exception {
2    for (int i = 0; i < 100; i++) {
3      // some time consuming processing
4      progressReport(id, i);
5      partialResult(partialResult);
6    }
7  }
8
9  // First injection method
10 @GLocal
11 public void progressReport(int id, int percentage, RemoteContext rc) ↩
       throws Exception {
12   // Some local procedure
13   // e.g.: use of rc to estimate the
14   // remaining time
15 }
16
17 // Second injection method
18 @GLocal
19 @GcontextHandler (MyRemoteContextReceiver.class)
20 public void partialResult(byte[] result) throws Exception {
21   // Some local procedure
22   // e.g.: storage of the partial
23   // result
24 }
```

Figure 1.4: Example showing how to use the GLocal methods

give to this instance a reference to the remote context object and a reference to the object from which the method partialResult was called. This second injection method keeps the concerns between classes separated but also gives limited information to the remote context handler objects.

In the example of Figure 1.4 there are actually two methods called progressReport. Their signatures are:
public void progressReport(int id, int percentage, RemoteContext rc)
public void progressReport(int id, int percentage)

The declaration of method m2 is explicit in Figure 1.4 while the method m1 is dynamically generated by our annotation processor and inserted into a class by an AOP inter-type declaration. Attempts to call the method m1 directly will result in a compilation error that tells the programmer to call the method m2 instead.

All compilation issues are shown directly in the programmer's IDE a few seconds after the programmer saved any source code file.

Behind the scenes, the newly created m2 is responsible for creating an instance of RemoteContext and sending a message back to the initiator node asking this node to execute the method m1 with the parameters supplied by the programmer.

The same process used to inject a RemoteContext to GLocal methods can also be used to inject a RemoteContext to GRemote methods, to allow these methods to decide about their own progress without the intervention of the initiator node.

The use of dependency injection in GRemote and GLocal methods could create a dependency with our implementation of the grid infrastructure. To avoid this happening we used all types exposed for programmers, such as RemoteContext, to be Java interfaces. So all elements of our implementation can be replaced using our aspects to map to other implementing classes.

1.3.2 Remote objects

There are cases in which the same data transferred to a remote node needs to be used in more than one remote method call. In the Bag-of-Task model, consecutive calls to remote methods don't allow for reuse of the same data between method calls since each call to a remote method should pass all the data the remote method needs.

```
1  @GridManaged
2  public class PatternFinder {
3    // constructor
4    public PatternFinder(int[] data) {
5      this.data = data;
6    }
7
8    // field
9    private int[] data;
10
11   public int findPattern(int[] pattern) {
12     // searches for pattern in data
13     // and returns the position
14     return position;
15   }
16 }
```

Figure 1.5: Example of a remote object class

```
1 Object around(): call( Class1.new(..)) {
2   return new Class2();
3 }
```

Figure 1.6: Example of advice declaration to replace a constructor

Therefore, the creation of a local memory for the remote methods to store data avoids unnecessary data movement and can consequently increase the overall processing performance.

Also, in some class responsibilities it's more natural for programmers to use private fields to store data used by the object than passing the data in each method invocation. In the class depicted in Figure 1.5, the findPattern method is clearly designed to be called several times with different patterns to be found inside of the same data.

In such a case, creation of remote objects (an instance residing in a specific remote node) also implemented in [73], can solve the problem elegantly while not breaking with the natural object-oriented design and without forcing the programmer to use classes from a specific API to ensure the data location.

Attempts to implement such a behavior using AspectJ will fatally deal with a limitation in the way AspectJ deals with constructors around advices. Figure 1.6 shows a declaration of such advice. In this case, calls to any constructor of Class1 will receive an object of Class2, which necessarily should be a subclass of Class1.

If the remote object class is called ROC and it was created by the programmer, by the time we created our transformation tool we do not know which sort of initialization the constructor of ROC will conduct. So we cannot allow the constructor of ROC to be executed locally to avoid wasting computation time.

Using only standard AOP and OOP, a better alternative would be to count with the programmer's cooperation to use a less natural design to allow for aspects to create a dynamic proxy, which are classes that implement a certain interface in runtime.

A programmer would then need to create not only ROC but also an interface ROCInt that ROC should implement. All references should refer to objects of the type ROCInt instead of the type ROC. Aspects could create a dynamic proxy that implements ROCInt and make the local application to use the dynamic proxy instead of an actual instance of ROC. The dynamic proxy would then forward all the method calls to the actual remote object which was instantiated in a remote node.

Although this method works, it forces programmers to create an extra entity (ROCInt) to support the remote object behavior. So in the final source code the concern about the remote object behavior is not completely isolated in the aspects but also has impacts on the programmer's choices over the structures of his classes and interfaces.

In our implementation we could use the kind of advice of Figure 1.6 without forcing the programmer to create compliant structures. The solution was a dynamic creation of a class ROCS that is subclass of ROC.

All explicit instantiations of ROC are replaced with instantiations of ROCS and all methods of ROC are

```
1 Object around(int[] x): call( PatternFinder.new(int[])) && args(a){
2   // creation of an instance of PatternFinder
3   // in a remote node passing x as a parameter
4   return new PatternFinderSub();
5 }
```

Figure 1.7: Dynamically generated advice declaration to replace a remote object constructor

overridden by methods of ROCS. The methods of ROCS take the method calls and forward them to the actual remote object.

Obviously, ROC cannot be a final class to allow for the creation of ROCS as a subclass of it. To avoid this mistake we raise a compilation error whenever a class marked as a remote object class is declared as final. We understand this last limitation as a trade-off.

To mark a class to be used as a remote object class, we introduced the annotation @GridManaged, shown in Figure 1.5. It can be noticed that the only difference between a remote object class and an ordinary class is the presence of the @GridManaged annotation in the class declaration.

Figure 1.7 shows an example of an advice created by our annotation processor specifically to replace the instantiation of the class PatternFinder. PatternFinderSub is a subclass of PatternFinder that was also created by our annotation processor.

1.4 Method verification

To test the effectiveness of our proposal, we implemented a parallel Apriori [69] algorithm to data mining. The algorithm was implemented both with and without a feedback method. For our tests we used a small implementation of a grid that is based on reliable multicast messages. The programming environment was composed by our tool installed in a Elipse IDE `3.4.1`, AJDT `1.6.3` [5], AspectJ `1.6.4`, and Java `1.6.0_11`.

To take advantage from parallel processing in data mining, we split the data to be analyzed in n pieces and sent one of each pieces to each of the n remote nodes used. The algorithm searches for frequent item sets in a large collection of item sets. Each round of the the algorithm uses a distinct set of item sets. The goal of each round is to exclude infrequent item sets. In the end of each round the infrequent item sets are excluded and new larger item sets are built using the remnant item sets. The new set of item sets is used to the next round. When the frequency of a certain item set surpasses a certain preset value, the item set can be already considered frequent and there is no use to keep searching for new occurrences of it in the data provided.

Then a simple optimization of the algorithm would require each of the n nodes to inform the initiator node in case an item set was already found to be frequent in one of the n pieces in which the data was split.

It's not possible to determine the set of item sets for a certain round before the previous round is over. So the end of each round requires the new set of item sets to be broadcasted to the n nodes.

In the implementation that uses only remote methods, all the remote methods should terminate to allow for N0 to start a new round. Also, the termination of the remote methods implies that the whole data needs to be retransmitted in each round, which for large amounts of data makes the algorithm unusable.

Table 1.1 summarizes the results obtained. We compared the use of simple remote methods (RM column) with the use of remote methods combined with feedback methods and remote objects (RM+FB+RO column).

The results show that the new behaviors offered by our approach can easily allow programmers for using the initiator node as a central point that is responsible for task synchronization and data sharing.

1.5 Conclusions

In this paper we propose a compiling workflow that is able to create dynamic signatures and classes as a result of an analysis done by a procedure written in Java. This feature allowed us for providing more sophisticated system behavior at a very little cost to the programmer in therms of source code bindings. We demonstrated the capabilities of our source code alteration schema to add two computation distribution behaviors, namely, feedback methods and remote object instantiation.

[5] AspectJ Development Tools, http://www.eclipse.org/ajdt/

	RM	RM+FB+RO
Early frequent item discovery	Not shared	Shared asynchronously
Next round	Requires method termination and new method call	Method termination not required
Data movement	Redundant (all data is transferred in each round)	Optimized (only once per remote node)

Table 1.1: Implementation comparison for a distributed Apriori algorithm

On the other hand, the main limitation of our approach, and other similar programming methods, is the lack of control over objects that cannot be managed by our aspects. Some classes, for instance those that follow the singleton design pattern (as introduced in [34]), may have static objects to centralize data. Static objects are not shared among different nodes. In other words, we could not synchronize or copy the whole local environment to the remote nodes, which is a feature left for a future implementation.

Nevertheless, our results show that it's possible to provide a rich programming interface that is able to show distribution misuse in the programmer's IDE using our code transformation tool. We have also shown that such a tool allows programmers to create advanced distribution strategies, while writing source code naturally with almost no new API to learn.

Chapter 2

Message Domains

Abstract

Effectiveness of message passing as the means to provide communication in distributed computing motivates the use of such strategy in highly distributed service environments such as grid and cloud computing. Grids and clouds share some important features such as multiplicity of administrative domains and complexity to create applications to be executed over these platforms. In this paper we argue that fine control over which nodes are accessible for message exchange can ease programming in those scenarios by allowing for partition of concerns in well-defined boundaries. Additionally, message domains may provide simplicity to split layers to implement security and define user and agent roles within a layer.

2.1 Introduction

Highly distributed computing models such as grid and cloud computing are becoming popular as companies and organizations realize that specialization can be achieved without compromising quality of service. Organizations can have part of their software, platform, or infrastructure as a service and have more of their human resources dedicated to their core business.

But computation as a commodity comes at a price. As systems scale, so the complexity to keep them working and evolving increases. Popular systems tend to have a large number of customers, therefore, developers of those systems are expected to receive constant requests for enhancements.

Although message passing is largely used to create distributed applications in a more controlled scenario, the principles of message passing can be also applied to grids and clouds. Adaptations to address the new questions introduced by global scale system distribution must be done.

When computational resources are dynamically provisioned, exchanged messages are not only about data and execution control, but also, for instance, about resource finding, reservation, and payment. When nodes are separated by continents, multicasting must be used wisely. If public networks are utilized to transport the messages, extra care should be taken with security. If more than one organization is exchanging messages in the same grid, message exchange within nodes that belong to the same organization should be isolated from messages that provide communication between organizations. As customer expectations increase due to a high offer of remote services, so does the struggle of development teams to keep the costs of software development and software change under control.

To address these problems, we advocate that global scale distributed systems should be based on reliable multicast domains. Those domains should be flexible to allow for application designers to create custom configurations that will support decisions over module partitioning, user roles, secrecy of data, and division of concerns into independent layers.

The next section will present our argument for the adoption of multicast domains to address those issues. Section 2.3 presents an organization to provide layer modularity using message domains. Section 2.4 presents related research. Finally, we present our conclusions in section 2.5.

2.2 Message domains

Within a message domain, all connected nodes are able to send and receive both multicast and point to point messages to and from all other nodes. Multicast is necessary since, by definition, there is no central

Layer	Section	a	b	c	d	e	f	g
Data mining app.	client	rw		r		rw		×
	data bus			r		rw	rw	×
	data sync			r			rw	×
Sensor data app.	collection	rw		r		×	×	rw
	data sync			r		×	×	rw
Service market			rw	r		rw		rw
Resource brokerage	location	rw	rw	r				
	availability		rw	rw				
Monitoring			w	r	w	w	w	
Authentication	consumer	w			r	w		w
	provider	r			w	r		r

Table 2.1: Example of a message domain structure. Rows are message domains and columns are software agents or nodes. (a) service consumer; (b) resource broker; (c) monitoring agent; (d) certificate server; (e) data mining service; (f) database; (g) sensor. Access to message domains are marked as r for read, and w for write. The symbol '×' means that the message domain does not provide access to the node since access in these cases are considered an architectural error.

control in grids. Limitations in multicast coverage are necessary in grids and clouds due to the large scale of such networks, the presence of more than one administrative domain, and the existence of multiple service providers, implementations, and service instances.

If, on one hand, limiting messages to specific domains forces the existence of bridges between domains, on the other hand, this separation makes it possible for system architects to partition distributed systems horizontally, into stacked layers, and vertically within each layer, into subdivisions that represent distinct roles of software components.

Table 2.1 shows an example of a message domain partition to support two distributed applications (a data mining application and a sensor data collection application) and some accessory services (service market, resource location, monitoring, and authentication). This partition does not define service cardinality, resource location, neither whether the services are written, provided, or maintained by a single or by several organizations.

Each line represents a message domain. Participants of message domain should exchange messages freely, therefore ideally they should share the same protocol. Conversely, different domains could adopt different protocols and translation may be necessary.

Each layer is supposed to offer a specific service to the stack. To organize its internal structure and to protect message buses that should be private to its components, layers may also be divided into sections. For example, in Table 2.1, service consumers (a) cannot access the availability section, since this section should be used only for resource brokers and monitoring agents to communicate.

The resource brokerage layer is used to support for replicas of the same service and dynamic service location. Resource brokers are commonly utilized to optimize access to resources in grids [26]. If, for the sake of high availability or to optimize performance, a set of resource brokers is used as in [26], these brokers need to keep a database of resources synchronized among all of them. As resource availability in a grid is expected to be dynamic and unpredictable, this resource database is also expected to change frequently.

Message traffic used to keep synchronization of the distributed resource database should be kept away from non-resource broker nodes to minimize network traffic and to simplify development of agents. Also, data about resources should be kept among authorized hosts only, as paid service providers could use access to those messages to cheat in the competition for service consumers.

In this example, the design decision was that distributed databases do not need to verify the identity of consumers, since databases are only reachable through the 'data bus' section. Also, it was decided that it is not a threat to secrecy or security if all data mining service instances receive data from databases. These assumptions reflect into simpler source code for the database nodes.

Structure of processes can influence the design of those domains. For instance, participating in service bidding may be a complex task as it involves several steps until the negotiation is done. Service consumers should not have to know how to participate in service bidding since having to do so would create a dependency between consumers and resource providers. So instead of having direct access to the service market layer, a service consumer needs to indirectly utilize the service market via communication with a resource broker. A resource broker will then intermediate the negotiation and use the service market message domain to

			client	infrastructure agents	application specific agents
	Layer	Section	a	b c d	e f g
Application Layers	Data mining app.	client	α	β	γ
		data bus			
		data sync			
	Sensor data app.	collection			
		data sync			
Infrastructure Layers	Service market			δ	ϵ
	Resource brokerage	location			
		availability			
	Monitoring				
	Authentication	consumer			
		provider			

Table 2.2: Partitions of Table 2.1. (α) entry point; (β) requirements; (γ) application organization; (δ) supporting infrastructure organization; (ϵ) infrastructure services

exchange messages with the data mining services and the sensors.

From the view point of the data mining service (e), it is necessary to participate on six message domains. If data received from one of the message domains needs to be fed into another message domain, and the two message domains do not use the same protocol, translation may be necessary.

2.3 Modularity

Design of message domains should provide a framework for distributed software construction. Communication constraints reinforced by the message domains should serve to outline the degrees of liberty the developers of each layer have. For example, in our example a constraint could be that layers declared by applications should enable the monitoring agent to read the messages exchanged.

For example, a domain can be used by replicas of the same agent to communicate. This is the case of the 'data sync' domains, which are used by databases to synchronize their data. Because other nodes do not need to receive messages regarding data synchronism, there should be a section in which only databases participate. Since the two applications should be independent, each of them can declare its own data sync domain. An alternative solution would be to provide a common database agents and layer.

There is a fundamental difference between domains that provide services for the infrastructure as a whole (e.g. the monitoring layer), and domains that provide services for a specific application (e.g. the data sync). In face of these differences, the cells in Table 2.1 can be categorized according with the architectural role they play. Table 2.2 shows the map of Table 2.1 into those roles. The motivation to create such map is to define rights and responsibilities for each component.

2.3.1 (α) Entry point

Message domains in this partition are interfaces with service clients. Infrastructure nodes can be used by distributed applications as a distributed middleware container if the application does not allow clients to connect in α and if the application delegates the interface with clients to the infrastructure layers.

2.3.2 (β) Requirements

The supporting infrastructure can use this partition to force applications to comply with a certain set of requirements. In the example, it is required that all message domains in the application layers should allow the monitoring agent to listen to the multicast messages. Obviously, this requirement does not guarantee that the monitoring agent will be able to understand the traffic in the application layers. Therefore, requirements declared in this partition should be considered only communication capability requirements.

2.3.3 (γ) Application organization

Applications may require the existence of private message buses to work. This is the case of the data mining application in Table 2.1. Message domains in this partition aim to provide this sort of intra-application message exchange. As the internal functioning of applications should be not exposed to neither other applications nor to the infrastructure, each application has its own sub-partition inside of γ, and sub-partitions do not overlap.

The symbol '$\times$' in Table 2.1 marks the situations in which message domains should not exist since they would be used to provide direct application to application communication. This is considered a design flaw, since it creates internal dependencies between applications. If one application needs to utilize another as a service, this communication should occur in an entry point.

2.3.4 (δ) Supporting infrastructure organization

This partition is invisible to applications and can be used by infrastructural services to exchange messages.

It is common that infrastructure layers have intrinsic relations with other infrastructure layers. This is the case, in our example, of the dependencies between the resource brokerage and the monitoring layers. In contrast with γ, infrastructure layers are allowed to directly communicate (and therefore have dependencies) in δ.

2.3.5 (ϵ) Infrastructure services

A partition for the supporting infrastructure to provide services to the applications. Designers of application layers are free to utilize those services or not. For example, in Table 2.1, the sensor service utilizes the authentication service, but not the monitoring service.

2.4 Related work

In global-scale environments, it is expected that message senders and message receivers could be two pieces of software developed and/or maintained by different teams. Service-orientation is used as a solution to decouple dependent software components, but wrapping message passing in service calls adds an abstraction whose consequence is the impoverishment of distributed algorithms. Fine control over data flow and execution, operations such as receiving buffers from participants, and sending messages to more than one node, are all features not present in standard service orientation.

Although in the past some versions of Message Passing Interface (MPI) to grid computing were created [31] [18] [3], the objective of these libraries is to make message passing programming style available to grid computing, which allows for the construction of distributed parallel machines, instead of an infrastructure of distributed services.

Here, we assume that a library for message passing is already available. Instead of a version of a message passing interface to grid, we propose an additional control over the messages that are transferred by the grid infrastructure. This control can be used to enhance component organization in development using an underlying message passing mechanism.

In a previous paper [55] we investigated some methods to translate local OOP into grid-enabled local applications. During translation, OOP statements are translated into routines that utilize message passing to data transfer and execution control. We expect to apply message domain partitioning to define visibility between software components in OOP. For instance, a service that is not visible to application layers should be represented by OOP classes that are not accessible by application classes. This cohesion between message exchange framework and programming constructs can be ensured by applying special pre-processors in a general purpose OOP language or by using a domain-specific programming language created for this purpose.

The configuration for message exchange we presented should work in cooperation with, or be part of a infrastructure for governance, as discussed in [23]. For example, policies that control access to resources or services can be reinforced by creating classes of clients. Each class could be represented by a column in the entry point partition. The model in [23] also suggests that tools should be used to retrieve meta-information about service infrastructures, which can be implemented using directory services such as Universal Description, Discovery, and Integration (UDDI) [4] . In addition, the model we propose can also benefit the development of services similar to those currently in use in grid computing, such as the grid resource allocation management (GRAM) [32].

2.5 Conclusions

The structure of grid and cloud computing requires complex message deliver systems in which message recipients should be carefully selected to provide scalability and protection against data stealing, abusive behavior, and attacks. Nevertheless, message passing can be utilized in highly distributed systems, if correct adaptations are made to ensure that it supports dynamic and unpredictable server availability, long latency, and cross-administrative domain issues.

We propose a process in which first the outline of a distributed system architecture is created using message domains (e.g. by a consortium that decides how the grid will be managed). Next, infrastructure layers can be designed and implemented to support applications. Finally, this structure will be utilized to free applications from having to implement base services.

From the perspective of network usage, it can control complexity and can be used to minimize network traffic. Also, partitioning multicast message domains can simplify the design of service-oriented applications since they can be used to guide data flow. For these reasons, we argue that message passing should be a fundamental building block of grid and cloud computing environments.

Part II

FSM approach for contracts

Chapter 3

Versioning for highly distributed services

Abstract

Distributed service contracts have been used as a way to allow for independent evolution of clients and service implementations. As grid and cloud services become more popular, and as service composition becomes common place, so one can expect that the life cycle of service contracts to become more dynamic. This scenario presents new challenges to service and client evolution. More precisely, detection of compatibility becomes a more critical aspect of service specification management. Compatibility is specially hard to assess when it depends on aspects others than the signatures of service methods. Composites of services may also be externalized as services themselves, then creating a manageability problem to service developers that has impacts in the whole service project life cycle. We argue that client-service contract compatibility is more complex than simple service method signature verification, but, for the best of our knowledge, no supporting model was yet created to tackle this issue. We propose a method to allow for FSMs inside of services to be used to enhance compatibility verification. We propose that FSMs should be utilized as an additional element to describe which sort of service contract clients require from servers, and which service contract each service offer to clients. In this paper we introduce our method to assess compatibility between a generic client source code and legal sequences of service methods calls. Finally, we exemplify our method utilizing a remote data mining service.

3.1 Introduction

The complexity of highly distributed service environments, such as grid and cloud computing, present several management challenges. Among the neglected questions is the problem concerning the evolution of such systems along time. As the complexity of service composites increases, so the manageability to evolve such services becomes harder to achieve. The same service specification may have several implementations, may be available in several organizations, and may be consumed by several different clients. In this scenario, we need a method to quantify compliance between clients, service specifications, and service implementations. While compliance between service implementation and service specification is usually trivial, compliance between clients and service specifications may be much more difficult, or even impossible, to determine.

In this paper, we present a method to verify client source code compliance based on FSMs in the server side. We do not aim to eliminate the compatibility problem, as we believe compatibility issues are the result of the natural evolution of distributed services. There is a number of constraints in software creation (such as cost, design mistakes, and technology momentum change) that leads components to become incompatible as versions evolve. Instead of offering a solution, our model aims to provide additional information for developers to detect and manage compatibility problems. Our method only guarantees that the incompatibilities are accurately found, but cannot guarantee that all incompatibilities were detected. Nevertheless, the strength of our method is that it can be easily applied to any section of source code in isolation to perform a best-effort to search for invalid method call sequences.

The rest of this paper is organized as follows. The next section will present work related to this topic. Next, Sections 4.3 and 3.4 we will breafly present, respectively, the distributed services model we utilize and the intermediate representation we utilize to represent source code. Section 3.5 presents our method to

analyze compatibility. Section 3.6 illustrates our method using a remote data mining service as an example. The last section concludes our discussion.

3.2 Related Work

The questions about distributed service versioning were identified by Vinoski [76]. A specific analysis for versioning in Service-Oriented Architecture (SOA) focusing on decoupling of service and clients was presented by Lublisky [50]. Both papers analyze the problem from the perspective of the mechanisms responsible for service and client decoupling, but no attention was given to the role of clients' source codes.

Service states were recognized as an important feature of distributed systems. According to an informational document [30] about Open Grid Service Architecture (OGSA), published by the Global Grid Forum (GGF), the Web Services Resource Framework (WSRF) standard [20] was created by the Organization for the Advancement of Structured Information Standards (OASIS) consortium [1] to meet the requirements for state representation and manipulation in grid services.

The problem of having derivation from an original service was investigated by Ponnekanti and Fox [68]. This paper proposes a model to deal with compatibilities between web services from distinct vendors that are originated from the same root web service. In contrast, here we deal with problems originated from successions of versions of the same service contract. Besides the differences, in both cases detecting service compatibilities is identified as an important factor to the maintenance of services. Their compatibility model is based on Web Service Description Language (WSDL) comparisons but here we propose a new and more detailed service contract to identify changes in service implementation internals, that could present impacts in service compatibility.

Weinreich et al. [79] propose a model in which products are defined as a set of clients and services, and both may evolve independently. In our model there is a separation between service implementation and service specification (service interface), which allows for multiple providers to offer alternative implementations of the same service contract.

In their proposal, the concept of compatibility is based on comparisons between different versions of the same service interface. This simplicity is motivated by the control offered by the concept of 'product', in which both clients and services participate. In this model, version numbers are used by clients to choose between utilizing the latest version of the service that follows the same compatible interface or to select a specific minor implementation.

In our model we do not specify whether a client wants to follow the latest version of a service or not, since we focus on compatibility detection based on non-functional aspects of services and clients. We do not propose strategies to service evolution and version branches as defined by Weinreich et al., since we assume that multiple service implementations may be provided by a number of development teams. In our model we assume that the number of client applications can be too large for compatibility analysis to be conducted manually.

A service change model was proposed by Leitner et al. [49], but their proposed model only comprehends functional characteristics of services, while here we precisely try to include internal behaviors of remote services that are relevant for compatibility analysis.

Versioning models for SOA generally discuss versioning of Web Services but, according to Vogels [77], Web Services should not be understood as distributed objects. Therefore we argue that versioning in the context of distributed objects (as an abstraction that can be concretized by Web Services or other underlying distribution technology) may support more robust compatibility detections embedded in the very programming tools utilized to build and validate clients.

Finally, compatibility between two web services was defined in [14]. Compatibility is based on comparisons between FSMs in each web service. While in their model the transitions are messages exchanged between the services, here we modeled state transitions as method calls. An earlier work [13] related to [14] proposed that compatibility analysis could ultimately allow for automatic service composition. While in our work the service-side is similar, in behavior, to the web services as modeled in [13] and [14], our model for the client-side is more complex, as we accept a generic client source code, even if it cannot be translated into an FSM.

[1] OASIS consortium website: http://www.oasis-open.org/

3.3 Distributed services model

Here we assume that a service is represented by a contract that may evolve along time. A contract $C(M, F)$ is composed of a list M of method prototypes, and a set F of P FSMs which express relationships of precedence between methods, when such relationships exist. Each FSM $\phi(S, S_0, T)$ is composed by a list of states S, a set of initial states $S_0 \subseteq S$, and a list of transitions T. Each transition t is associated with zero or more methods of M. Then, we denote $t(s_a \rightarrow s_b, X \subseteq M)$, where $s_a \rightarrow s_b$ means a transition from the state s_a to the state s_b, and X is the set of methods associated with t. In short:

$$M = \{m_1, m_2, \ldots, m_N\} \tag{3.1}$$
$$F = \{\phi_1, \phi_2, \ldots, \phi_P\} \tag{3.2}$$
$$S = \{s_1, s_2, \ldots, s_Q\} \tag{3.3}$$
$$T = \{t_1, t_2, \ldots, t_R\} \tag{3.4}$$

A transition $s_a \rightarrow s_b$ being associated with a method m means that if m is called when the current state is s_a, then either the termination of m, or an asynchronous process started by m, is able to change the state to s_b. If a method is associated with more than one transition, decision about which transition to perform will be made by the service. A state being associated with a set transitions automatically means that the method cannot be called when a state is such that none of these transitions can be performed. In other words, when the current state is not the initial state of any of the transitions the state can perform, a state cannot be executed. States that are not related to any transition are supposed to tolerate any current state.

FSMs do not change their states unless a client calls a service method, to ensure that service-side states are in accordance with the contract specification that the client expects. Each remote service behaves, then, as a non-shared object, from the client's perspective.

Nevertheless, the decisions of which state transition to execute may depend on factors others than those visible to the client, such as other clients competing to utilize a shared resource concurrently.

In our examples we assume that a remote service is presented to the client as a local object. Client source code will then utilize service methods by simply calling methods of the local object.

As for the transparency of service-side FSMs, we do not assume that the service keeps the client updated about which is the current state of each FSM, since the process described here is a validation that can be performed during the process to build a client. However, as we will discuss later, compatibility assessment may be inaccurate during client build. Therefore, there are benefits to satisfactory client-server cooperation if the client knows which are the states in the service side during the execution of the distributed system.

If it is the case that the client is aware of the states in service-side, the same method we introduce here can be utilized during client execution to detect a service behavior that differs from the expected by the client.

3.4 AST intermediate representation

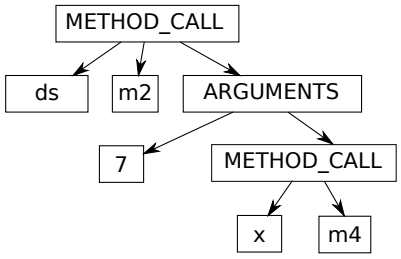

Figure 3.1: AST for 'ds.m2(7, x.m4());' containing imaginary nodes METHOD_CALL and ARGUMENTS

Although we will utilize Java as the concrete OOP language for our examples, we will utilize an abstract syntax representation of source code during our argumentation, as our proposal is not specific to Java, but generic to any programming language that can be translated into the same intermediate representation. We will utilize Abstract Syntax Trees (ASTs) as the intermediate source code representation that abstracts a concrete syntax. Figure 3.1 is a graphical representation of the AST of the source code fragment `ds.m2(7, x.m4());`.

```
1 (METHOD_CALL ds m2 (ARGUMENTS 7 (METHOD_CALL x m4)))
```

Figure 3.2: Textual representation of the AST from 'ds.m2(7, x.m4());'

Table 3.1: Syntax of the EBNF used

Syntax	Meaning
`x : y`	Declaration of the rule x
`x | y | z`	x, y, and z are mutually excluding alternatives
`x?`	x is optional (may appear once or do not appear)
`x*`	x may appear zero or more times
`x+`	x may appear one or more times
`^(X y z)`	X is a tree node. y and z are children of X
`(x y)`	parenthesis, as in ordinary mathematics

The equivalent textual representation of this tree is shown in Figure 3.2. The first word inside of a parenthesis is a node and the following elements are child nodes or leafs. `DECLARATION`, `METHOD_CALL`, and `ARGUMENTS` are called imaginary nodes, since they do not have an immediate correspondent representation in the source code. A programmer knows that the source a source code fragment refers to a declaration since there is a type modifying a variable name. Relationships between tokens, and not a specific token, allows us to draw such a conclusion. But tree manipulation is easier when nodes are explicitly represented by imaginary nodes that specify their function.

We will represent imaginary nodes with all capital letters, and leafs as they appear in the original source code.

```
1 cMethodCall : ^(METHOD_CALL id+ cArguments?)
2 cArguments : ^(ARGUMENTS (literal | id | cMethodCall)+)
```

Figure 3.3: Partial tree grammar for client source code

Finally, to describe the structure of our ASTs, we will utilize the Extended Backus-Naur Form (EBNF) [29] , which is used to describe context-free languages. There are several notations for the EBNF. The notation we will utilize is derived from regular expressions and is used in ANTLR [67], which is a language tool to create lexers and parsers, among other applications to manipulate and analyze source code. The notation is roughly the same as the one utilized to describe the XML syntax [1], and the syntax is summarized in Table 3.1.

The grammar of `METHOD_CALL` imaginary node is described in Figure 3.3 using EBNF. Based on Figure 3.3, a `METHOD_CALL` imaginary node should therefore necessarily have at least one identifier and may have one `ARGUMENTS` node. The words `cMethodCall` and `cArguments` are rule names. The prefix 'c' stands for 'client', in contrast with two other grammars we will introduce later.

Here we do not want to provide a complete specification of AST structures, as these specifications are too big for this text. For example, in Figure 3.3, we not specify the structure of `id` and `literal`. Along the text, we will omit these declarations as their meanings can be inferred.

3.5 Compatibility assessment process

For the sake of clarity, we will separate the compatibility assessment process in two parts: simple sequences of method calls (Section 3.5.1), and method calls within loops and conditionals (Section 3.5.2).

Compatibility assessment is a three-step process. The first step is parsing a client source code (e.g.: Figure 4.1) to retrieve its AST representation (e.g.: Figure 4.2), that conforms to the grammar in Figure 3.3. The second step is transforming the AST representation into a reduced AST (e.g.: Figure 4.3) that contains only the elements that are relevant to compatibility analysis. The third part is to verify the reduced AST against the service contract. The algorithm for this third part utilizes a string to represent the current step in the assessment process. Each of these three steps will utilize its own grammar.

```
1 int i = 0;
2 DS ds = serviceFactory(parameters);
3 ds.m1(i);
4 String s = ds.m2();
5 ds.m3();
```

Figure 3.4: Simple client source code in Java.

```
1 (VARIABLE_DECLARATION i int 0)
2 (VARIABLE_DECLARATION ds DS (METHOD_CALL serviceFactory (ARGUMENTS ↩
     parameters)))
3 (METHOD_CALL ds m1 (ARGUMENTS i))
4 (VARIABLE_DECLARATION s String (METHOD_CALL ds m2))
5 (METHOD_CALL ds m3)
```

Figure 3.5: AST of the source code in Figure 4.1

When naming the grammar rules, we will utilize the prefix 'c' for the **client** grammar (step 1), the prefix 'r' for the **reduced** AST grammar (step 2), and the prefix 'a' for the grammar to describe the string used in the **assessment** process (step 3).

3.5.1 Simple sequences of method calls

We will start by analyzing the simplest case of client source code: a simple sequence of method calls without any execution flow control block (`IF` or `LOOP` blocks, for instance). Let's take the client source code depicted in Figure 4.1. The object `ds` is a reference to a remote service whose class is `DS`.

Step 1 - Construction of client source code AST

The actual process to generate an AST from the source code involves parsing algorithms that are out of the scope of this paper. Besides, each client programming language will specific parsing strategies and parsing rules [67]. As an example, a complete AST for the source code in Figure 4.1 could be as in Figure 4.2.

Step 2 - Construction of a reduced AST

Not all elements of this AST interest us when we want to check compatibility. We can then reduce the AST to only the elements that are relevant for the role of a remote service client. A reduced AST is depicted in Figure 4.3. We can see that the first line in Figure 4.2 can be ignored since it does not have any call to `ds`. Also, we can notice in the second line in Figure 4.3 that the argument `i` was replaced by it's type `int` since signature verifications only compare types. Objects `ds` where replaced by their type `DS`. Figure 3.7 shows the grammar for the reduced AST.

Method signature verification utilizing the AST in Figure 4.3 against the method signature definition of a service interface is a straightforward process.

In summary, to create the reduced AST, we will:

- ignore the statements that are unrelated to the remote service,
- replace initializations of remote service references with an `INITIALIZATION` node. As we do not base our method on any specific service technology, ow references to remote services are actually created (and therefore how creation is detected) is out of the scope of this paper.
- replace object references with remote service types,
- replace method calls with signature-like subtrees, and
- extract method call subtrees from other statements (in our example, we extracted `(METHOD_CALL ↩ DS m2)` from `(VARIABLE_DECLARATION s String (METHOD_CALL ds m2))`)

```
1 (INITIALIZATION DS)
2 (METHOD_CALL DS m1 (ARGUMENTS int))
3 (METHOD_CALL DS m2)
4 (METHOD_CALL DS m3)
```

Figure 3.6: A reduced AST derived from the one shown in Figure 4.2

```
1 rInitialization : ^(INITIALIZATION remote_service_type)
2 rMethodCall : ^(METHOD_CALL remote_service_type method_name rArguments?)
3 rArguments : ^(ARGUMENTS type+)
```

Figure 3.7: Grammar of METHOD_CALL and ARGUMENT in reduced AST

Step 3 - Compatibility assessment through equivalent linear transformation matrices

Let us consider an FSM $\phi(S, S_0, T)$ with Q states as in (4.3). Each state will be represented by a unit vector in a Q-dimensional space. The set of initial states S_0 will be represented by a matrix in which the diagonal has a value 1 in the positions equivalent to each of the initial states, and all other entries contain zeros. For instance, for an FSM in which $S = \{s_1, s_2, s_3, s_4, s_5\}$ and the initial states are s_1 and s_3, S_0 will be translated into $S_0 = diag(1, 0, 1, 0, 0)$. In the extreme case in which $S_0 = S$, the equivalent matrix will be the identity matrix I_Q.

Each method will be represented by an Linear Transformation Matrix (LTM) from $\mathbb{N}^Q$ to $\mathbb{N}^Q$, that will have a number 1 placed at each state transition the method is able to perform, according with the following map:

$$\begin{bmatrix} s_1 \to s_1 & s_1 \to s_2 & \cdots & s_1 \to s_Q \\ s_2 \to s_1 & s_2 \to s_2 & \cdots & s_2 \to s_Q \\ \vdots & \vdots & \ddots & \vdots \\ s_Q \to s_1 & s_Q \to s_2 & \cdots & s_Q \to s_Q \end{bmatrix} \tag{3.5}$$

For instance, let's consider a service contract in which there is one FSM with two states s_1 and s_2. A method that could cause the transitions $s_1 \to s_1$ or $s_1 \to s_2$ will be represented by $\left(\begin{smallmatrix} 1 & 1 \\ 0 & 0 \end{smallmatrix}\right)$. Methods that are not related to any state transition will be represented by the identity matrix I_Q.

A sequence of method calls is equivalent to applying a composite linear transformation. Then, for a simple sequence of w method calls, where $m_{(i)}$ represents the i-th method called, the equivalent linear transformation m' is simply the product of the linear transformations of the methods called:

$$m' = \prod_{i=0}^{w} m_{(i)} \tag{3.6}$$

For example, the equivalent linear transformation of the AST in Figure 4.3 is given by $m' = S_0 \cdot m_1 \cdot m_2 \cdot m_3$.

The value of each entry of a composite linear transformation represents the number of ways each transition can be made. For example, if the composite linear transformation $m_1 \cdot m_2 \cdot m_3$ is capable of leading s_1 to s_2 using the paths $s_1 \xrightarrow{m_1} s_3 \xrightarrow{m_2} s_3 \xrightarrow{m_3} s_2$ and $s_1 \xrightarrow{m_1} s_4 \xrightarrow{m_2} s_5 \xrightarrow{m_3} s_2$, then the entry that corresponds to the position $s_1 \to s_2$ will have a value 2.

Therefore, a composite linear transformation equals to a zero matrix 0_Q means that the sequence of method calls that generated the linear transformation cannot lead any state to any other state. In other words, the sequence of method calls is invalid for the contract. The point, in the successive multiplications, in which the matrix becomes 0_Q is where a remote service method is called over a state that it cannot accept as an input state. In other words, the statement that causes a matrix to become 0_Q is the statement responsible for the incompatibility.

The application of an `INITIALIZATION` after the execution of some methods will reset the FSM to one of the initial states. The consequence for the calculation of the equivalent linear transformation is that the current LTM will be simply replaced by S_0. We will translate the application of a subtree (↩ `INITIALIZATION X`) over the current LTM m into the operation

```
1 (METHOD_CALL DS m1)
2 (METHOD_CALL DS m2)
3 (INITIALIZATION DS)
4 (METHOD_CALL DS m3)
5 (METHOD_CALL DS m4)
```

Figure 3.8: An example of an initialization surrounded by method calls

```
1 rLoop : ^(LOOP rBlock)
2 rCond : ^(COND rBlock rBlock+)
3 rBlock : ^(BLOCK id rStatement+)
4 rIdentity : IDENTITY
5 rStatement : rLoop | rCond | rMethodCall | rIdentity
```

Figure 3.9: Reduced AST grammar rules of LOOP and COND imaginary nodes

$$m \cdot 0_Q + S_0 \equiv m \odot S_0$$

We will utilize the symbol '$\odot$' to represent this operation, and will consider this operation atomic. I.e., we will not interpret the 0_Q obtained before the addition as an incompatibility.

For instance, the equivalent linear transformation to the reduced AST in Figure 3.8 will be

$$m' = m_1 \cdot m_2 \cdot 0_Q + S_0 \cdot m_3 \cdot m_4 = m_1 \cdot m_2 \odot S_0 \cdot m_3 \cdot m_4$$

The calculation process will find an incompatibility if $(m_1 \cdot m_2)$, $(S_0 \cdot m_3)$, or $(S_0 \cdot m_3 \cdot m_4)$ are equal to 0_Q.

3.5.2 Method calls within loops and conditionals

Step 1 - Construction of client source code AST

At this point it should be clear that the actual process to build an AST from a programmer's source code is irrelevant as long as we can obtain a reduced AST according with our specification. Therefore, we will skip this explanation.

Step 2 - Construction of a reduced AST

We cannot expect to be feasible to inspect loops and conditionals created by a programmer to try to predict how many times or if the statements within them will be executed. Instead, we will simply assume that loops may repeat zero or more times and that all the blocks enclosed by conditionals can be executed.

In reduced AST, all loops (for, while, do-while, etc) will be represented by the imaginary node `LOOP` and all conditionals (if, if-else, switch, etc) will be replaced by the imaginary node `COND`. If a conditional has no default block (for example, an `if` without an `else`), we will add a default block containing the identity matrix I_Q (the `IDENTITY` node in Figure 3.9), which will make calculations simpler. The grammar of both imaginary nodes is shown in Figure 3.9. `BLOCK` is an auxiliary imaginary node that groups statements. Each block will have a unique identification that is sequentially created to simplify the step 3, as we will see later. A statement may be `LOOP`, `COND`, `METHOD_CALL`, or `IDENTITY`. A loop must have exactly one block, while a conditional may have two or more blocks [2].

Each Execution Control Subtree (ECS) (`LOOP` and `COND`) should contain at least one `METHOD_CALL`↩, otherwise the ECS should be eliminated. ECSs may contain other nested subtrees. In all cases, there will be a terminal subtree without child ECSs, but only one or more `METHOD_CALL` nodes.

[2] Multiple blocks may happen, for instance, in successive if-elseif-elseif-..., or in switch blocks

```
1 aLTM: '[' number+ (';' number+)* ']'
2 aVagueLTM: '?[' number+ (';' number+)* ']'
3 aBlock: '(' id aBlockContent+ ')' aPendingMult?
4 aLoop: '<' id aBlockContent+ ':' aBlockContent+ '>' aPendingMult?
5 aPendingMult: '*' aLTM
6 aBlockContent: aLTM | aVagueLTM | aBlock | aLoop
```

Figure 3.10: Grammar rules of assessment strings

Table 3.2: Assessment string initialization

Outermost element	Initialization
(BLOCK A)	(A I_Q)
(COND (BLOCK A) (BLOCK B) ...)	(A I_Q) (B I_Q) ...
(LOOP (BLOCK A))	<A I_Q : I_Q >

Step 3 - Compatibility assessment through equivalent linear transformation matrices

As we already stated, in this step we will utilize a string to represent our solution to find transformation matrices. The grammar for this string is shown in Figure 3.10.

This step is composed of three sub-steps: (3.1) initialization, (3.2) application of each child, and (3.3) reduction to a set of LTMs.

(3.1) **Initialization** is how the assessment string is created. We take the outermost element (a `BLOCK`, a `COND`, or a `LOOP`), and create the string according to Table 4.1.

(3.2) **Application of each child** combines a block A with its children A1, A2, ... according to Table 4.2. When all children of A were applied, and A does not have any inner blocks to be reduced, we can reduce A to a set of LTMs, which is the next step.

(3.3) **Reduction to a set of LTMs** utilizes Table 4.3. The meaning of $M'(A)$ will be explained later.

To clarify the meanings and usage of Tables 4.1, 4.2, and 4.3, let us consider the reduced AST in Figure 4.5, derived form the source code in Figure 4.4. In Figure 4.5, block are identifiers as A, B, C, and D. The column on the left represents children identification. The symbol '`--`' means 'no child in this line', while letters are the parent name and numbers are the child order. For instance, `A2` means 'the second child of the block A'. The `IDENTITY` node was added to the reduced AST (line `D1`) since the conditional does not have an `else` block.

Figure 4.6 shows the sequence of values of the assessment string. The column on the left shows child application or block reduction. Child application is marked by a letter followed by a number (e.g.: `B1`), while block reduction is marked by the block identifier followed by an exclamation mark (e.g.: `C!`). The grammar in Figure 3.10 states that matrices should be represented as a string, but for generality and clarity we will utilize a symbolic notation (e.g: I_3 instead of `[1 0 0; 0 1 0; 0 0 1]`).

Table 3.3: Application of a child node to its parent node

Parent	Child	Result
(A $A1$ $A2$ ...)	(INITIALIZATION X)	(A $A1 \odot S_0$ $A2 \odot S_0$ ...)
(A $A1$ $A2$ ...)	(METHOD_CALL X m)	(A $A1 \cdot m$ $A2 \cdot m$ ...)
(A $A1$ $A2$ ...)	(COND (BLOCK X) (BLOCK Y) ...)	(A (X $A1$ $A2$ ...) (Y $A1$ $A2$ ...) ...)
(A $A1$ $A2$ ...)	(LOOP (BLOCK X))	(A <X $A1$ $A2$ ... : I_Q >)
<A $A1$ $A2$ ... : $B1$ $B2$ ... >	(INITIALIZATION X)	<A $A1$ $A2$ ... : $B1 \odot S_0$ $B2 \odot S_0$ ... >
<A $A1$ $A2$ ... : $B1$ $B2$ ... >	(METHOD_CALL X m)	<A $A1$ $A2$ ... : $B1 \cdot m$ $B2 \cdot m$ ... >
<A $A1$ $A2$ ... : $B1$ $B2$ ... >	(COND (BLOCK X) (BLOCK Y) ...)	<A $A1$ $A2$ ... : (X $B1$ $B2$ ...) (Y $B1$ $B2$ ...) ... >
<A $A1$ $A2$ ... : $B1$ $B2$ ... >	(LOOP (BLOCK X))	<A $A1$ $A2$ ... : <X $B1$ $B2$ ... : I_Q >>

Table 3.4: Reduction to LTMs

Element	Reduction
(A m_1 m_2 ...)	m_1 m_2 ...
(A m_1 m_2 ...) * m_x	$m_1 m_x$ $m_2 m_x$ ...
<A b_1 b_2 ... : i_1 i_2 ... >* p	$M'(A)$ = all possible outcomes of equations (7)

```
1 ds.m1(); ds.m2();
2 if(theDayIsSunny){
3   ds.m3();
4   ds.m4();
5 }
6 elseif(itIsSnowing)
7 ds.m5();
8 m6(); m7();
```

Figure 3.11: Example of client source code containing a conditional

```
1  -- (BLOCK A
2  A1      (METHOD_CALL DS m1)
3  A2      (METHOD_CALL DS m2)
4  A3      (COND
5  --          (BLOCK B
6  B1              (METHOD_CALL DS m3)
7  B2              (METHOD_CALL DS m4)
8  --          )
9  --          (BLOCK C
10 C1              (METHOD_CALL DS m5)
11 --          )
12 --          (BLOCK D
13 D1              IDENTITY
14 --          )
15 --      )
16 A4      (METHOD_CALL DS m6)
17 A5      (METHOD_CALL DS m7)
18 -- )
```

Figure 3.12: Reduced AST of the source code in Figure 4.4

We start with the outermost block A and initialize it with I_Q (first line in Figure 4.5). Next we apply each of the children of this block. Application of a matrix to a block will distribute the multiplication to each of the elements inside of the bock (second rule of Table 4.2). In the example, we apply the child `A1` to the first line and obtain the line `A1` in Figure 4.5. We repeat the same process and apply A2 to obtain the line `A2` in Figure 4.5. In line `A3` we need to apply a `COND` child. The effect will be the creation of three inner blocks with the contents of the parent (third rule of Table 4.2). Next, we apply A4 to the current string. Now we have three inner elements in A, therefore we need to multiply each of the three elements by m_6.

A multiplication of a matrix by a block will add the matrix to the pending multiplications slot of the block represented by an asterisk followed by the method name (rule `aPendingMult` in Figure 3.10). In other words, the multiplication is marked to be executed in the future. The last child of A is A5. Again, we will distribute the multiplication to the three inner blocks B, C, and D. But, this time, as the blocks already have a pending matrix to multiply, we can simply replace m_6 with $m_6 m_7$.

Until this point (line `A5` in Figure 4.6) we know that we have at least three alternative method sequences, and that all method sequences will necessarily start with $m_1 m_2$ and end with $m_6 m_7$. Therefore, if we find $m_1 m_2$ or $m_6 m_7$ to be equal to 0_Q, we can immediately abort the process and declare the source code to be

```
1   (A I_Q)
2  A1: (A m1)
3  A2: (A m1m2)
4  A3: (A (B m1m2) (C m1m2) (D m1m2))
5  A4: (A (B m1m2)*m6 (C m1m2)*m6 (D m1m2)*m6)
6  A5: (A (B m1m2)*m6m7 (C m1m2)*m6m7 (D m1m2)*m6m7)
7  B1: (A (B m1m2m3)*m6m7 (C m1m2)*m6m7 (D m1m2)*m6m7)
8  B2: (A (B m1m2m3m4)*m6m7 (C m1m2)*m6m7 (D m1m2)*m6m7)
9  B!: (A m1m2m3m4m6m7 (C m1m2)*m6m7 (D m1m2)*m6m7)
10 C1: (A m1m2m3m4m6m7 (C m1m2m5)*m6m7 (D m1m2)*m6m7)
11 C!: (A m1m2m3m4m6m7 m1m2m5m6m7 (D m1m2)*m6m7)
12 D1: (A m1m2m3m4m6m7 m1m2m5m6m7 (D m1m2I_Q)*m6m7)
13 D!: (A m1m2m3m4m6m7 m1m2m5m6m7 m1m2m6m7)
14 A!: m1m2m3m4m6m7 m1m2m5m6m7 m1m2m6m7
```

Figure 3.13: Assessment string sequence of the AST in Figure 4.5

```
1 if(itIsRaining){
2   ds.m8();
3 }
4 while(!userPressedCancel){
5   ds.m9(); ds.m10();
6 }
7 ds.m11();
```

Figure 3.14: Example of client source code containing a COND and a LOOP

incompatible. This is the reason why the algorithm was chosen to traverse the tree in a breadth-first way.

At this point we already applied all children of A and the next step would be to try to reduce A to a set of matrices. But we cannot reduce A since this block has inner blocks that need to be reduced before A.

Therefore we start the same process with block B. Now we should apply B1 to the contents of B. It is important to notice that this time we are applying m_3 to the parent node B. Therefore, m_3 should multiply the current matrix of B, not its pending matrix. The same process is repeated with the child B2. Now, at line `B2`, all the children of B were applied to this block and, since B does not have any inner block, we can calculate the linear transformation that is equivalent to B (second rule of Table 4.3) and eliminate this block (line `B!`).

The same process is repeated for blocks C and D until they are both reduced to matrices (lines `C!` and `D!`). Finally, we can reduce the A block and have as a final result that the possible method sequences are $m_1m_2m_3m_4m_6m_7$, $m_1m_2m_5m_6m_7$, and $m_1m_2m_6m_7$.

`LOOP` subtrees should follow a similar process, but, as the number of iterations can't be determined in the general situation, we are forced to iterate the contents of the `LOOP` subtree, collect all possible outcomes and analyze them.

As the grammar in Figure 3.10 shows, the structure of a `LOOP` assessment string is an identification (randomly generated), a series of elements (let us call it the 'before loop section'), a colon sign, another series of elements (let us call it the 'iteration section'), and an optional pending multiplication. In other words, we could represent the structure of the assessment string of a loop as: `<identification (before loop section): (iteration section)> *(pending multiplication)`.

When applying a `LOOP` to a parent, all the contents of the parent will be placed in the before loop section, and the iteration section will be initialized with I_Q (fourth rule of Table 4.2). Just as the `COND` block, when a LTM is applied to a `LOOP`, the LTM will change the pending multiplication section. The iteration section behaves as the contents of the `COND` block, receiving the LTMs within the block. `LOOP` reduction can also be done only when its contents are only LTMs. Unlike `COND` subtrees, `LOOP` subtrees cannot be reduced symbolically, but only numerically, since we need to collect the possible outcomes of iterations.

Let us consider the source code in Figure 3.14. The equivalent reduced AST is in Figure 3.15 and the sequence of assessment strings is in Figure 3.16.

At this point, how the sequence of strings in Figure 3.16 is generated should be clear, except for reduction

```
1  -- (BLOCK E
2  E1      (COND
3  --          (BLOCK F
4  F1              (METHOD_CALL DS m8)
5  --          )
6  --          (BLOCK G
7  G1              IDENTITY
8  --          )
9  --      )
10 E2      (LOOP
11 --          (BLOCK H
12 H1              (METHOD_CALL DS m9)
13 H2              (METHOD_CALL DS m10)
14 --          )
15 --      )
16 E3      (METHOD_CALL DS m11)
17 -- )
```

Figure 3.15: Reduced AST of the source code in Figure 3.14

```
1   (E I_Q)
2  E1: (E (F I_Q) (G I_Q))
3  E2: (E <H (F I_Q) (G I_Q) : I_Q>)
4  E3: (E <H (F I_Q) (G I_Q) : I_Q>*m_11)
5  F1: (E <H (F m_8) (G I_Q) : I_Q>*m_11)
6  F!: (E <H m_8 (G I_Q) : I_Q>*m_11)
7  G1: (E <H m_8 (G I_Q) : I_Q>*m_11)
8  G!: (E <H m_8 I_Q : I_Q>*m_11)
9  H1: (E <H m_8 I_Q : m_9>*m_11)
10 H2: (E <H m_8 I_Q : m_9m_10>*m_11)
11 H!: (E M'(H))
12 E!: M'(H)
```

Figure 3.16: Assessment string sequence of the AST in Figure 3.15

of the H block (line `H!`). $M'(H)$ means the collection of all matrices that can be generated by the H subtree. Given that a `LOOP` assessment string has the structure `<id b1 b2 ... bx : i1 i2 ... iy>*p`, the possible outcomes should be calculated based on the expressions

$$\begin{matrix} b_1 i_1^n p & b_1 i_2^n p & \cdots & b_1 i_y^n p \\ b_2 i_1^n p & b_2 i_2^n p & \cdots & b_2 i_y^n p \\ \vdots & \vdots & \ddots & \vdots \\ b_x i_1^n p & b_x i_2^n p & \cdots & b_x i_y^n p \end{matrix} \tag{3.7}$$

Where $n \in \mathbb{N}$. We start with $n = 0$. When one of these expressions generates an LTM that was already generated we discard this expression. We increment n until we discarded all expressions and finally we have M' for the loop.

In the example of Figure 3.16, line `H2`, to generate $M'(H)$ we need to calculate all possible outcomes of expressions $m_8(m_9m_{10})^n m_{11}$ and $I_Q(m_9m_{10})^n m_{11}$.

The output of a `LOOP` can only be said to be incompatible for sure if the only possible outcome is 0_Q. It is expected then, that the presence of a `LOOP` will cause the compatibility assessment to be less precise. The collection of possible outcomes will generate 'vague' matrices that are represented with a leading question mark (grammar rule aVagueLTM). All operations with vague matrices are identical to operations with ordinary matrices, but the resulting matrix is also marked as being vague. I.e., the uncertainty added by a `LOOP` subtree will be propagated.

Table 3.5: Descriptions of methods in the DM service

Method	Description
m_1	Upload file chunk
m_2	Start data mining
m_3	Download data mining result
m_4	Erase file (and cancel data mining)
m_5	Erase data mining result
m_6	Cancel data mining

Regarding compatibility assessment, the difference between a vague matrix and an ordinary matrix is that finding a vague 0_Q as an outcome of an element does not guarantee that an incompatibility was found. Actually, having one of its outcomes as a vague 0_Q will only determine that a certain element is incompatible if all other outcomes are also vague 0_Q matrices.

3.6 Example - A data mining service

To demonstrate the process, let us consider a data mining service called DM.

3.6.1 First version of the data mining service: DM0.1

A first version of DM will be called $DM0.1$. In this version, data mining is performed over data provided by the client. Then, service utilization is a two-step process. In the first step, a client must upload a big file containing all data to data mine to the server, and in the second step the client can request one data mining to the service.

The two FSMs of this service are depicted in Figure 5.1. This state machine operates over two state variables s_α and s_β.

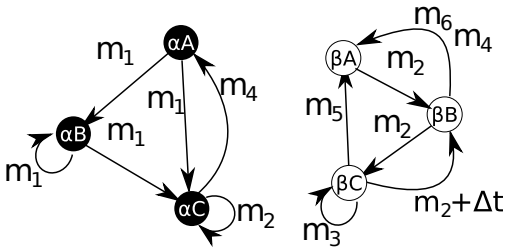

Figure 3.17: A composite state machine of $DM0.1$, the first version of the DM service

This service begins in the composite state $(s_{\alpha,A}, s_{\beta,A})$. A client is then supposed to call the method m_1 several times to upload a file to the server. After the file is completely transferred, the client is able to call the method m_2 that will start a data mining process. This method does not block waiting for the data mining to finish, since this process may take a long time to complete. Instead, the method m_2 returns immediately after being called and starts a thread in the service-side. This thread is invisible to the client, but affects the state machine β, causing the transition $s_{\beta,B} \rightarrow s_{\beta,C}$ asynchronously, which is marked with '$m_2 + \Delta t$' in Figure 5.1.

Tables 5.1 and 3.6 summarize the service contract. Table 5.1 shows the meanings of each of the methods, while Table 3.6 shows the meanings of each state. The method m_4 will cancel a data mining if there is one in process. So the transition $s_{\beta,B} \rightarrow s_{\beta,A}$ may be executed by either m_4 or m_6.

The service may have separated buffers to store the file and the data mining results. Then, method m_3 can be called to retrieve a previous data mining result even when a new file is being uploaded. Let's represent the states as

Table 3.6: Meanings of the states in $DM0.1$

State	Meaning
αA	No data file
αB	Data file partially received
αC	Data file completely received
βA	No data mining result
βB	Data mining in process
βC	Data mining result available

$$s_{\alpha,A} = s_{\beta,A} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$
$$s_{\alpha,B} = s_{\beta,B} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$
$$s_{\alpha,C} = s_{\beta,C} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

Also, let's represent the method transformations for the state machine α as

$$m_{1,\alpha} = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \qquad m_{2,\alpha} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
$$m_{4,\alpha} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

and the transformations for the state machine β as

$$m_{2,\beta} = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \qquad m_{3,\beta} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$
$$m_{4,\beta} = m_{6,\beta} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad m_{5,\beta} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

For instance, the operator $m_{2,\beta}$ means that the operation is capable of causing the transitions $s_{\beta,A} \to s_{\beta,B}$, $s_{\beta,A} \to s_{\beta,C}$, $s_{\beta,C} \to s_{\beta,B}$, and $s_{\beta,C} \to s_{\beta,C}$. Transitions $s_{\beta,A} \to s_{\beta,C}$ and $s_{\beta,C} \to s_{\beta,C}$ use the state $s_{\beta,B}$ as intermediate and depend on the help of the internal thread. Methods that do not cause transitions in a state will be represented by the identity matrix:

$$m_{1,\beta} = m_{3,\alpha} = m_{5,\alpha} = m_{6,\alpha} = I_3$$

In some cases the transition will depend on conditions verified by the method. This is the case of m_1 operating over the state $s_{\alpha,B}$, for instance. Whether the transition will be to $s_{\alpha,B}$ itself or to $s_{\alpha,C}$ will depend on whether the file transfer terminated or not.

Let us consider the client source code in Figure 3.18. The equivalent reduced AST structure is shown in Figure 3.19, and the sequence of assessment strings is shown in Figure 3.20.

Let the linear transformations for each of the three outcomes (last line in Figure 3.20) be m_a, m_b, and m_c, respectively. Using $m_{1,\alpha}$, $m_{2,\alpha}$, $m_{4,\alpha}$, $m_{1,\beta}$, $m_{2,\beta}$, and $m_{4,\beta}$ as defined before, we have as a result,

$$m_{a,\alpha} = 0_{3,3}$$
$$m_{b,\alpha} = m_{1,\alpha}$$
$$m_{c,\alpha} = m_{1,\alpha}$$
$$m_{a,\beta} = m_{2,\beta}$$
$$m_{b,\beta} = I_3$$
$$m_{c,\beta} = I_3$$

```
1  ds.m1();
2  if(/*condition*/) {
3    ds.m1(); ds.m1();
4    if(/*condition*/) { ds.m2(); }
5    else { ds.m4(); }
6  }
7  else {
8    print(ds.m1() + someNumber);
9  }
10 ds.m1();
```

Figure 3.18: Example of IF blocks in client source code

```
1  -- (BLOCK A
2  A1      (METHOD_CALL m1)
3  A2      (COND
4  --          (BLOCK B
5  B1              (METHOD_CALL m1)
6  B2              (METHOD_CALL m1)
7  B3              (COND
8  --                  (BLOCK C
9  C1                      (METHOD_CALL m2)
10 --                  )
11 --                  (BLOCK D
12 D1                      (METHOD_CALL m4)
13 --                  )
14 --              )
15 --          )
16 --          (BLOCK E
17 E1              (METHOD_CALL m1)
18 --          )
19 --      )
20 A3      (METHOD_CALL m1)
21 -- )
```

Figure 3.19: Reduced AST extracted from the client code in Figure 3.18

```
1        (A $I_Q$)
2  A1:   (A $m_1$)
3  A2:   (A (B $m_1$) (E $m_1$))
4  A3:   (A (B $m_1$)*$m_1$ (E $m_1$)*$m_1$)
5  B1:   (A (B $m_1m_1$)*$m_1$ (E $m_1$)*$m_1$)
6  B2:   (A (B $m_1m_1m_1$)*$m_1$ (E $m_1$)*$m_1$)
7  B3:   (A (B (C $m_1m_1m_1$) (D $m_1m_1m_1$))*$m_1$ (E $m_1$)*$m_1$)
8  C1:   (A (B (C $m_1m_1m_1m_2$) (D $m_1m_1m_1$))*$m_1$ (E $m_1$)*$m_1$)
9  C!:   (A (B $m_1m_1m_1m_2$ (D $m_1m_1m_1$))*$m_1$ (E $m_1$)*$m_1$)
10 D1:   (A (B $m_1m_1m_1m_2$ (D $m_1m_1m_1m_4$))*$m_1$ (E $m_1$)*$m_1$)
11 D!:   (A (B $m_1m_1m_1m_2$ $m_1m_1m_1m_4$)*$m_1$ (E $m_1$)*$m_1$)
12 B!:   (A $m_1m_1m_1m_2m_1$ $m_1m_1m_1m_4m_1$ (E $m_1$)*$m_1$)
13 E1:   (A $m_1m_1m_1m_2m_1$ $m_1m_1m_1m_4m_1$ (E $m_1m_1$)*m1)
14 E!:   (A $m_1m_1m_1m_2m_1$ $m_1m_1m_1m_4m_1$ $m_1m_1m_1$)
15 A!:   $m_1m_1m_1m_2m_1$ $m_1m_1m_1m_4m_1$ $m_1m_1m_1$
```

Figure 3.20: Assessment strings sequence of the reduced AST in Figure 3.19

Table 3.7: Meanings of states in $DM0.2$

State	Meaning
αA	No data file
αB	Data file partially received
αC	Data file completely received
βA	Processor idle
βB	Data mining in process
γA	Empty data mining result queue
γB	Non-empty data mining result queue

Therefore, there is an error in the sequence m_a, in the FSM α. During the process illustrated in Figure 3.20, the 0_3 is found during the execution of the line `B!`, when we multiply $m_1 \cdot m_1 \cdot m_1 \cdot m_2$ by m_1 (originated from the last line of Figure 3.18, and that was reduced to the line A3 of Figure 3.19).

3.6.2 Second version of the data mining service: DM0.2

Let us analyze now an example of new version $DM0.2$ of the same service, that has a different state transition diagram as depicted in Figure 5.9. Table 3.7 has the meanings of each state in $DM0.2$. In $DM0.2$, all methods have the same signatures and the same meanings as in $DM0.1$ (Table 5.1), but now there is a new state machine called γ. The initial state is $(s_{\alpha,A}, s_{\beta,A}, s_{\gamma,A})$, and this new specification of the service requires the data mining to have a FIFO queue to store results of data mining requests. This queue is manageable by the client using the methods m_3 and m_5.

Let's assume that the designers of $DM0.2$ considered the state $s_{\beta,C}$ unnecessary since a processor may be idle or doing data mining independently on the existence or not of data mining results to download. The transition $s_{\gamma,A} \rightarrow s_{\gamma,B}$ can be caused by m_2 but not instantly. Instead, the transition is done by the same hidden thread that causes the transition $s_{\beta,B} \rightarrow s_{\beta,A}$, which may happen long after the method m_2 finished (marked with '$m_2 + \Delta t$' in Figure 5.9).

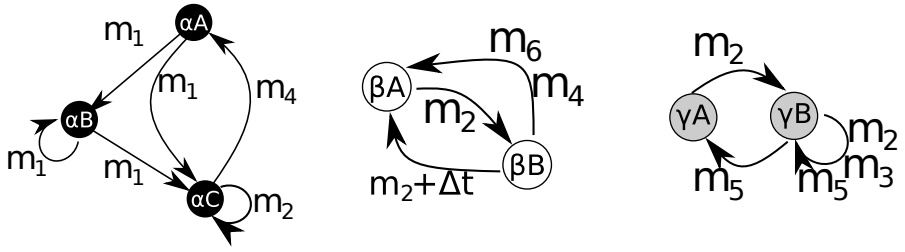

Figure 3.21: FSMs of $DM0.2$, a new version of DM service

So despite of having the same method signatures, and no extra or missing methods, a client designed to work with $DM0.2$ may be incompatible with $DM0.1$. So $DM0.2$ is not backward compatible.

For instance, let's consider a sequence $p = m_1^x \cdot m_2^y \cdot m_3^y \cdot m_5^y$ where x is the number of chunks of the file to be uploaded and y is the number of data mining executions. The sequence p is legal in $DM0.2$ and will generate the transformations:

$$p_\alpha = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \qquad p_\beta = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \qquad p_\gamma = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Matrices p_α, p_β, and p_γ are all different from 0_Q, therefore p is compatible with $DM0.2$. But verifications using $DM0.1$ will find that $m_5^y = 0_{N,N}$ for any $y > 1$, which means the sequence p is only valid in $DM0.1$ for $y = 1$. I.e., DM is not backward compatible from $DM0.2$ to $DM0.1$.

On the other hand, a client made to utilize $DM0.1$ will be compatible with $DM0.2$, since $DM0.2$ has less restrictions about the methods m_3 and m_5. The result of a client made for $DM0.1$ utilizing $DM0.2$ is the wastage of the result queue, but the design of the interface guaranties forward compatibility from $DM0.1$.

3.7 Conclusion

We have shown that version compatibility analysis can be enhanced if we expand the functional properties of service contracts to contain FSM specifications. The method is robust to be applied to complex client source code since we eliminate all blocks that are not relevant to the remote service and we can apply the method to any isolated sequence of method calls. On the other hand, every time we apply an LTM m_N to the current equivalent LTM m', it is possible that we discard some non-zero entries of m'. If, at run time, the discarded entry is chosen by the service to be the next transition, a client may not be able to call m_N. Although this does not characterize an incompatibility, it is a potential risk at run time.

All incompatibilities found by our method are actual illegal sequences of method calls (no false positives), assuming that a remote service is not utilized concurrently by more than one client-side process. Nevertheless, our method does not guarantee that a client is free from incompatibilities. This happens because, depending on how the client was built, client behavior may be only determined at execution time.

False negatives is precisely the weakness of our method. I.e., not finding incompatibilities does not completely prevent client source code to try from calling service methods in illegal sequences. The only way to avoid false negatives is to have client source codes that are clear about how it intends to utilize the remote service. As a consequence, the reliability of our process is heavily dependent on how much a programmer understands how the compatibility verification works. We will address these limitations in our future research.

Chapter 4

Client handling of service-side FSMs and versioning

Abstract

In a service-oriented environment, service contracts play an important role to provide interoperation between clients and services. As contracts are the de facto insulation layer between clients and services, we argue that contracts should not only present specifications of method formats, but also pre and post-conditions that could provide more sophisticated client-service interactions. We have proposed that service contracts should contain specifications of service-side finite state machines (FSM). The immediate benefits of pre and post-conditions in distributed services are less defensive source codes in both sides, and avoidance to execute remote service with invalid parameters, which translates into rationalization of resources. But we argue that FSMs can also be used to provide client-service synchronization and advanced compatibility assessment, if the client source code is specially prepared to support these features. In other words, if the client source code contains contructs that are specific to deal with distributed services that follow this format. In this paper, we provide a framework for such extensions and present details about our implementation.

4.1 Introduction

Distributed computing is getting more popular and complex. The momentum of interconnection technology enables distribution to reach more clients, locality to be virtualized, and new markets to emerge.

In service-oriented architectures, contracts have a very important role. Not only they specify how services should be called but, if correctly designed, also allows for independent evolution of clients and services. Service contract design should therefore be considered a key component to control service project life cycles. A contract may be utilized as a standard implemented by several vendors.

Externalizing state information was the motivation to create the WSRF [20], which deprecated the older Open Grid Services Infrastructure (OGSI) specification. In a distributed environment, resources are shared and therefore clients need interact with service-side states to manage resource usage (allocation, release, service instance life-cycle management, etc).

In a previous work [57], we introduced an algorithm to test compatibility between a client source code and a service contract, based on FSMs on the service side. FSMs in service-side define legal sequences of method calls. So we could check if a client is compatible with a certain service by checking if the sequences of method calls that the client may produce are in fact legal sequences according to the contract.

In this paper we extend that idea. Here we propose a set of client-side programming language constructs to interact with service-side FSMs. We will also analyze how compatibility analysis can be enhanced in presence of these constructs. For the best of our knowledge, there are no previous methods to verify client compliance when states in the service are referenced explicitly. As we will see in Section 4.5, each reference to a service-side FSM may reduce uncertainty in compatibility assessment.

We start the paper by analyzing the related work in Section 4.2. As we will show, for the best of our knowledge, there is little research trying to analyze client source code and most of concern is on interaction between web services. Next, in Section 4.3, we will define which distributed services we target with our algorithm. Before we introduce the special client-side constructs for FSM interaction, we will introduce the basic algorithm to client source code verification in Section 4.4. Section 4.5 contains the contribution of this

paper, and will present the client side constructs that reference service-side FSMs. Section 4.6 will compare our proposal with other methods and, finally, Section 4.7 will conclude this paper.

4.2 Related work

The questions about distributed service versioning were identified by Vinoski [76]. A specific analysis for versioning in SOA focusing on decoupling of service and clients was presented by Lublisky [50]. Both papers analyze the problem from the perspective of the mechanisms responsible for service and client decoupling, but no attention was given to the role of clients, or more specifically, the roles of client software developers, to detect compatibility problems.

The importance of method invocation order in grids and clouds has its roots in their objectives. Ultimately, grids and clouds aim to virtualize remote resources, and offer them to clients as remote services. Therefore, remote resources should be allocated, reserved, and released in a certain order. Also, system distribution requires transactions to be created, which translates into additional restrictions on method execution order. For instance, in a typical transaction, 'commit' must be the last method called.

Service states were recognized as an important feature of distributed systems. According to an informational document [30] about OGSA, published by the GGF, the WSRF standard [20] was created by the OASIS consortium [1] to meet the requirements for state representation and manipulation in grid services.

The problem of having derivation from an original service was investigated by Ponnekanti and Fox [68]. This paper proposes a model to deal with compatibilities between web services from distinct vendors that are originated from the same root web service. In contrast, here we deal with problems arisen from the version progression of the same service when a single team is responsible for the evolution of the service specification. Besides the differences, in both cases detecting service compatibilities was identified as an important factor to the maintenance of services. Their compatibility model is based on WSDL comparisons but here we propose a new and more detailed service contract to identify changes in service implementation internals, that could present impacts in service compatibility.

Weinreich et al. [79] propose a model in which products are defined as a set of clients and services, and both may evolve independently. In our model there is a separation between service implementation and service specification (service interface), which allows for multiple providers to offer alternative implementations.

In their proposal, the concept of compatibility is based on comparisons between different versions of the same service interface. This simplicity is motivated by the control offered by the concept of 'product', in which both clients and services participate. In this model, version numbers are used by clients to choose between utilizing the latest version of the service that follows the same compatible interface or to select a specific minor implementation.

In our model we do not specify whether a client wants to follow the latest version of a service or not, since we focus on compatibility detection based on non-functional aspects of services and clients. We do not propose strategies to service evolution and version branches as defined by Weinreich et al., since we assume that multiple service implementations may be provided by a number of development teams. In our model we assume that the number of client applications can be too large for compatibility analysis to be conducted manually.

A service change model was proposed by Leitner et al. [49], but the proposed model only comprehends functional characteristics of services, while here we precisely try to include non-functional characteristics or internal behaviors of remote services that are relevant for compatibility analysis.

Versioning models for SOA generally discuss versioning of Web Services but, according to Vogels [77], Web Services should not be understood as distributed objects. Therefore we argue that versioning in the context of distributed objects (as an abstraction that can be mapped to Web Services or other underlying distribution technology) may support more robust compatibility detections embedded in the very programming tools utilized to generate and utilize the distributed objects.

Although we argue that adding aspects others than method signatures can enhance compatibility evaluation, we do not try to propose a concrete infrastructure to support how contract specifications should be shared. A proposal for such a infrastructure was presented in [9]. They evaluated how non-functional properties could be implemented utilizing web services .

Compatibility in terms of service contracts is also analyzed in [63]. But their focus is on message exchange sequences and service termination recognition. Instead of checking client routines, they analyze client contracts.

[1] OASIS consortium website: http://www.oasis-open.org/

Finally, our approach to extend an OOP language and to create a DSL to support evolvable systems was also used by Lee et al. [48]. While here we propose to externalize aspects of service behavior that influence the way clients should be built (and ultimately can be a factor that causes client-service compatibility issues), Lee et al. propose an extension of the Java programming language grammar to allow for automated manipulation of the messages exchanged between client and service via pattern matching.

4.3 Distributed services model

Here we assume that a service is represented by a contract that may evolve along time. A contract $C(M, F)$ is composed of a list M of method prototypes, and a set F of P FSMs which express relationships of precedence between methods, when such relationships exist.

As here we focus on the client side, functional unities of service will be understood as methods, but all argumentation is also valid if "method" is "replaced" by message.

Each FSM $\phi(S, S_0, T)$ is composed by a list of states S, a set of initial states $S_0 \subseteq S$, and a list of transitions T. Each transition t is associated with zero or more methods of M. Then, we denote $t(s_a \rightarrow s_b, X \subseteq M)$, where $s_a \rightarrow s_b$ means a transition from the state s_a to the state s_b, and X is the set of methods associated with t. In short:

$$M = \{m_1, m_2, \ldots, m_N\} \tag{4.1}$$
$$F = \{\phi_1, \phi_2, \ldots, \phi_P\} \tag{4.2}$$
$$S = \{s_1, s_2, \ldots, s_Q\} \tag{4.3}$$
$$T = \{t_1, t_2, \ldots, t_R\} \tag{4.4}$$

A transition $s_a \rightarrow s_b$ being associated with a method m means that if m is called when the current state is s_a, then either the termination of m, or an asynchronous process started by m, is able to change the state to s_b. If a method is associated with more than one transition, decision about which transition to perform will be made by the service. A state being associated with a set transitions automatically means that the method cannot be called when a state is such that none of these transitions can be performed. In other words, when the current state is not the initial state of any of the transitions the state can perform, a state cannot be executed. States that are not related to any transition are supposed to tolerate any current state.

FSMs do not change their states unless a client calls a service method, to ensure that service-side states are in accordance with the contract specification that the client expects. Each remote service behaves, then, as a non-shared object, from the client's perspective.

Nevertheless, the decisions of which state transition to execute may depend on factors others than those visible to the client, such as other clients competing to utilize a shared resource concurrently.

In our examples we assume that a remote service is presented to the client as a local object. Client source code will then utilize service methods by simply calling methods of the local object.

As for the transparency of service-side FSMs, we do not assume that the service keeps the client updated about which is the current state of each FSM, since the process described here is a validation that can be performed during the process to build a client. However, as we will discuss later, compatibility assessment may be inaccurate during client build. Therefore, there are benefits to satisfactory client-server cooperation if the client knows which are the states in the service side during the execution of the distributed system.

If it is the case that the client is aware of the states in service-side, the same method we introduce here can be utilized during client execution to detect a service behavior that differs from the expected by the client.

4.4 Background information - code compatibility assessment

Before introducing how explicit FSM references in the client side can influence client-contract compatibility assessment, we need to introduce a simpler scenario: compatibility assessment between a client that simply calls methods of the contract. For a detailed discussion about this process, we recommend referring our previous paper [57]. Here we just present the outline of the algorithm for clarity.

Consider the client source code of Figure 4.1. A possible equivalent AST is shown in Figure 4.2. The object ds is a reference to a remote service. This AST contains elements that do not interest us to analyze client-service compatibility. For instance, although the variable i is passed as a parameter to a service method, we will ignore the declaration of this variable when we check compatibility.

```
1 int i = 0;
2 DS ds = serviceFactory(parameters);
3 ds.m1(i);
4 String s = ds.m2();
5 ds.m3();
```

Figure 4.1: Simple client source code in Java.

```
1 (VARIABLE_DECLARATION i int 0)
2 (VARIABLE_DECLARATION ds DS (METHOD_CALL serviceFactory (ARGUMENTS ↩
     parameters)))
3 (METHOD_CALL ds m1 (ARGUMENTS i))
4 (VARIABLE_DECLARATION s String (METHOD_CALL ds m2))
5 (METHOD_CALL ds m3)
```

Figure 4.2: AST of the source code in Figure 4.1

```
1 (INITIALIZATION DS)
2 (METHOD_CALL DS m1 (ARGUMENTS int))
3 (METHOD_CALL DS m2)
4 (METHOD_CALL DS m3)
```

Figure 4.3: A reduced AST derived from the one shown in Figure 4.2

Therefore, we will reduce the AST of Figure 4.2 to the AST shown in Figure 4.3. To obtain reduced ASTs, we will:

- ignore the statements that are unrelated to the remote service, or that do not control execution flow (loops, etc),
- replace initializations of remote service references with an `INITIALIZATION` node. As we do not base our method on any specific service technology, how references to remote services are actually created (and therefore how creation is detected) is out of the scope of this paper.
- replace object references with remote service types,
- replace method calls with signature-like subtrees, and
- extract method call subtrees from other statements (in our example, we extracted `(METHOD_CALL DS m2)` from `(VARIABLE_DECLARATION s String (METHOD_CALL ds m2))`)

Let us consider an FSM $\phi(S, S_0, T)$ with Q states as in (4.3). Each state will be represented by a unit vector in a Q-dimensional space. The set of initial states S_0 will be represented by a matrix in which the diagonal has a value 1 in the positions equivalent to each of the initial states, and all other entries contain zeros. For instance, for an FSM in which $S = \{s_1, s_2, s_3, s_4, s_5\}$ and the initial states are s_1 and s_3, S_0 will be translated into $S_0 = diag(1, 0, 1, 0, 0)$. In the extreme case in which $S_0 = S$, the equivalent matrix will be the identity matrix I_Q.

Each method will be represented by an LTM from $\mathbb{N}^Q$ to $\mathbb{N}^Q$, that will have a number 1 placed at each state transition the method is able to perform, according with the following map:

$$\begin{bmatrix} s_1 \to s_1 & s_1 \to s_2 & \cdots & s_1 \to s_Q \\ s_2 \to s_1 & s_2 \to s_2 & \cdots & s_2 \to s_Q \\ \vdots & \vdots & \ddots & \vdots \\ s_Q \to s_1 & s_Q \to s_2 & \cdots & s_Q \to s_Q \end{bmatrix} \tag{4.5}$$

For instance, let's consider a service contract in which there is one FSM with two states s_1 and s_2. A method that could cause the transitions $s_1 \to s_1$ or $s_1 \to s_2$ will be represented by $\left(\begin{smallmatrix} 1 & 1 \\ 0 & 0 \end{smallmatrix}\right)$. Methods that are not related to any state transition will be represented by the identity matrix I_Q.

Table 4.1: Assessment string initialization

Outermost element	Initialization
(BLOCK A)	(A I_Q)
(COND (BLOCK A) (BLOCK B) ...)	(A I_Q) (B I_Q) ...
(LOOP (BLOCK A))	<A I_Q : I_Q >

A sequence of method calls is equivalent to applying a composite linear transformation. Then, for a simple sequence of w method calls, where $m_{(i)}$ represents the i-th method called, the equivalent linear transformation m' is simply the product of the linear transformations of the methods called.

The value of each entry of a composite linear transformation represents the number of ways each transition can be made. For example, if the composite linear transformation $m_1 \cdot m_2 \cdot m_3$ is capable of leading s_1 to s_2 using the paths $s_1 \xrightarrow{m_1} s_3 \xrightarrow{m_2} s_3 \xrightarrow{m_3} s_2$ and $s_1 \xrightarrow{m_1} s_4 \xrightarrow{m_2} s_5 \xrightarrow{m_3} s_2$, then the entry that corresponds to the position $s_1 \rightarrow s_2$ will have a value 2.

Therefore, a composite linear transformation equals to a zero matrix 0_Q means that the sequence of method calls that generated the linear transformation cannot lead any state to any other state. In other words, the sequence of method calls is invalid for the contract. The point, in the successive multiplications, in which the matrix becomes 0_Q is where a remote service method is called over a state that it cannot accept as an input state. In other words, the statement that causes a matrix to become 0_Q is the statement responsible for the incompatibility.

The application of an `INITIALIZATION` after the execution of some methods will reset the FSM to one of the initial states. The consequence for the calculation of the equivalent linear transformation is that the current LTM will be simply replaced by S_0. We will translate the application of a subtree `(↩ INITIALIZATION X)` over the current LTM m into the operation

$$m \cdot 0_Q + S_0 \equiv m \odot S_0 \tag{4.6}$$

We will utilize the symbol '$\odot$' to represent this operation, and will consider this operation atomic. I.e., we will not interpret the 0_Q obtained before the addition as an incompatibility.

In reduced AST, all loops (for, while, do-while, etc) will be represented by the imaginary node `LOOP` and all conditionals (if, if-else, switch, etc) will be replaced by the imaginary node `COND`. If a conditional has no default block (for example, an `if` without an `else`), we will add a default block containing the identity matrix I_Q, which will make calculations simpler. `BLOCK` is an auxiliary imaginary node that groups statements. Each block will have a unique identification that is sequentially created to simplify the step 3, as we will see later. A statement may be `LOOP`, `COND`, `METHOD_CALL`, or `IDENTITY`. A loop must have exactly one block, while a conditional may have two or more blocks [2].

After the reduced AST was created, we will utilize a tree data structure to find all equivalent LTMs that are equivalent to the interaction between client and service. Again, what we search for is a zero matrix, which represents an incompatibility. We will represent the tree data structure as an AST, which we will call assessment string. The algorithm has three steps (1) initialization, (2) application of each child, and (3) reduction to a set of LTMs.

(1) **Initialization** is how the assessment string is created. We take the outermost element (a `BLOCK`, a `COND`, or a `LOOP`), and create the string according to Table 4.1.

(2) **Application of each child** combines a block A with its children A1, A2, ... according to Table 4.2. When all children of A were applied, and A does not have any inner blocks to be reduced, we can reduce A to a set of LTMs, which is the next step.

(3) **Reduction to a set of LTMs** utilizes Table 4.3. The meaning of $M'(A)$ will be explained later.

To clarify the meanings and usage of Tables 4.1, 4.2, and 4.3, let us consider the reduced AST in Figure 4.5, derived form the source code in Figure 4.4. In Figure 4.5, block are identifiers as A, B, C, D, and E. The column on the left represents children identification. The symbol '`--`' means 'no child in this line', while letters are the parent name and numbers are the child order. For instance, `A2` means 'the second child of the block A'. The `IDENTITY` node was added to the reduced AST (line `D1`) since the conditional does not have an `else` block.

[2] Multiple blocks may happen, for instance, in successive if-elseif-elseif-..., or in switch blocks

Table 4.2: Application of a child node to its parent node

Parent	Child	Result
(A $A1$ $A2$ ...)	(INITIALIZATION X)	(A $A1 \odot S_0$ $A2 \odot S_0$ ...)
(A $A1$ $A2$ ...)	(METHOD_CALL X m)	(A $A1 \cdot m$ $A2 \cdot m$ ...)
(A $A1$ $A2$ ...)	(COND (BLOCK X) (BLOCK Y) ...)	(A (X $A1$ $A2$ ...) (Y $A1$ $A2$ ...) ...)
(A $A1$ $A2$ ...)	(LOOP (BLOCK X))	(A <X $A1$ $A2$ ... : I_Q >)
<A $A1$ $A2$ ... : $B1$ $B2$ ... >	(INITIALIZATION X)	<A $A1$ $A2$ ... : $B1 \odot S_0$ $B2 \odot S_0$ ... >
<A $A1$ $A2$ ... : $B1$ $B2$ ... >	(METHOD_CALL X m)	<A $A1$ $A2$ ... : $B1 \cdot m$ $B2 \cdot m$ ... >
<A $A1$ $A2$ ... : $B1$ $B2$ ... >	(COND (BLOCK X) (BLOCK Y) ...)	<A $A1$ $A2$ ... : (X $B1$ $B2$ ...) (Y $B1$ $B2$ ...) ... >
<A $A1$ $A2$ ... : $B1$ $B2$ ... >	(LOOP (BLOCK X))	<A $A1$ $A2$ ... : <X $B1$ $B2$ ... : I_Q >>

Table 4.3: Reduction to LTMs

Element	Reduction
(A m_1 m_2 ...)	m_1 m_2 ...
(A m_1 m_2 ...) * m_x	$m_1 m_x$ $m_2 m_x$ ...
<A b_1 b_2 ... : i_1 i_2 ... >* p	$M'(A)$ = all possible outcomes of equations (7)

```
1  ds.m1(); ds.m2();
2  if(theDayIsSunny) {
3    ds.m3();
4    ds.m4();
5  }
6  elseif(itIsSnowing) {
7    ds.m5();
8    while(!userPressedCancel) {
9      ds.m8(); ds.m9();
10   }
11 }
12 ds.m6(); ds = new DS(); ds.m7();
```

Figure 4.4: Example of client source code containing a conditional

Figure 4.6 shows the sequence of values of the assessment string. The column on the left shows child application or block reduction. Child application is marked by a letter followed by a number (e.g.: `B1`), while block reduction is marked by the block identifier followed by an exclamation mark (e.g.: `C!`).

We start with the outermost block A and initialize it with I_Q (first line in Figure 4.5). Next we apply each of the children of this block. Application of a matrix to a block will distribute the multiplication to each of the elements inside of the bock (second rule of Table 4.2). In the example, we apply the child `A1` to the first line and obtain the line `A1` in Figure 4.5. We repeat the same process and apply the line A2 in Figure 4.5 to obtain the line `A2` in Figure 4.5. In line `A3` we need to apply a `COND` child. The effect will be the creation of three inner blocks with the contents of the parent (third rule of Table 4.2). Next, we apply A4 to the current string. Now we have three inner elements in A, therefore we need to multiply each of the three elements by m_6.

A multiplication of a matrix by a block will add the matrix to the pending multiplications slot of the block represented by an asterisk followed by the method name. In other words, the multiplication is marked to be executed in the future. Next, we apply the initialization of line A5. The last child of A is A6. Again,

```
1  --  (BLOCK A
2  A1       (METHOD_CALL DS m1)
3  A2       (METHOD_CALL DS m2)
4  A3       (COND
5  --           (BLOCK B
6  B1               (METHOD_CALL DS m3)
7  B2               (METHOD_CALL DS m4)
8  --           )
9  --           (BLOCK C
10 C1               (METHOD_CALL DS m5)
11 C2               (LOOP
12 --                   (BLOCK E
13 E1                       (METHOD_CALL DS m8)
14 E2                       (METHOD_CALL DS m9)
15 --                   )
16 --               )
17 --           )
18 --           (BLOCK D
19 D1               IDENTITY
20 --           )
21 --       )
22 A4       (METHOD_CALL DS m6)
23 A5       (INITIALIZATION DS)
24 A6       (METHOD_CALL DS m7)
25 --  )
```

Figure 4.5: Reduced AST of the source code in Figure 4.4

we will distribute the multiplication to the three inner blocks B, C, and D. But, this time, as the blocks already have a pending matrix to multiply, we can simply replace $m_6 \odot S_0$ with $m_6 \odot S_0 m_7$.

```
1  (A I_Q)
2  A1: (A m_1)
3  A2: (A m_1m_2)
4  A3: (A (B m_1m_2) (C m_1m_2) (D m_1m_2))
5  A4: (A (B m_1m_2)*m_6 (C m_1m_2)*m_6 (D m_1m_2)*m_6)
6  A5: (A (B m_1m_2)*m_6 ⊙ S_0 (C m_1m_2)*m_6 ⊙ S_0 (D m_1m_2)*m_6 ⊙ S_0)
7  A6: (A (B m_1m_2)*m_6 ⊙ S_0m_7 (C m_1m_2)*m_6 ⊙ S_0m_7 (D m_1m_2)*m_6 ⊙ S_0m_7)
8  B1: (A (B m_1m_2m_3)*m_6 ⊙ S_0m_7 (C m_1m_2)*m_6 ⊙ S_0m_7 (D m_1m_2)*m_6 ⊙ S_0m_7)
9  B2: (A (B m_1m_2m_3m_4)*m_6 ⊙ S_0m_7 (C m_1m_2)*m_6 ⊙ S_0m_7 (D m_1m_2)*m_6 ⊙ S_0m_7)
10 B!: (A m_1m_2m_3m_4m_6 ⊙ S_0m_7 (C m_1m_2)*m_6 ⊙ S_0m_7 (D m_1m_2)*m_6 ⊙ S_0m_7)
11 C1: (A m_1m_2m_3m_4m_6 ⊙ S_0m_7 (C m_1m_2m_5)*m_6 ⊙ S_0m_7 (D m_1m_2)*m_6 ⊙ S_0m_7)
12 C2: (A m_1m_2m_3m_4m_6 ⊙ S_0m_7 (C <E m_1m_2m_5 : I_Q>)*m_6 ⊙ S_0m_7 (D m_1m_2)*m_6 ⊙ S_0m_7)
13 E1: (A m_1m_2m_3m_4m_6 ⊙ S_0m_7 (C <E m_1m_2m_5 : m_8>)*m_6 ⊙ S_0m_7 (D m_1m_2)*m_6 ⊙ S_0m_7)
14 E2: (A m_1m_2m_3m_4m_6 ⊙ S_0m_7 (C <E m_1m_2m_5 : m_8m_9>)*m_6 ⊙ S_0m_7 (D m_1m_2)*m_6 ⊙ S_0m_7)
15 E!: (A m_1m_2m_3m_4m_6 ⊙ S_0m_7 M'(E)m_6 ⊙ S_0m_7 (D m_1m_2)*m_6 ⊙ S_0m_7)
16 C!: (A m_1m_2m_3m_4m_6 ⊙ S_0m_7 M'(E)m_6 ⊙ S_0m_7 (D m_1m_2)*m_6 ⊙ S_0m_7)
17 D1: (A m_1m_2m_3m_4m_6 ⊙ S_0m_7 M'(E)m_6 ⊙ S_0m_7 (D m_1m_2I_Q)*m_6 ⊙ S_0m_7)
18 D!: (A m_1m_2m_3m_4m_6 ⊙ S_0m_7 M'(E)m_6 ⊙ S_0m_7 m_1m_2m_6 ⊙ S_0m_7)
19 A!: m_1m_2m_3m_4m_6 ⊙ S_0m_7 M'(E)m_6 ⊙ S_0m_7 m_1m_2m_6 ⊙ S_0m_7
```

Figure 4.6: Assessment string sequence of the AST in Figure 4.5

Until this point (line `A5` in Figure 4.6) we know that we have at least three alternative method sequences, and that all method sequences will necessarily start with m_1m_2 and end with $m_6 \odot S_0m_7$. Therefore, if we find m_1m_2 or $m_6 \odot S_0m_7$ to be equal to 0_Q, we can immediately abort the process and declare the source

code to be incompatible. This is the reason why the algorithm was chosen to traverse the tree breadth-first.

At this point we already applied all children of A and the next step would be to try to reduce A to a set of matrices. But we cannot reduce A since this block has inner blocks that need to be reduced before A.

Therefore we start the same process with block B. Now we should apply B1 to the contents of B. It is important to notice that this time we are applying m_3 to the parent node B. Therefore, m_3 should multiply the current matrix of B, not its pending matrix. The same process is repeated with the child B2. Now, at line `B2`, all the children of B were applied to this block and, since B does not have any inner block, we can calculate the linear transformation that is equivalent to B (second rule of Table 4.3) and eliminate this block (line `B!`).

Application of C2 will follow the fourth line of Table 4.3. During compliance verification time, we do not know how many times a loop will be repeated. Therefore, we need to numerically calculate the possible LTMs that can be generated from repetitions of the LOOP contents. The structure of the assessment string of a loop is: `<identification (before loop section): (iteration section)> *(pending multiplication)`.

$M'(E)$ means the collection of all matrices that can be generated by the E subtree. Given that a `LOOP` assessment string has the structure `<id b1 b2 ... bx : i1 i2 ... iy>*p`, the possible outcomes should be calculated based on the expressions

$$\begin{matrix} b_1 i_1^n p & b_1 i_2^n p & \cdots & b_1 i_y^n p \\ b_2 i_1^n p & b_2 i_2^n p & \cdots & b_2 i_y^n p \\ \vdots & \vdots & \ddots & \vdots \\ b_x i_1^n p & b_x i_2^n p & \cdots & b_x i_y^n p \end{matrix} \tag{4.7}$$

Where $n \in \mathbb{N}$. We start with $n = 0$. When one of these expressions generates an LTM that was already generated we discard this expression. We increment n until we discarded all expressions and finally we have M' for the loop.

To generate $M'(E)$ we need to calculate all possible outcomes of the expressions $m_1 m_2 m_5 (m_8 m_9)^n$.

The same process is repeated for blocks C and D until they are both reduced to matrices (lines `C!` and `D!`). Finally, we can reduce the A block and have as a final result that the possible method sequences are $m_1 m_2 m_3 m_4 m_6 \odot S_0 m_7$, $M'(E) m_6 \odot S_0 m_7$, and $m_1 m_2 m_6 \odot S_0 m_7$.

4.5 Explicit FSM references in client-side

We just saw an algorithm to analyze client source code and to find compatibility problems related to service-side FSMs. So far, the way clients utilize a service does not differ from the way local objects are utilized. In general, remote service APIs are built to provide local objects that represent remote services, which feels natural for programmers. The main differences will be how this object is obtained, and the behavior during run time. For instance, a remote object may throw exceptions that are simply consequent to distribution: the interruption of a network connection, or a server process that freezes.

Nevertheless, there are cases in which client developers may want to interact with service-side FSMs explicitly. For instance, as we saw above, state transitions may happen after a service method finished, as a result of some process that is being executed in the service-side. Also, a client may want to use a service-side FSM as a semaphore.

We propose that the reduced AST could contain information about how the client wants to interact with service-side FSMs. Besides allowing for a more sophisticated cooperation between client and service, we argue that analyzing how clients utilize service-side FSMs can be used to enhance accuracy in compatibility assessment.

As we saw, our compatibility assessment process is based on finding zero LTMs. A non-zero LTM does not mean that client and service are compatible, but only that no incompatibility was found. A method call sequence may be valid according to the service contract FSMs, but the execution of client and service may lead the service to a state that the client does not want.

When the execution of a method can cause more than one transition, the context in which the service is executed (current service-side states, service-side data, data provided by the client, random variables, etc.) will determine which will be the next state. At client-service contract assessment time, these conditions cannot be determined. Therefore, the assessment process can only calculate whether there is a chance that the method call sequence will generate any valid state or not.

But, if the client specifies which state it expects the service to reach, compatibility assessment can verify whether or not the expected state is among the potentially reachable ones. The more clients and services are explicit about how these states are expected to transit from value to value, the more accurate can be a compatibility assessment.

Having an equivalent linear transformation containing several non-zero values means that the client procedure can be applied to several initial states and lead to several initial states. But it may also be a symptom that the client procedure is vague or not specific enough about under which assumptions the client procedure was written. A vague description of client expectations towards service behavior may make the client appear to be more flexible than it actually is. As a consequence, compatibility assessment processes may erroneously evaluate clients to be compatible with versions of the service that actually do not behave as expected by the clients.

Contracts also play an important role to minimize vagueness related to client source code, since uncertainty towards state transitions can easily appear if the same method is capable of making more than one state transition, depending on the circumstances in which the method was called.

Contracts that allows the same method to execute several kinds of transitions will leave state transition decision as responsibility of the service. Consequently, client-contract compatibility is hard to assess.

On the other hand, contracts that map state transitions to a single method will leave the state transition decision in the hands of clients. As a consequence, client-contract compatibility assessment usually has no ambiguities.

Here we focus on the former kind of contract, the one that makes it hard to assess client compatibility.

We saw that, in client-side, the presence of loops cause the equivalent LTM to be vague, and the presence conditionals multiply the number of equivalent LTMs. Nevertheless, programmers may want to make sure that the version assessment process will be accurate. In this way, a client application programmer can try to minimize the effects of a vague or too flexible contract.

Not only explicit references to service-side FSMs can be used to access compatibility, but they can also (1) make source code easier to comprehend and, (2) during the execution of the client, verify if the behavior of state machines is as expected.

(1) Source code comprehension can be improved as assumptions regarding the service-side states are explicit. Then the reader of the source code does not need to guess what the programmer expected from the service-side states.

(2) State behavior assessment during the execution of the client application allows for the client to catch exceptions in case the service does not execute the transition the client expected. During the client construction process, version checking processes can only verify if the structure of method calls is consistent, which ultimately means that they can generate valid states. Nevertheless, eventual misconducts of services will be not found during this process, which makes run time state behavior verifications important to ensure that client applications do not need to be built to have defensive code to respond to defects or run time errors in the service-side.

Here we will describe new language constructs in terms of imaginary nodes for clarity and to keep our discussion independent from concrete syntax.

Table 4.4 is an extension of Table 4.2, containing the application of each extension. $t(s_a \rightarrow s_b)$ stands for the LTM that contains a 1 at the $s_a \rightarrow s_b$ cell and zero in all other cells. In order to have a concise notation, we will utilize:

$$t(s_x) \equiv t(s_1 \rightarrow s_x) + t(s_2 \rightarrow s_x) + \ldots + t(s_Q \rightarrow s_x) \tag{4.8}$$

$$t(\overline{s_x}) \equiv t(s_x \rightarrow s_1) + t(s_x \rightarrow s_2) + \ldots + t(s_x \rightarrow s_Q) \tag{4.9}$$

$$t(s_x, s_y) \equiv t(s_x) + t(s_y) \tag{4.10}$$

$$t(s_x \rightarrow s_y, s_z \rightarrow s_w) \equiv t(s_x \rightarrow s_y) + t(s_z \rightarrow s_w) \tag{4.11}$$

At follows, we will present each of the constructs for the client-side programming. For each construct, we will present

- AST structure - the AST grammar rules, represented using the XML EBNF [1] [29] [3].
- an example of syntax, utilizing standard OOP. We will use Java in our examples.

[3] In short, in the EBNF notation, the symbol ^ marks a tree node, ? marks the preceding item to be optional, + marks the preceding item to be repeatable one or more times, and * marks the preceding item to be repeatable zero or more times

Table 4.4: Extension of Table 4.2 to include client-side FSM references

	Expected State Transition
Parent	(A $A1$ $A2$ ...)
Child	(METHOD_CALL (EST (TRANSITION s_1 s_2) | (TRANSITION s_3 s_4) | ...) X m)
Result	(A $A1 \cdot m \cdot *t(s_1 \to s_2, s_3 \to s_4, \ldots)$ $A2 \cdot m \cdot *t(s_1 \to s_2, s_3 \to s_4, \ldots)$ ...)
Parent	(A $A1$ $A2$ ...)
Child	(METHOD_CALL (EST (TRANSITION s_1 s_2) & (TRANSITION s_3 s_4) & ...) X m)
Result	(A $A1 \cdot m \cdot *t(s_1 \to s_2) \cdot *t(s_3 \to s_4) \cdot * \ldots$ $A2 \cdot m \cdot *t(s_1 \to s_2) \cdot *t(s_3 \to s_4) \cdot * \ldots$ ...)
Parent	<A $A1$ $A2$ ... : $B1$ $B2$ ... >
Child	(METHOD_CALL (EST (TRANSITION s_1 s_2)) X m)
Result	<A $A1$ $A2$ ... : $B1 \cdot m \cdot *t(s_1 \to s_2)$ $B2 \cdot m \cdot *t(s_1 \to s_2)$ ... >
	State Assertion / Wait for
Parent	(A $A1$ $A2$ ...)
Child	(WAIT_FOR X (STATE_LIST s_1 s_2 ...)) or (STATE_ASSERT X (STATE_LIST s_1 s_2 ...))
Result	(A $A1 \cdot *t(s_1, s_2, \ldots)$ $A2 \cdot *t(s_1, s_2, \ldots)$ ...)
Parent	<A $A1$ $A2$ ...: $B1$ $B2$ ...>
Child	(WAIT_FOR X (STATE_LIST s_1 s_2 ...)) or (STATE_ASSERT X (STATE_LIST s_1 s_2 ...))
Result	<A $A1$ $A2$ ...: $B1 \cdot *t(s_1, s_2, \ldots)$ $B2 \cdot *t(s_1, s_2, \ldots)$ ... >
	If-state
Parent	(A $A1$ $A2$ ...)
Child	(IF_STATE X s_1 s_2 ... (BLOCK $B1$) (BLOCK $B2$) ...)
Result	(A ($B1$ $A1 \cdot *t(s_1, s_2, \ldots)$ $A2 \cdot *t(s_1, s_2, \ldots)$ ...) ($B2$ $A1 \cdot *t(s_1, s_2, \ldots)$ $A2 \cdot *t(s_1, s_2, \ldots)$ ...) ...)
Parent	<A $A1$ $A2$ ...: $B1$ $B2$ ...>
Child	(IF_STATE X s_1 s_2 ... (BLOCK $C1$) (BLOCK $C2$) ...)
Result	<A $A1$ $A2$ ...: ($C1$ $B1 \cdot *t(s_1, s_2, \ldots)$ $B2 \cdot *t(s_1, s_2, \ldots)$ ...) ($C2$ $B1 \cdot *t(s_1, s_2, \ldots)$ $B2 \cdot *t(s_1, s_2, \ldots)$ ...) ...>

- an example of syntax, extending the syntax of Java. Other programming languages could have been utilized as well, and Java was chosen arbitrarily.
- the ASTs of the examples - The transformation of the example of syntax into an AST form, according to the AST structure.
- compatibility assessment - how compatibility assessment is affected by construct utilization

4.5.1 Expected State Transition

This construct can be utilized for client applications to specify which state transition they expect to happen when a certain method is called.

AST structure

```
methodCall: ^(METHOD_CALL IDENTIFIER est? ARGUMENTS)
est: ^(EST IDENTIFIER? transition (transitionOperation transition)*)
transition: ^(TRANSITION '!'? '*' IDENTIFIER) | ^(TRANSITION '!'? ↩
    IDENTIFIER '*') | ^(TRANSITION '!'? IDENTIFIER IDENTIFIER)
transitionOperation: '|' | '&'
reference: ^(REFERENCE referenceToken*)
```

Example of syntax - OOP

```
ds.m1(7, new EST(ds.s1, ds.s2).or(ds.s2, ds.s3));
ds.m2(9, new EST(ds.ANY, ds.s2).and().not(ds.s3, ds.ANY));
```

Example of syntax - DSL

```
ds.m1[s1=>s2 | s2=>s3](7);
ds.m2[*=>s2 & !s3=>*](9);
```

AST of the examples

```
(METHOD_CALL (REFERENCE ds m1) (EST (TRANSITION s1 s2) '|' (TRANSITION ↩
    s2 s3)) (ARGUMENTS 7))
(METHOD_CALL (REFERENCE ds m2) (EST (TRANSITION '*' s2) '&' (TRANSITION ↩
    '!' s3 '*')) (ARGUMENTS 9))
```

Linear transformation

The effect of this construct will be to select certain transitions from the state transition matrix of the method. We start by defining the matrix that is equivalent to the Expected State Transition (EST) subtree and then we utilize this matrix as a mask to select transitions from the method matrix.

Each TRANSITION subtree is translated into a matrix utilizing the following rules:

- A transition '`* => sx`' containing an asterisk at the left side is translated into the matrix $t(s_x)$.
- A transition '`sx => *`' containing an asterisk at the right side is translated into the matrix $t(\overline{s_x})$.
- A transition '`sx => sy`' containing no asterisk sign is translated into the matrix $t(s_x \rightarrow s_y)$.

Table 4.5: Operations used in expected transitions declarations

Operation	Cell value
$C_{M,N} = A_{M,N} \cdot * B_{M,N}$	$c_{i,j} = a_{i,j} \cdot b_{i,j}$
$C_{M,N} = A_{M,N} \wedge B_{M,N}$	$c_{i,j} = a_{i,j} \wedge b_{i,j}$
$C_{M,N} = A_{M,N} \vee B_{M,N}$	$c_{i,j} = a_{i,j} \vee b_{i,j}$
$C_{M,N} = \neg A_{M,N}$	$c_{i,j} = \neg a_{i,j}$

Operations '`|`', '`&`', and '`!`' utilizing matrices should be executed cell by cell according to the Table 4.5.

Finally, when all the TRANSITION subtrees are reduced to a single matrix m_{EST}, the linear transformation m' equivalent to the METHOD_CALL subtree m will be given by

$$m' = m \cdot * m_{EST} \tag{4.12}$$

Where '$\cdot *$', a dot product, is a simple multiplication cell by cell, as defined in Table 4.5.

For example, the ESTs in the examples above are translated into $m_1 \cdot *(t(s_1 \rightarrow s_2) \vee t(s_2 \rightarrow s_3))$ and $m_2 \cdot *(t(s_2) \wedge \neg t(\overline{s_3}))$.

4.5.2 State Assertion

ESTs are specific to a certain method call but there are situations in which a client programmer may want to make sure what is the current state at a certain source code line. This could be the case, for example, at the end of a loop. We introduce the state assertion construct for this purpose.

At design time, state assertions can only be used to reduce the number of possible states at a certain source code statement. At execution time, a state assertion is used to interrupt client procedure execution if the expected state is not the actual one.

AST structure

```
^(ASSERT_STATE IDENTIFIER IDENTIFIER? (STATE_LIST IDENTIFIER+))
```

Example of syntax - OOP

```
ds.assertState(ds.alpha.S1, ds.alpha.S2);
```

Example of syntax - DSL

```
[ds.alpha: S1 | S2];
```

AST of the examples

```
(ASSERT_STATE ds alpha (STATE_LIST S1 S2))
```

Linear transformation

In terms of linear transformation, an assertion about the state s will only lead to non-zero matrices if the current LTM can produce s. In other words, we will apply the dot product of $t(s)$ to the current LTM.

4.5.3 Waitfor

A waitfor statement will cause the client to halt the execution of a procedure and wait until a service sends a message informing the client that a certain state was reached in the service-side. Waitfor statements are useful for situations in which the client needs the service to be in a certain state, but this state cannot be reached as the result of the termination of any method.

This situation happens when the service requires the implementations to reach a certain state not at the end of a method execution but by means of a process that is asynchronous with the client (a parallel thread or process, or an external system).

AST structure

```
^(WAIT_FOR IDENTIFIER IDENTIFIER+)
```

Example of syntax - OOP

```
ds.waitfor(ds.S2, ds.S3);
```

Example of syntax - DSL

```
waitfor(ds:S2|S3);
```

AST of the examples

```
(WAIT_FOR ds S2 S3)
```

Linear transformation

Applying a waitfor is equivalent to applying a state assertion. In other words, if the statement waits for states s_x or s_y, we should apply a dot product with $t(s_x, s_y)$ to the current LTM.

4.5.4 If-state

For cases in which a programmer wants to control execution flow based on service state, we propose the if-state construct. If-state blocks can only refer to service states while ordinary `COND` blocks can not refer to service states. This is important to ensure that compatibility assessment can utilize if-state blocks to make the correct changes in the equivalent linear transformation.

AST structure

```
^(IF_STATE IDENTIFIER IDENTIFIER+ BLOCK)
```

Example of syntax - OOP

```
if(ds.isState(S1)) { }
```

Example of syntax - DSL

```
ifstate(ds:S1) { }
```

AST of the examples

```
(IF_STATE ds S1 BLOCK)
```

Linear transformation

An if-state subtree will affect the assessment data structure similarly to a COND block. The difference is that an if-state subtree that refers to the states $s_1, s_2, \ldots, s_N$ will perform a dot product between the current LTM and $t(s_1, s_2, \ldots, s_N)$.

4.5.5 If-version

There are cases in which the programmer wants the client to be compatible with more than one service contract. An if-version block allows a programmer to specify which block of code will be executed in presence of which service version.

If-version blocks are similar to the if-state blocks. If-version blocks also accept only boolean expressions that refer to states. This construct is used to allow for clients to be built in a way that they are compatible with a range of service contract versions.

AST structure

```
^(IF_VERSION IDENTIFIER VERSION_ID+ BLOCK)
^(VERSION_ID IDENTIFIER)
^(VERSION_ID IDENTIFIER '+')
^(VERSION_ID IDENTIFIER '-')
```

Example of syntax - OOP

```
if(ds.getVersion().atLeast(1) && ds.getVersion().lessThan(2)) { }
```

Example of syntax - DSL

```
ifversion(ds: 1+ & 2-) { }
```

AST of the examples

```
(IF_VERSION ds (VERSION_ID 1 '+') (VERSION_ID 2 '-') BLOCK)
```

Linear transformation

During compatibility analysis, the contents of an if-version block will considered if the contract version matches the IF_VERSION imaginary node. Otherwise, the if-version block will be simply ignored.

Table 4.6: Qualitative comparison between compliance assessment methods

	(a) method signature	(b) message order	(c) client source code	(d) client source code with FSM references
service contract	yes	yes	yes	yes
client contract	–	yes	–	–
client-service conversation	–	yes	yes	yes
service-side FSMs	–	yes	yes	yes
client-side FSMs	–	yes	–	–
analysis of client fragments	yes	–	yes	yes
service termination	–	detection	by design	by design
client structure	general	general	general	specific
independence from programmer's ability	high	moderate	moderate	low
compliance accuracy	low	moderate	moderate	high

4.6 Proposal evaluation

In this section we present a qualitative comparison between four processes to compliance assessment. Table 4.6 presents a comparison between four client-service contract compliance verification processes. The 'method signature' process (a) simply checks whether all method calls follow the basic contract specification. The 'message order' column (b) refers to [63], in which client-service message exchange is analyzed. The 'client source code' column (c) refers to the procedure introduced in Section 4.4. Finally, 'client source code with FSM references' (d) stands for the process of Section 4.5.

Process (a) offers just a basic compliance accuracy, and is a very simple process whose accuracy does not depend on programmer's ability.

Process (b) specifically analyzes client-service conversations, detects service termination, and utilizes client contracts. It needs some programmer's ability to correctly design client contracts. Client fragments cannot be analyzed, since a client is represented by the client contract.

Process (c), in this context, is a mere simplified version of (d). Finally, (d) can be the most accurate but is also the process whose accuracy is more reliant on programmer's ability to utilize special programming language constructs that remove vagueness in service contract usage. Service termination is not detected, but should be modeled as final states.

4.7 Conclusions

In this paper we introduced an extension of the process to analyze a client source code in face of a service contract. All constructs are translated into requirements for the current state in the service-side, which can be both utilized to check compatibility and to detect errors during execution time.

One of the big limitations of previous approaches was the inability to allow for client programmers to express information that could remove uncertainty in the compatibility assessment. For instance, we saw that the presence of loops create vague conclusions about compatibility. A programmer can easily remove

vagueness utilizing the assert state construct.

Our process can take any portion of client source code as input, as long as we can extract the reduced ASTs we need, but it requires services to be able to expose FSM in their contracts. Although there are some standards to expose service-side FSMs, these standards are yet to be popularized, as much as grid and cloud computing for instance. As our process is specially suitable for complex, hard to debug systems, the current reality of service computing restricts the process to be applicable in specific niches such as grid computing (in which virtual organizations can adopt its own standards) or private service clouds (in which service orientation does not translate directly into systems that run across organizations).

Chapter 5

Service-side constructs

5.1 Fundamental constructs

5.1.1 return

A state transition is an event similar to the completion of a method call as it marks the end of the procedure, and as it is directly related to the method contract. For these reasons, we propose that a return statement should not only give the result of the method execution, but also perform one of the state transitions the method can execute.

AST structure

```
^(RETURN ^(TRANSITION IDENTIFIER? IDENTIFIER) value?)
```

Example of syntax - OOP

```
alpha.doTransitionAfterReturn(alpha.S2);
return 30;
```

Example of syntax - DSL

```
return [alpha:S2]30;
```

AST of the examples

```
(RETURN (TRANSITION alpha S2) 30)
```

Linear transformation

Text

5.1.2 Thiscall

AST structure

```
THIS_CALL
```

Example of syntax - OOP

```
thiscall.getESTSet();
```

Example of syntax - DSL

```
thiscall.getESTSet();
```

AST of the examples

```
(METHOD_CALL THIS_CALL getESTSet)
```

Linear transformation

Text

5.2 Proposed features for a DSL

In OOP, semantics of service contracts frequently cannot be expressed by means of traditional method signatures (a collection of argument types, a return type, and possible exceptions). Some semantic features, however, may have direct impact on the flow of conversations between service and client. Differently from aspects of contracts regarding data format, for example, aspects related to method sequences can be checked during compilation of each side. Also, programmers could provide explicit clues on which sort of conversational behavior (which pattern of method calls) each side expect. That allows programmers to tune how strict are the required compatibility verifications. In this section we will present our assumptions for distributed application environments and our proposal for compatibility verification.

Services may declare zero or more finite state machines. Those machines are described as a set of states and transitions between states. Each method may be declared to be capable of executing zero or more of these transitions.

During service programming, a finite state machine object is made available to the programmer's classes and will check if the conditions of state transition are followed.

From the client-side programming perspective, such finite state machine information is useful to detect changes in the contracts, and to verify the consistency of method request order before actually calling the service. This verification can be conducted both during compilation and run times.

We argue that service extensibility with backward compatibility cannot be always guaranteed by only adding new methods to a service. Let's take, for instance, an example of a database service called DB, which provides remote access to a database. Let's assume that in the first version of this DB service, DB_1, clients have access to two methods whose signatures are `String[ ] [ ] query(String sql)` and `void update(String sql)`.

For simplicity, let's assume the `query` method simply returns a two-dimensional array containing the data obtained from the `sql` query parameter.

A second version of the same service could simply add another method with the signature '`int getDatabaseSize()`', which is just a convenience method.

After some time operating, the service design team in charge of the DB interface specification decides that the service should not immediately write the changes in an 'update' call since, for some applications, the caller may want to control when actual database writing should occur.

The third version of the DB service, DB_3 could contain a fourth method called `commit()`. And clients should, by default, explicitly call this method in order to write data to the database.

Although the signatures in DB_3 are a superset of the signatures in DB_1, a client written to work with DB_1 will be not compatible with DB_3, since an internal behavior of the service changed. On the other hand, it's expected that clients written to work with DB_1 are compatible with DB_2, since DB_2 just adds an utility method.

Table 5.1: Descriptions of methods

Method	Description
m_1	Upload file chunk
m_2	Start data mining
m_3	Download data mining result
m_4	Erase file (and cancel data mining)
m_5	Erase data mining result
m_6	Cancel data mining

Therefore, although the sole inspection of service interfaces would make us conclude otherwise, there is an important difference between the version changes $DB_1 \rightarrow DB_2$, $DB_1 \rightarrow DB_3$, and $DB_2 \rightarrow DB_3$. This difference relies on an internal state of the service, which should be known by the client to command the server correctly.

In our example, the functions $DB_1.update$ and $DB_2.update$ are not related to any state while $DB_1.update$ may change an internal state called 'writing buffer' from the value 'empty' to the value 'non-empty'.

Let us define a finite state machine as a collection of N states $\{s_1, s_2, \ldots, s_N\}$ and M state transition methods $\{m_1, m_2, \ldots, m_N\}$. Each state is an N-dimensional unit vector and each method is a linear transformation from N to N. Methods may generate and operate upon linear combinations of the states. A linear combination $z_{A,B} = p_A \cdot s_A + p_B \cdot s_B$ of two states s_A and s_B denotes the possible outcomes of the application of one or more successive operations. Scalars p_A and p_B denote the number of ways each state can be reached. Here we are only interested on whether these scalars are equal to zero or not. A zero means that the state cannot be reached through the application of the sequence of operations.

For example, let's consider the state machine depicted in Figure 5.1, a first version $DM0.1$ of a service DM that receives large files and performs some sort of data mining on it. This state machine operates over two state variables s_α and s_β. State transitions are related to zero or more methods $m_1, m_2, \ldots, m_6$.

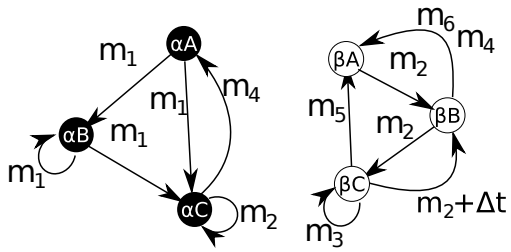

Figure 5.1: A composite state machine of $DM0.1$, the first version of the DM service

Once created, this service starts in the composed state $(s_{\alpha,A}, s_{\beta,A})$. A client is then supposed to call the method m_1 several times to upload a file to the server. After the file is completely transferred, the client is able to call the method m_2 that will start a data mining process. This method does not block waiting for the data mining to finish, since this process may take a long time to complete. Instead, the method m_2 returns immediately after being called and starts a thread in the service-side. This thread is invisible to the client, but affects the state machine β, causing the transition $s_{\beta,B} \rightarrow s_{\beta,C}$ asynchronously.

5.1 shows the meanings of each of the methods. The method m_4 will cancel a data mining if there is one in process. So the transition $s_{\beta,B} \rightarrow s_{\beta,A}$ may be executed by either m_4 or m_6.

Transitions between states can either happen as a result of method execution or some internal service process that is not visible to the client.

The service may have separated buffers to store the file and the data mining results. Then, method m_3 can be called to retrieve a previous data mining result even when a new file is being uploaded. Let's represent the states of the state machines as

$$s_{\alpha,A} = s_{\beta,A} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \tag{5.1}$$

$$s_{\alpha,B} = s_{\beta,B} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \tag{5.2}$$

$$s_{\alpha,C} = s_{\beta,C} = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

Also, let's represent the method transformations for the state machine α as

$$m_{1,\alpha} = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \tag{5.4}$$

$$m_{2,\alpha} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.5}$$

$$m_{4,\alpha} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{5.6}$$

and the transformations for the state machine β as

$$m_{2,\beta} = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} \tag{5.7}$$

$$m_{3,\beta} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.8}$$

$$m_{5,\beta} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \tag{5.9}$$

$$m_{6,\beta} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{5.10}$$

For instance, the operator $m_{2,\beta}$ means that the operation is capable of causing the transitions $s_{\beta,A} \rightarrow s_{\beta,B}$, $s_{\beta,A} \rightarrow s_{\beta,C}$, $s_{\beta,C} \rightarrow s_{\beta,B}$, and $s_{\beta,C} \rightarrow s_{\beta,C}$. Transitions $s_{\beta,A} \rightarrow s_{\beta,C}$ and $s_{\beta,C} \rightarrow s_{\beta,C}$ use the state $s_{\beta,B}$ as intermediate and depend on the help of the internal thread. Methods that do not cause transitions in a state may be defined as the identity matrix. Therefore:

$$m_{1,\beta} = m_{3,\alpha} = m_{5,\alpha} = m_{6,\alpha} = I_3 \tag{5.11}$$

In this example, the application of $m_{2,\beta}$ on the same state $s_{\beta,A}$ can cause a transition to either $s_{\beta,B}$ or $s_{\beta,C}$ depending on how long one waits to read the state.

In other cases the transition will depend on conditions checked by the method. This is the case of m_1 operating over the state $s_{\alpha,B}$, for instance. Whether the transition will be to $s_{\alpha,B}$ itself or to $s_{\alpha,C}$ will depend on whether the file transfer terminated or not.

5.3 Client-side programming

Transformation matrices can be used to verify if method calls in the client-side follow a legal order. Illegal application of sequences of transformations will lead to a zero matrix $0_{N,N}$, which means, the sequence can not lead to any state. For instance, let us consider the method call sequence of Figure 5.2, which is also represented in Figure 5.3 (a).

```
1  m1();
2  if(/*condition*/) {
3    m1(); m1();
4    if(/*condition*/) { m2(); }
5    else { m4(); }
6  }
7  else {
8    m1();
9  }
10 m1();
```

Figure 5.2: Example of IF blocks in client code

In presence of an IF block, the pre-compilation of such source code should fork the sequence in two sequences, as in Figure 5.3 (b). Finally, we will have:

$$M_a = m_1 \cdot m_1 \cdot m_1 \cdot m_2 \cdot m_1 \tag{5.12}$$
$$M_b = m_1 \cdot m_1 \cdot m_1 \cdot m_4 \cdot m_1 \tag{5.13}$$
$$M_c = m_1 \cdot m_1 \cdot m_1 \cdot m_1 \tag{5.14}$$

Using $m_{1,\alpha}$, $m_{2,\alpha}$, and $m_{4,\alpha}$ as defined before, we have as a result:

$$M_{a,\alpha} = 0_{3,3} \tag{5.15}$$

$$M_{b,\alpha} = 0_{1,\alpha} \tag{5.16}$$

$$M_{c,\alpha} = 0_{1,\alpha} \tag{5.17}$$

Therefore, there is an error in the sequence M_a. The error is easily detectable by checking at which point the value of the multiplication became $0_{3,3}$, which is the call m_2. The programmer's IDE or compiler may then show an error in the line 6.

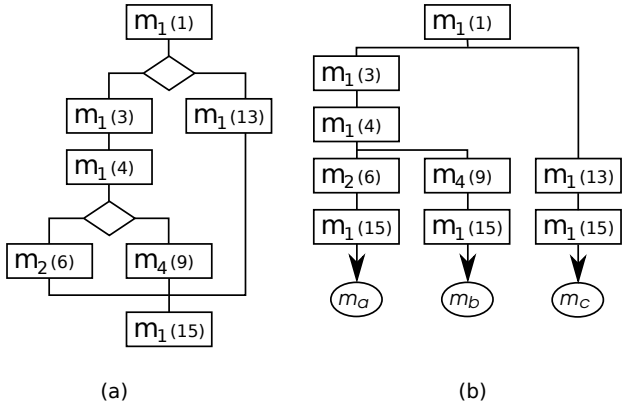

Figure 5.3: (a) sequence of method calls and (b) equivalent execution outcomes. Number in parenthesis are source code line numbers

The presence of loops in the client procedures may also be analyzed. Different numbers of iterations in a loop may generate different transformations and each of the possible transformations in a loop must be analyzed to check for consistency with methods that may come after the loop.

Let's consider the source code of Figure 5.4. Whatever the transformation generated in the end of the loop, the transformation needs to be necessarily compatible with m_5. If the loop cannot generate any of these compatible transformations, compiling such source code should result in a syntax error. Absence of syntax errors in this case does not guarantee that the client source code has no defects.

```
1 m1();
2 for(/* for statements */){
3   m2();
4   if(/* boolean expression */){ m3(); }
5   else{ m4(); }
6 }
7 m5();
```

Figure 5.4: Example of a loop in client source code

Given that $\sigma(a, b)$ is equal to either a or b randomly, passing the compilation verification just guaranties that there is at least one combination that follows the pattern

$$m = m_1 \cdot (m_2 \cdot \sigma(m_3, m_4))^x \cdot m_5 \tag{5.18}$$

in which x is a natural number, for which $m \neq 0_{N,N}$.

Figure 5.5 illustrates the tree that should be traversed to verify the syntax of the source code of Figure 5.4. Each iteration generates 2^i (where i is the iteration number) new transformations that should be checked against m_5.

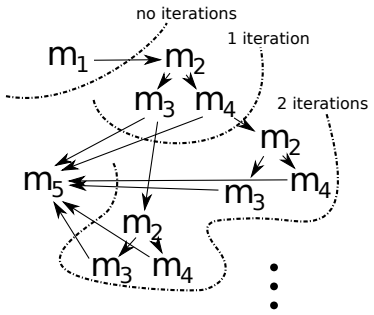

Figure 5.5: Method invocation tree for the source code of Figure 5.3

Transformations already checked should be kept in memory and if a certain transformation was already checked, it should not be considered in the next iteration. For instance, let's say $m_1 \cdot m_2 \cdot m_3$ generates a certain transformation that is equals to the one obtained by $m_1 \cdot m_2 \cdot m_4 \cdot m_2 \cdot m_4$. In this case, the rightmost branch of Figure 5.5 should be ignored when the process applies the third iteration. In other words, the sequences $m_1 \cdot m_2 \cdot m_4 \cdot m_2 \cdot m_4 \cdot m_2 \cdot m_3$ and $m_1 \cdot m_2 \cdot m_4 \cdot m_2 \cdot m_4 \cdot m_2 \cdot m_4$ do not need to be considered.

The process should be repeated until there are no more transformations to check or until it finds a certain transformation that, when applied to m_5, result in a transformation different from $0_{N,N}$.

As we already mentioned, methods can be defined to be capable of executing several transitions (which is the case of m_1 in both versions of our example). Clients may find it convenient to declare which of the possible transitions they expect to happen in the service-side for the following reasons:

1. Client source code readability may be improved if method calls show clearly which service-side state should be reached.

2. Giving specific requirements to the service during runtime may allow for the service to cancel the execution if the requirements given by the client application cannot be met.

3. As in programming languages that use design-by-contract, such as Eiffel [1], declaring such requirements frees the client application from having to verify service state or guess which is the service state. In other words, it avoids defensive programming in the client-side.

4. Incompatibility detection may be improved. Because clients are specifying exactly which transitions they expect from a certain method, if the transition used is not present in a certain version, such incompatibility can be easily detected.

[1] Eiffel Software website: http://www.eiffel.com/

```
1 m1[alpha:*=>B | B=>C](argument1, argument2);
```

Figure 5.6: Example of a syntax for expected transitions declaration

We argue that declaration of EST should be checked for syntax and consistency with the state machines as defined in the service interface. Therefore, such declarations of EST should be included as part of the DSL grammar and read during the source code verification step.

Figure 5.6 shows an example of syntax for a DSL based on Java. The declaration between square brackets means that the EST refers to the state alpha, and accepted transitions are any transition that leads to the state B or a transition from B to C.

In therms of matrices such declarations should be applied as a mask that selects some cells of a transformation matrix. Let's use the characters '.*' to denote such operation. The result $C_{M,N}$ of an application of mask between two matrices $A_{M,N}$ and $B_{M,N}$ is depicted in the second row of 4.5. Boolean operations should be performed analogously.

An asterisk in the left side of a transition is translated into a column of ones. Then the strings `*=>B` and `B=>C` should be translated, respectively, into the matrices

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \tag{5.19}$$

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \tag{5.20}$$

Applying (5.19) and (5.20) in (5.4) we have

$$\begin{aligned} [s]m'_{1,\alpha} &= m_{1,\alpha} \cdot * \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix} \vee m_{1,\alpha} \cdot * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \\ m'_{1,\alpha} &= \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \vee \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix} \end{aligned} \tag{5.21}$$

The transformation $m'_{1,\alpha}$ is more selective than $m_{1,\alpha}$, since $m'_{1,\alpha}$ does not have the transition $s_{\alpha,A} \rightarrow s_{\alpha,C}$. When applying a mask, the selectiveness of the original method can be unchanged or increased. If, in a new version of the service DM, the new definition of $m_{1,\alpha}$ generates $m'_{1,\alpha} = 0_{N,N}$, this new version is incompatible with the declaration of Figure 5.6. Analogously to `*=>B`, strings with an asterisk in the right side should be translated into a matrix with a row of ones.

The use of ESTs can be specially useful when declared in consecutive method calls, or inside of FOR loops. Figure 5.7 shows an example containing both cases. In case of loops or conditional blocks, a state may become ambiguous, as in the end of the FOR loop, in Figure 5.7. For these cases, a client may also declare an EST unbounded to a method invocation (line 5). This EST does not necessarily send any message to the service side. Instead, it may check at runtime if the state is compliant with the EST and raise an exception in case the check does not pass.

Verification errors may happen (1) while applying ESTs to method calls if the mask operation generates a zero matrix, or (2) if unbounded ESTs require an unreachable state. In both cases, loop and conditional blocks expansion trees should be used to search for inconsistencies.

Also, warnings can be shown whenever a certain EST does not eliminate any transition. So programmers will know that removing an EST does not have any effect in the execution.

In (5.21) we can see that `*=>B` did not filter any possible transition of $m_{1,\alpha}$. Therefore, `*=>B` can be marked as unnecessary in the programmers' IDE.

5.4 Service-side programming

State transitions in service-side programming source code, on the other hand, are easier to ensure because service containers can be used to control the behavior of services.

```
1 m1[alpha:A=>B](arg1, arg2);
2 for(/*for statements*/){
3   m1[alpha:B=>B|B=>C](arg1, arg2);
4 }
5 [alpha:C];
6 m2();
7 m3();
8 m4();
```

Figure 5.7: Example of a syntax for expected transitions declaration

```
1 public class m1Alpha {
2   public void toB(){ /* transition to state B */ }
3   public void toC(){ /* transition to state C */ }
4   public boolean isA(){ /* returns true if the state is A */ }
5   public boolean isB(){ /* returns true if the state is B */ }
6   public boolean isC(){ /* returns true if the state is C */ }
7   public Set<EST> getESTSet(){ /* returns the EST set defined by the ↩
        client */ }
8 }
```

Figure 5.8: Example of state object class for the method m_1

We propose that service methods should receive state objects as an extra argument. These objects should be built based on the service interface definition and only have valid state transition operations. For example, Figure 5.8 shows the outline of a class for state objects referring to α, given to the method m_1.

Since m_1 is not supposed to make transitions to the state $s_{\alpha,A}$, there is no method called `toA()`.

As we already said, ESTs declared in the client-side may also be used by the service. A service may use a received EST to evaluate, at runtime, the feasibility to reach the state expected. If m_1 concludes that none of the expected states can be reached, the service may choose to send a remote exception back to the client instead of executing its instructions properly.

ESTs, if declared by the client, can be provided as part of the state object, as in the line 7 of Figure 5.8.

After a method finishes, the container must check the state object to make sure one of the legal transitions (as defined in the service interface, and potentially narrowed by a client EST) was performed. In case a transition was not performed, the service container must return an error message to the client. Also, if the container is responsible for managing transactions, it can try to rollback the operations executed by the service method.

5.5 Service Specification Evolution

A new version $DM0.2$ of the same service may have a different state transition diagram as depicted in Figure 5.9. In $DM0.2$ all methods have the same signatures and the same meanings as in $DM0.1$, but now there is a new state machine called γ. This new specification of the service requires the data mining to have a FIFO queue manageable by the client using the methods m_3 and m_5.

Let's assume that the designers of $DM0.2$ considered the state $s_{\beta,C}$ unnecessary since a processor may be idle or doing data mining independently on the existence or not of data mining results to download. The transition $s_{\gamma,A} \rightarrow s_{\gamma,B}$ can be caused by m_2 but not instantly. Instead, the transition is done by the same hidden thread that causes the transition $s_{\beta,B} \rightarrow s_{\beta,A}$, which may happen long after the method m_2 finished.

So despite of having the same method signatures, and no extra or missing methods, a client designed to work with $DM0.2$ may be incompatible with $DM0.1$. So $DM0.2$ is not backward compatible.

For instance, let's consider a sequence $p = m_1^x \cdot m_2^y \cdot m_3^y \cdot m_5^y$ where x is the number of chunks of the file to be uploaded and y is the number of data mining executions. The sequence p is legal in $DM0.2$ and will generate the transformations:

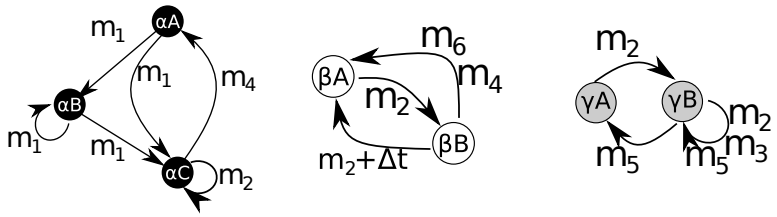

Figure 5.9: A new version $DM0.2$ of the same service

$$p_\alpha = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \tag{5.22}$$

$$p_\beta = \begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix} \tag{5.23}$$

$$p_\gamma = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{5.24}$$

But verifications using $DM0.1$ will find that $m_5^y = 0_{N,N}$ for any $y > 1$, which means the sequence p is only valid in $DM0.1$ for $y = 1$.

On the other hand, a client made to utilize $DM0.1$ will be compatible with the second version since the second one has less restrictions in the methods m_3 and m_5. The result of a client made for $DM0.1$ utilizing $DM0.2$ is the underuse of the result queue, but the design of the interface guaranties forward compatibility from $DM0.1$.

Part III

π-calculus approach for contracts

Chapter 6

A Contract-Centric Approach to Compatibility

Abstract

Distributed computing has reached the big audiences. From mobile devices that can access user's photo collections stored in the cloud, to on line cooperation between scientific institutions, distributed computing has become part of the daily life of an expressive amount of people. It is, then, important to rethink the way we express compatibility between systems. This paper proposes a mechanism to check service compatibility based on contracts. We propose that a contract should be specified in terms of a process calculus and that interacting services should have their algorithms verified against such process calculus to see if they can reach a target state, meaning, they can successfully interact. In order to guide the compatibility check we propose a variation of the Java programming language to create a domain specific language (DSL). This DSL, along with a run time model, was specially designed to allow for an automated formal verification of behavior. This section is based on [59].

6.1 Introduction

Distributed services are fast increasing in importance. With the advent of cloud computing, nowadays users can enhance their mobile gadgets with remote services provided through the network. Currently, what we are seeing is the multiplication of rich client applications that run inside of the mobile devices and usually communicate with a number of services that provide capabilities to the devices, such as additional storage, additional processing power, and domain-specific algorithms such as the ones in geographic information systems.

In this scenario, it is important to analyze how developers create clients and services. In an environment in which services are expected to be composed easily as software components provided by more than one vendor, application developers do not have the luxury to fully understand the internals of the remote services they use as part of their applications. Likewise, developers of such services can hardly completely foresee all client implementations. So trying to come out with service interfaces that prevent misuse by clients can be tricky. In cloud computing, a service is usually provided by a company for a fee. This is one of the key differences between computing clouds and computer grids, according to [33]. So the same client application may interact with a range of service providers that are chosen based on a number of factors, including the price of the service.

Ideally, client applications should be checked before a remote service is hired and the same client should be able to successfully interact with any service that implements a certain contract to avoid vendor lock-in. One of the problems we need to address is then how to allow for the developers of the client application to formally verify if their client applications can successfully interact with a service specification rather than with a specific service implementation. We also assume that the selection of the right service provider can be done in an automated fashion, for instance, using some scheme in which services compete to serve a certain client (as in the Grid economy model [16]), so the actual software composition cannot be determined during coding.

We propose that service specifications should contain state dynamics, which tells the clients in which circumstances each service method can be called. Also, we need to take into account object mobility in

order to allow for computation to be placed close to data without impacts in contracts. Both mobility and interaction between services are aspects addressed by the π-calculus [71], which we apply to be part of contracts. The π-calculus is build around the concept of bisimulation or mutual simulation, a formalization that provides a criteria to tell if a process emulates the behavior of another one.

The rest of this paper is organized as follows. The following section introduces related research. Section 6.3 introduces our model for service contracts, while Section 6.4 introduces a DSL we created on top of the Java programming language to support advanced contract verification. In section 8.7 we discuss our contributions. Finally, we conclude this paper on section 8.8.

We also provide two appendices at the end of this paper: Appendix **??** provides a brief introduction of the π-calculus, while Appendix 6.7 defines a special π-calculus context we use to check client implementations of contracts. This context provides remote service state inspection, which is needed by the DSL.

6.2 Related research

Interoperability between old and new code based on types was proposed by [72]. It also uses π-calculus, but as a means to define a type system, instead of trying to check imperative code against a contract.

Papers [57] and [58] propose that internal finite state machines should be added to the service definition in order to enable the service contract to describe legal sequences of method calls. An approach using finite state machines was proposed in [19], which is similar to [57]. Verifications based on finite state machine allow us to identify wrong patterns of service methods, but they do not take into account behavior equivalence by means of simulation, they do not offer a way to represent channels, and they do not allow us to easily represent unobservable transitions. The π-calculus we are using here includes those missing features.

Service compatibility is an issue addressed both in [14] and [13]. Both address the problem from the point of view of message types and termination protocols. Termination is an important feature also addressed here (a termination can be modeled as both interacting services reaching a zero state), but we also tackle the problem of message sequences, especially in cases in which states are time dependent (which we model using the π-calculus τ).

In this paper, we claim that exposing service state can improve client-service cooperation and formal verification of compatibilities. This approach is similar to "design by contract" [61], which is the idea behind some programming languages such as Eiffel [1]. Design by contract defines pre-conditions and post-conditions respectively as conditions that the caller of a function should comply with in order to call the function, and the promise made by the called function regarding the result of the function execution. A contract in a design by contract is intended to simplify development of applications in general, not only distributed applications. As a result, a contract aims at reducing defensive coding, which is source code that tries to check pre-conditions within the execution of a method. Also, the caller of a method needs to check if the post-conditions were observed by the method called.

Analyzing Web service compatibility using graphs and protocols was addressed in [17], [12], and [25]. A formalization of compatibility was also proposed in [36].

OurGrid also proposes using transformations over an OOP language to create distributed systems [52] [53]. The strategy of OurGrid is to allow programmers to mark certain Java threads as points to be exported for the grid to execute. AOP aspects will identify those explicit marks and replace calls to the execution of threads, by procedures that will request the execution of the threads in a remote node. The model is quite simple, but can easily improve performance of programs written to be executed locally, by offering memory and CPUs that reside in a remote node without the programming costs usually required in distributed execution of tasks defined locally. AOP is based on the idea that standard OOP design does not allow for correct mapping of crosscutting concerns, which are addressed by aspects (an AOP primitive similar to a class). In contrast, in our design, we propose a model in which interactions with external services should be located at a special layer. Restricting interactions to a layer is what allowed us to check contract compatibility as we will present on the next section.

On [63] it is proposed that Web Service Business Activity (WS-BA) [65] termination protocols (coordinator initiated or participant initiated completion) should be applied to web services as a set of constraints. Such declared constraints allows for formal verification of algorithms to ensure that a compatibility criteria is met. This criteria is one in which both services should reach an acceptable state. To avoid deadlocks and race conditions, the paper proposes to use SOAP Service Description Language (SSDL) as the way to express those constraints. SSDL constraints are then translated into Process Meta Language (PROMELA) source code, which is in turn execute by the SPIN model checker [2,41]. The idea is that interaction between

[1] http://www.eiffel.com/

services should be grouped into activities that take place sequentially. Each activity should complete in a consistent state. Our compatibility criteria is different from [63] in that we propose that compatible services are those that can interact based on a process expressed in terms of π-calculus. Our model allows for what would be equivalent to parallel activities taking place.

Our proposal is also related to DbC. The Java Modeling Language (JML) [46, 47] is perhaps the most widely used approach for DbC in Java. JML is an already well-established language and has many tools [15] to support its usage by programmers. The idea behind JML (as is the approach of other DbC languages such as Eiffel [6]) is that contracts should be written as part of the source code. In JML, contract constraints are expressed by annotations that are added in especially formatted comments. Here we also exploit DbC in the sense that what we call "a contract" can be the same as the concept in DbC. In other words, our contracts can also express pre-conditions, post-conditions, and invariants.

But our contracts aim at describing behavior rather than providing legal conditions of data to avoid defensive programming. For instance, our contracts are able to specify that a certain agent should be able to spawn more agents and pass a reference to such newly-created agents over the network. After those references are made available to remote nodes, they allow for new interactions to take place.

Another difference is that our concept of contract is one in which the contract describes an abstraction in which interaction events are described. In Java, those interaction events are mapped to method calls. In web services, interaction events could be web service calls. Our approach is that, once interfaces have been defined for distributed agents to interact in a particular programming language such as Java, it is possible to describe behavior of interactions using π-calculus not as a means to ensure data validity but rather to judge if the possible multiple implementations of the contract can simulate the contract. A "simulation" in π-calculus terms, as we will see later, is a relation that ensures that the behavior of a process P' remains confined in the limits imposed by another process P. In our case, P is the contract, while P' is one of the implementations of agents that are subject to the contract.

6.3 A model for service contracts

In this section we describe our model for service contracts. Because introducing the π-calculus is beyond the scope of this paper, we provide a brief introduction to it on Appendix **??**. We discuss an extension of the π-calculus to conveniently represent object mobility on Appendix **??**.

Before we describe our model, it is important to explain our naming conventions. The WS-BA naming defines that one of the services should be called the "coordinator" (the party that initiated a termination) and the other should be called "participant". Services may switch roles during interaction.

We are aiming to develop a programming model for both parties based on a single contract, so we need to define fixed roles for each party. Hereinafter we will call "client" the software agent that requests a remote service instance, and the "service" the software agent that is instantiated or allocated to serve a client. After being contacted by a client, services may initiate communication with the client asynchronously (therefore playing the role of a coordinator in WS-BA terms) and send data through this communication.

A contract $C := (C_C, C_S)$ is a pair of π-calculus expressions that represent, respectively, the service from the standpoint of clients, and a family of clients that can interact with the service. We say that C_C defines a family of clients, instead of a single client, since services should implement the whole contract, while clients can implement only part of the contract.

6.3.1 Compatibility criteria

The expression of the client implementation is obtained from the client source code. We discuss the extraction of these expressions on the next section. As we will see, we allow for the DSL source code to interact with the service and to query about the availability of methods. For this to be possible, the client implementation is put on a context that differs from the one in that the client contract expresses.

We call such a context Θ. We use $\Theta(C_{C,i}, C_C)$ to represent a client implementation $C_{C,i}$ in the context Θ created based on the client contract C_C. In the Θ context, $C_{C,i}$ can call some special channels called ϑ_x to query for the availability of method x. The caller of ϑ_x should pass two reply channels. One for TRUE, and another for a FALSE response.

For example, let us assume a contract implementation C_{impl} that calls a method called m_1 and then checks if the method m_2 becomes available. If m_2 is available, the implementation will call m_3. If not, the implementation will make m_4 callable instead. The equivalent expression is:

$$C_{impl} = \overline{m_1}.\overline{\vartheta_{m2}}\langle\vartheta_T, \vartheta_F\rangle.(\vartheta_T.\overline{m_3} + \vartheta_F.m_4) \tag{6.1}$$

Interactions with ϑ_x channels are not observable from a process put in parallel with Θ, but interactions with methods in the client contract are visible. In the example above, a process X put in parallel with $\Theta(C_{impl}, C_C)$ would be able to observe a call to $m1$, and then either a call to m_3 or m_4 becoming available to be called. But X could not observe interactions involving ϑ_{m2}, ϑ_T, or ϑ_F.

Our compatibility criteria is that a client implementation $C_{C,i}$ is considered to be compatible with the client contract C_C if $\Theta(C_{C,i}, C_C)$ simulates C_C.

In other words, if there is a relation R such that $\Theta(C_{C,i}, C_C)RC_C$ is a simulation. When R is not a bisimulation, $C_{C,i}$ implements the client only partially, which is still considered a compatibility relation. A more forgiving definition of compatibility accepts all $\Theta(C_{C,i}, C_C)$ that are weak simulations of C_C. We will use both criteria, which we will call strong compatibility and weak compatibility respectively. A formal definition of Θ is too long for the body of this paper, so we left it for Appendix 6.7.

Fig.6.1 shows the relationships between contracts and implementations. The pair $C1_C \mid C1_S$ represents the first version of the contract while $C2_C \mid C2_S$ represents the second version. Two client implementations and two service implementations are represented in the figure. A client implementation may simulate one or more client side expressions. While $C_{C,i5}$ simulates both $C1_C$ and $C2_C$, $C_{C,i2}$ simulates only $C2_C$. In Fig.6.1 all four implementations ($C_{C,i5}$, $C_{C,i9}$, $C_{S,i12}$, and $C_{S,i20}$) were identified with arbitrary IDs (5, 9, 12, and 20) in order to clearly show that implementations are not directly connected with a single version of the contract.

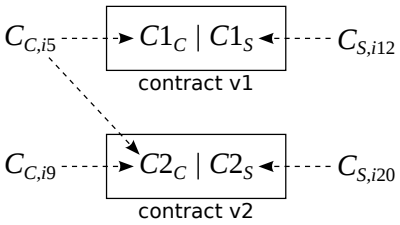

Figure 6.1: Example of relationships between contract expressions and implementations. Dashed arrows mean simulations.

6.3.2 Example of compatibility checking

In order to illustrate this concept, let us analyze an example. Let us say we want to implement a frequent item set data mining service, such as a priori [69] or fp-growth [37]. Our service will consist of the functions: "upload chunk of data" (for instance, a list of item sets), "start data mining", "cancel data mining", and "deliver result". To simplify notation we will call these functions, respectively, m_1, m_2, m_3, and m_4. Functions m_2 can only be called after function m_1 was called at least once. Also, the completion of m_2 causes the function m_4 to be called on the client, which needs to implement a listener interface. So m_4 is the channel through which the service can send an asynchronous message to the client. During the execution of the data mining, a client may call m_3 to cancel a previously issued data mining task, allowing for the client to add more data. The contract is expressed by:

$$\begin{aligned}
C1_C &= S1_{C,1} \stackrel{\text{def}}{=\!=} \overline{m_1}.S1_{C,2} \\
S1_{C,2} &= \overline{m_2}.S1_{C,3} + \overline{m_1}.S1_{C,2} \\
S1_{C,3} &= \overline{m_3}.S1_{C,2} + m_4.0 \\
C1_S &= S1_{S,1} \stackrel{\text{def}}{=\!=} m_1.S1_{S,2} \\
S1_{S,2} &= m_2.S1_{S,3} + m_1.S1_{S,2} \\
S1_{S,3} &= \tau.S1_{S,4} + m_3.S1_{S,2} \\
S1_{S,4} &= \overline{m_4}.0
\end{aligned} \tag{6.2}$$

The internal action τ above represents the time it takes for the data mining to complete. In other words, the completion of the method m_2 does not imply that the server immediately becomes ready to execute m_4. Instead, the end of the execution of m_2 puts the server in a state in which the only externally observable action is m_3 ("cancel data mining") and only after an unknown delay (the time that is takes for the server to process the request, represented by the τ) the service will make a transition to $S1_{S,4}$ and the input action $\overline{m_4}$ will be ready. It is also important to note that we need a sum on state $S1_{S,3}$ in order to implement an

exclusive OR between two options: either the system will make a transition back to $S1_{S,2}$ or the service will advance to $S1_{S,4}$. If the service makes a transition to $S1_{S,2}$, the τ transition becomes unavailable, meaning the result of τ, the call of m_4 on the client, will be not observed by the client.

The behavior of τ shows that the π-calculus expression of the service contract does not completely specify how the service works internally. Instead, it provides constraints to which sort of state transitions the system can make. Service terminations are marked with a 0 (zero), as in $S1_{S,4}$. In general, services require a termination from the client side to ensure that the service can release resources allocated to serve the client.

It is also worth noting that both the a priori and the fp-growth algorithms are based on at least one pass over the data to determine most frequent items. This list is later used to find the most frequent item sets. Since there is not function to remove data, a call to m_3 during the data mining processing does not necessarily mean that the service should delete the partial item counting results, or even that the service should actually stop counting items. Adding more items can happen in parallel with counting the items already added. On the contract above, it is up to each service implementation to provide its own concrete interpretation of what m_3 does. Therefore the contract neither completely specifies how the service works internally, nor exposes all the internal states of the service.

Obviously, splitting the process above into states is merely a convenience to match service state with π-calculus expressions. The service contract above is equivalent to the following one:

$$\begin{aligned} C1_S = new\{s_2, s_{2L}, s_3\}\ (m_1.\overline{s_2}) \mid !(s_2.(m_2.\overline{s_3} + m_1.\overline{s_{2L}})) \mid \\ !(s_{2L}.\overline{s_2}) \mid !(s_3.(\tau.\overline{m_4}.0 + m_3.\overline{s_2}) \end{aligned} \tag{6.3}$$

In (6.3) we had to introduce new bound variables to represent transitions between states. We also needed three replications in order to represent the arbitrary repetitions of the same state according to commands issued by the client. We also had to add some tricks to make the process behave as expected: an invisible state ($s_{2L}.\overline{s_2}$) to allow for the loop caused on m_1. As we can see on this example, using states as in (6.2) usually leads to more readable contracts.

Now let us analyze the compatibility between this contract and a client implementation:

$$C_{C,i1} = \overline{m_1}.\overline{m_2}.\overline{m_3}.\overline{m_2}.m_4.0 \tag{6.4}$$

Because there are no references to ϑ channels, the behavior inside of the context $\Theta(C_{C,i1}, C_C)$ and outside is the same. So on this first example we will analyze $C_{C,i1}$ alone. It is not hard to see that $C_{C,i1}$ simulates $C1_C$ through the following binary relation $C_{C,i1} R C1_C$:

$$\begin{aligned} R = \{&(\overline{m_1}.\overline{m_2}.\overline{m_3}.\overline{m_2}.m_4.0, S1_{C,1}), (\overline{m_2}.\overline{m_3}.\overline{m_2}.m_4.0, S1_{C,2}), \\ &(\overline{m_3}.\overline{m_2}.m_4.0, S1_{C,3}), (\overline{m_2}.m_4.0, S1_{C,2}), \\ &(m_4.0, S1_{C,3}), (0,0)\} \end{aligned} \tag{6.5}$$

The existence of R above is enough for us to consider the client $C_{C,i1}$ compatible with any service that implements $C1_S$. The inverse of R is not a simulation because the pair $(S1_{C,3}, m_4.0)$ (the inverse of the penultimate pair in R) cannot be in any simulation, since $S1_{C,3} \xrightarrow{m_3} S1_{C,2}$ but there is no process P such that $m_4.0 \mid P \xrightarrow{(m_3)}$. Therefore, R is not a bisimulation. Event if there is another client implementation $C_{C,i2}$ such that $C_{C,i2} \sim C1_C$, we cannot say that $C_{C,i2}$ is more compatible with $C1_C$ than $C_{C,i1}$.

Naming states makes it easier not only to understand the contract, but also to debug it and program clients. It is easier to debug a service as the run time environment is able to log a state transition (which happens whenever a name such as $S1_{C,4}$ is replaced by its expression, in this case $\overline{m_4}.0$). Programming the client side is easier since exposure of server state allows the client to make decisions based on the server reactions and narrow the expectancies a client has about the service behavior, which are concepts that we introduce on the next section.

A contract is the client's perspective towards the service behavior. For example, the contract above also fits in a situation in which data mining is actually performed by a resource that is external to the service. Such external resource could asynchronously notify the service that it completed processing the data mining and this asynchronous notification would make the service make a transition to the $S1_{S,4}$ state. From the client point of view, the internal and non-observable structure of the service is irrelevant as long as the service follows the contract.

For the $S1_{S,4}$ state to work, the output action $\overline{m_4}$ should be callable on the client. So $\overline{m_4}$ represents a requirement that the client should comply with. What the contract does not explicitly shows is that the internal client dynamics should be such that the client is ready to have m_4 called when the server tries to call it. The source code of the client can be checked to have such a property, as we will explain on the next section.

6.3.3 Method signatures and object reference passing

So far, our contract only defines names for functions (or methods), but an actual contract needs also to specify what are the received data and returned result. We model such aspect of contracts not as π-calculus expressions, but as a list of required and optional arguments and a return type. The list of arguments may contain mandatory channels. Method names and channels are the only elements represented using π-calculus expressions. For instance, let us consider a service method whose signature is `void m1(T x, int y)`, in which `T` is a class defined by the client. The equivalent expression should ignore the argument `y` as it is a primitive, so it does not reflect a change in the structure of reachable references to objects. Therefore, the equivalent expression of this method would be simply $m_1(x)$. The object `x` should also comply with the later use made of it.

For instance, let us assume that the complete service contract is $C_S \stackrel{\text{def}}{=\!=} m_1(x).x{:}\overline{a}(b)$, in which a client receives an object [2]. Then whatever comes as x should have a channel called a that is able to receive a channel b. For a more detailed example, we refer the reader to Appendix 6.4.7. As we will see on the next section, references passed through channels should point to objects within a protected layer that insulates these objects from external factors that could change the behavior of objects at run time.

Verification of compatibility based on method signatures is trivial, so we will not cover this topic here.

Our method can be used to check for simultaneous termination, which can be modeled as both client and service processes reaching a zero state (also called stop state). We show two examples of reaching zero states on appendices 6.7.1 and 6.7.2.

6.4 A domain specific language for process calculus-based contracts

Now that we already established what our compatibility criteria, in this section we introduce the treatment we have created to translate source code into π-calculus expressions. We then take the obtained π-calculus expression and check if there is a simulation from this π-calculus expression on a Θ context and the expression of the client. A similar process is used to check compatibility between service implementation and service contract.

Both client and service implementations should be a compilation unit written using the DSL we describe in this section. Fig. 6.2 summarizes our programming model. Objects in the client are divided into three layers: a business layer that contains business logic objects from Java classes, a client layer that contains objects specified using our DSL, and the object proxy layer that contains local objects that are proxies for remote services.

Business objects are ordinary Java classes that can only indirectly interact with services through client objects. Business objects may have a behavior that is hard if ever possible to predict. For instance, some of these objects may respond directly to user input. So compatibility with a service contract is only checked on classes that were specified using our DSL: those on the client and service layers. Our strategy is to restrict the entities that we need to check compatibility to those entities that are in those two layers.

Each client object provides one or more public methods (marked as "c1m1" etc in Fig. 6.2). Those methods are accessed by business objects under conditions specified by the declaration of client object classes, as we will see later. Each client object interacts with one or more proxy objects.

A proxy objects is an implementation of the remote proxy design pattern, which were also applied by [75]. Each proxy object is a reference to a remote service and encapsulates a connection. Calls to methods in a proxy object start a process that uses the network to call the equivalent method on the server.

On the server side there is one service instance (a service object) for each proxy object. Each such object should behave as specified in the contract.

[2] The syntax $x{:}a$ means a channel a that is associated with the name x. Whenever the name x is passed, all associated channels are also implicitly passed. This is not part of the standard π-calculus, but an extension we created in order to make it easier for us to represent object reference passing, as we explain on Appendix **??**

On the top of Fig. 6.2 there are (database-like) cardinality relations. When a client object is instantiated this object receives one of more service references through inversion of control. Service references are not shared with other client objects, so client objects have a one-to-many cardinality with service proxies and service objects. Each service proxy represents a single instance of a service object, so the cardinality is one-to-one.

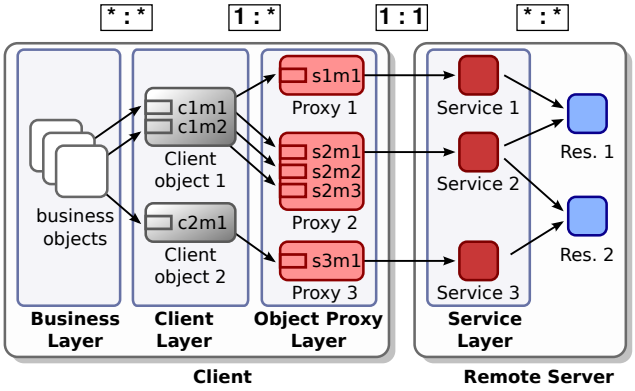

Figure 6.2: Programming model for client and service

Fig. 6.3 shows an example of a client object class. The `InitialState(A)` annotation decorating the client class sets the initial state of this object to be A. The two `State` annotations specify which should be the current state in order for each method to be called. If any business object tries to call any of these two methods while the client object is on the wrong state, an exception is thrown for the caller and the method does not get invoked.

The `Scope` annotation in the `m4` method makes this method available only for the service identified by `s`. So no object in the business layer and no other service can call this method.

```
1  @InitialState(A)
2  clientclass Client1 {
3      Service s;
4      @State(A)
5      public void sendDataAndRequestDM(DataSource d) {
6          s.m1(d); s.m2();
7          System.out.println("Canceling...");
8          s.m3(); s.m2();
9          to(B); // state transition to B
10     }
11     @State(B)
12     @Scope(s)
13     public void m4(DataSource d) {
14         // use data from service...
15         to(C); // state transition to C
16     }
17 }
```

Figure 6.3: Example of client object class

From the perspective of the service `s`, the source code in Fig. 6.3 translates into $C_{C,i1}$, as defined in equation (6.4). Note that the call to `System.out.println()` is ignored as it does not affect the execution flow, neither interacts with the service.

The service `s` is represented by a generic `Service` type, instead of a reference to any particular service type. The type `Service` simply states that `s` in fact should be checked against a yet to be specified service contract.

Fig. 6.4 shows another client object class `Client2` that interacts with two remote services. This class is translated into two distinct expressions, one for each service. The interaction with `s1` is translated into $C_{C,i3} = m_4.\overline{m_3}$, while the interaction with `s2` is translated into $C_{C,i4} = \tau.m_6$.

$C_{C,i3}$ means that `s1` has the opportunity to call m_4 and then some unknown event may trigger the call to m_3 on `s1`. It is unknown by `s1` that an interaction with `s2` is responsible for such call to m_3.

The expression of $C_{C,i4}$ shows a different perspective towards the `Client2` class. Although the method m_4 has no reference to `s2`, the state dependency adds a τ to $C_{C,i4}$. If, for instance, $C_C = \overline{m_3}.0$, then $C_{c,i4}$ simulates C_C, but only weakly, since $C_{C,i4} \rightarrow^{m_3} 0$. In other words, $C_{C,i4} \stackrel{m_3}{\Longrightarrow} 0$.

```
1  @InitialState(A)
2  clientclass Client2 {
3      Service s1;
4      Service s2;
5      @State(A)
6      @Scope(s1)
7      public void m4(DataSource d) {
8          System.out.println("x");
9          to(B);
10     }
11     @State(B)
12     @Scope(s2)
13     public void m6(DataSource d) {
14         s1.m3();
15         to(C);
16     }
17 }
```

Figure 6.4: A client object class interacting with two services

It is important to note that we do not need the contract in order to extract the equivalent π-calculus expression. Once we have obtained such expression we are able to check the compatibility of this client with one or more service contracts.

The field `s` gets instantiated and set by inversion of control when the client object is instantiated. This field will point to a remote proxy, a local object that represents the remote service and encapsulates all the complexity related to network communication.

Besides the extensions to the Java syntax that we introduce in this section, client objects also have some additional restrictions:

- **External observation** – A client method should not be called from another client method on the same client object. This avoids having a recursive call to the same method. The rationale is that recursion may lead to the execution flow of a single client method to include other client methods.
- **Controlled interaction domain** – A service reference cannot leave the client object. If a reference to a service leaves the client object, other objects in the client may call service methods, so from the point of view of the service, the client is not only what the client object represents. Coming up with the equivalent π expression is impossible in the general case, as we cannot predict which classes will end up having access to the service.
- **Opacity** – Fields of client objects should not be directly accessible from other objects.

Methods of objects in the client layer are all synchronized, which is a limitation added to ensure that the beginning and end of methods happen in a way that is consistent with the state dynamics.

At follows we present how control flow blocks and some DSL-specific features we have created are translated into π-calculus expressions.

6.4.1 Generic if-blocks

If-blocks are replaced by a sum in which internal actions control execution flow. All internal actions will be restricted to the context of the expression and represented by the Greek letter κ.

$$new\{\kappa_1, \kappa_2\}\ (\overline{m_1}.\overline{\kappa_1} \mid \kappa_1.\overline{m_2}.\overline{\kappa_2} + \kappa_1.\overline{m_3}.\overline{\kappa_2} \mid \kappa_2.\overline{m_4}) \tag{6.6}$$

```
1 public void x(boolean condition) {
2     s.m1();
3     if (condition) { s.m2(); } else { s.m3(); }
4     s.m4();
5 }
```

Figure 6.5: Simple if-block

An if-block without an else-block is modeled the same way with the exception of else part which is not present. If-blocks with many if-else conditions (which are equivalent to switch-case blocks) are modeled as many items in the summation, each of them starting with κ_1 and ending with $\overline{\kappa_2}$. Note that the condition on Fig. 6.5 was simply replaced by a sum, as the general case is that the condition cannot be evaluated from the perspective of the service contract. For instance, the values referenced by the test condition may come from outside the class, as a method argument.

Clearly, such execution flow controlled by something unrelated to the contract impoverishes all analysis that can be made using equivalent process calculus expressions such as (6.6). We offer a better construct on the next subsection.

6.4.2 Contract-based if-blocks

Our DSL allows for contract entities to be used as if block conditions. For instance, consider the following source code:

```
1 sm1();
2 if (s.m2 callable) { s.m2(); } else { s.m3(); }
3 s.m4();
```

Figure 6.6: Simple if-block

The main difference from the previous source code is that this one verifies if the method m_2 is available. The actual difference is on the equivalent π-calculus expression:

$$new\{\kappa_2, \vartheta_T, \vartheta_F\}\ (\overline{m_1}.\overline{\vartheta_{m2}}\langle\vartheta_T, \vartheta_F\rangle \mid \vartheta_T.\overline{m_2}.\overline{\kappa_2} + \vartheta_F.\overline{m_3}.\overline{\kappa_2} \mid \kappa_2.\overline{m_4}) \tag{6.7}$$

The call $\overline{\vartheta_{m2}}$ represents a test that interacts with the current service state to check what is the current state, while ϑ_T and ϑ_F stand for a true or false responses respectively. Interaction is provided by the Θ context we briefly introduced on the previous section and that we describe in details on appendix 6.7.

Let us now assume that the service contract is as follows:

$$\begin{aligned} S_1 &: m_1.S_2 + m_1.S_3 \\ S_2 &: m_2.S_4 \\ S_3 &: m_3.S_4 \\ S_4 &: m_4.0 \end{aligned} \tag{6.8}$$

Clearly, the source code in Fig. 6.5 is not guaranteed to be compatible with the contract above since the Boolean condition in the if clause is unknown during the verification step, and in the general case unrelated to the service state. So, in principle, nothing mandates that the call to m_2 will be placed only if m_2 is ready to be called. On the other hand, the source code in Fig. 6.6 is without a doubt compatible with the contract. In fact, it can be even argued that the contract suggests a client as in Fig. 6.6. An if-block is necessary in order to react to an arbitrary choice of state transition made by the service. This choice is represented by the plus sign in the expression of S_1 above.

If-blocks that interact with a reflection over the current state of the service can also be used to make a single client compatible with a range of contracts. In order to show that, we need to analyze a second version $C2$ of the data mining service. The main difference is that $C2$ allows for the client to add new item sets even during data mining processing using m_1. $C2$ contract is:

$$
\begin{aligned}
C2_C = S2_{C,1} &\stackrel{\text{def}}{=\!=} \overline{m_1}.S2_{C,2} \\
S2_{C,2} &= \overline{m_2}.S2_{C,3} + \overline{m_1}.S2_{C,2} \\
S2_{C,3} &= \overline{m_3}.S2_{C,2} + m_4.0 + \overline{m_1}.S2_{C,3} \\
C2_S = S2_{S,1} &\stackrel{\text{def}}{=\!=} m_1.S2_{S,2} \\
S2_{S,2} &= m_2.S2_{S,3} + m_1.S2_{S,2} \\
S2_{S,3} &= \tau.S2_{S,4} + m_3.S2_{S,2} + m_1.S2_{S,3} \\
S2_{S,4} &= \overline{m_4}.0
\end{aligned}
\tag{6.9}
$$

```
1  @InitialState(A)
2  clientclass Client3 {
3      @State(A)
4      public void loadDataAndStartMining(Data d) {
5      s.m1(d);
6      s.m2();
7      to(B);
8      }
9
10     @State(B)
11     public void addDataDuringMining(Data d) {
12     if (s.m1 callable) {
13         s.m1(d);
14     } else {
15         s.m3();  // Cancels the data mining
16         s.m1(d); // then we adds data
17         s.m2();  // and finally restarts data mining.
18     }
19     }
20
21     @State(B)
22     @Scope(s)
23     public void m4(Result r) {
24     // Some use of r, for example, save data locally.
25     to(C);
26     }
27 }
```

Figure 6.7: A client class compatible with both $C1$ and $C2$

Fig. 6.7 shows a more complex client class that is compatible with both $C1$ and $C2$. A formal verification is provided on appendices 6.7.1 and 6.7.2. The equivalent π-calculus expression is:

$$
\begin{aligned}
C_i = A &= \overline{m_1}.\overline{m_2}.B \\
B = \overline{\vartheta_{m1}}\langle\vartheta_T, \vartheta_F\rangle.(\vartheta_T.\overline{m_1} + \vartheta_F.\overline{m_3}.\overline{m_1}.\overline{m_2}).B + m_4.C &\\
C &= 0
\end{aligned}
\tag{6.10}
$$

The easiest way to implement a client that is simultaneously compatible with both $C1$ an $C2$ would be to refrain from calling m_1 while the data mining process is executing, but this would underuse a capability of version $C2$. The client on Fig. 6.7 adapts to each version. Boolean statements in if-blocks cannot be written based on both service states and local variables. A local variable adds uncertainty to the Boolean expression, that prevents us from creating an equivalent π-calculus expression of the source code.

Despite of what the expressions above may suggest, verifying the service state during run time does not necessarily imply sending a message to the server. If the current state is deterministic from the client's point

of view, there is no need to communicate with the service in order to determine the current state. Sources of non-determinism are τ and sums. For instance, let us say that the current service state is the following:

$$m_2.m_3 + m_2.m_4 \mid m_1.\tau.\overline{m_2} \tag{6.11}$$

This state is stable, i.e., will not change in a way that does not require interaction with the client and there is no possible internal reaction. This state has three observable actions: two m_2 actions and one m_1 action. If the client observes m_2 (by calling it), then the contract states that either m_3 or m_4 will become observable. So either the server can inform the client which of these two paths it took, or the client can request such data from the server right after observing $\overline{m_2}$. Therefore, although the client cannot determine the next state, it is guaranteed that the next state will be available immediately after the completion of m_2.

The situation is quite different if the client chooses to call m_1. Firstly, a τ starts to execute, which puts the service in a state in which the service can make a transition to another state any time without the client being aware of this change. Secondly, this unobservable transition can lead to two possible states: one in which m_3 is observable, and another in which m_4 is observable. Again, the client has no way to foresee in which of these three states the server currently is.

6.4.3 Generic loop blocks

Loops are translated into more complex structures. Again, we need restrictions in order to create internal reactions.

```
1 s.m1();
2 for (String a : collection) {
3     s.m2();
4     s.m3();
5 }
6 s.m4();
```

Figure 6.8: Loop block

$$\begin{aligned} new\{\kappa_{1S}, \kappa_{1E}, \kappa_{2S}, \kappa_{2E}\}\ (\overline{m_1}.\overline{\kappa_{1S}}.\kappa_{1E}.\overline{m_4}) \mid \\ !(\kappa_{1S}.\overline{\kappa_{2S}} + \kappa_{1S}.\overline{\kappa_{1E}}) \mid !(\kappa_{2S}.\overline{m_2}.\overline{m_3}.\overline{\kappa_{1S}}) \end{aligned} \tag{6.12}$$

Each of the three expressions in parallel as an equivalence in the source code. The first expression is the routine that contains the loop. A loop call and return point are emulated by $\overline{\kappa_{1S}}$ and κ_{1E} respectively. The first replication is equivalent to the loop control. Note that from the perspective of the contract, the decision to continue or stop the loop (represented by the plus sign) is simply non-deterministic. The second replication is the content of the loop block.

So we do not try to interpret the control of the loop since the number or iterations can be dependent on data that is provided externally. Because we model loop controls as a non-determinism (therefore the usage of the plus sign), we use the same process to model while-loops.

6.4.4 Java Threads

```
1 s.m1();
2 s.m2();
3 new Thread() { public void run() {
4     s.m5();
5     s.m6();
6 }}.start();
7 s.m3();
8 s.m4();
```

Figure 6.9: Java thread

$$new\{\kappa_1\}\ (\overline{m_1}.\overline{m_2}.\overline{\kappa_1}.\overline{m_3}.\overline{m_4}) \mid (\kappa_1.\overline{m_5}.\overline{m_6}) \tag{6.13}$$

Modeling a Java thread using the π-calculus is trivial since parallel computation is a central mechanism in the π-calculus.

6.4.5 Fork-join blocks

Java does not provide support for fork-join blocks in its syntax, but since here we are extending the syntax of the Java language, we are free to add this feature, which is both a syntactic sugar which compiles into standard Java source code, and a feature that allows for refined verification. Fig. 6.10 shows the syntax of fork-join blocks. The word "fork" becomes a reserved word and marks the beginning of a list of Java blocks. Each block is translated into a new thread and all blocks execute in parallel. The end of the list of blocks represent a synchronization point that waits for all blocks to finish (a join point).

```
1 s.m1();
2 s.m2();
3 fork {
4     s.m3(); s.m4();
5 } {
6     s.m5(); s.m6();
7 }
8 s.m7();
9 s.m8();
```

Figure 6.10: A fork-join example

The π-calculus expression that represents the source code in Fig. 6.10 is:

$$\begin{aligned} new\{\kappa_f, \kappa_j, \kappa_{1S}, \kappa_{1E}, \kappa_{2S}, \kappa_{2E}\}\ & (\overline{m_1}.\overline{m_2}.\overline{\kappa_f}.\kappa_j.\overline{m_7}.\overline{m_8}) \mid \\ & (\kappa_f.\kappa_{1S}.\kappa_{2S} + \kappa_f.\kappa_{2S}.\kappa_{1S}) \mid \\ & (\kappa_{1S}.\overline{m_3}.\overline{m_4}.\overline{\kappa_{1E}}) \mid \\ & (\kappa_{2S}.\overline{m_5}.\overline{m_6}.\overline{\kappa_{2E}}) \mid \\ & (\kappa_{1E}.\kappa_{2E}.\overline{\kappa_j} + \kappa_{2E}.\kappa_{1E}.\overline{\kappa_j}) \end{aligned} \tag{6.14}$$

In this equation, κ_f is the fork call and κ_j is the synchronization (the join point). Actions κ_{nS} and κ_{nE} are respectively the start and the end of the n-th parallel block. Sums represent the uncertainty of the order in which each thread starts and ends.

6.4.6 Monitor object

Another aspect that should be taken into account when extracting a π-calculus expression from a source code is whether methods in the client objects can be called in parallel or if there is anything synchronizing their execution. A monitor object design pattern [45] is a technique aiming at creating an exclusion zone for concurrency. Arguably, this design pattern is native in the Java programming language, in the form of synchronized methods. Two synchronized methods in Java cannot be called at the same time on the same object. For our purposes, this effect is important, as it is capable of changing the way we translate client processes to a π-calculus expression.

For instance, let us consider a client object that has only two synchronized methods: M_1 and M_2. Here each method is modeled as a π-calculus process (using capital letters) since they may not be equivalent to actions. We can represent the interaction between this client with a service S_1 using:

$$!(M_1 + M_2) \mid S_1 \tag{6.15}$$

We use mutually exclusive options (plus signs) instead of non-mutually exclusive options (a pipe sign) to represent the fact that only one of these methods is running at a time. The benefits of having non-concurrent

methods can be easily illustrated if we consider the service to be $S_1 := a.\overline{b}.c$ and the client to be $M_1 := \overline{a}.b.\overline{c}$ $M_2 := b.\overline{a}.\overline{c}$. So a variant of the client that allows concurrent calls of its methods may lead to a deadlock:

$$!(M_1 \mid M_2) \mid S_1 \xrightarrow{(a)}\xrightarrow{(b)} (b.\overline{c} \mid \overline{a}.\overline{c})!(M_1 \mid M_2) \mid c \tag{6.16}$$

The contract may specify that a service has sets of methods that cannot run simultaneously. For instance, if a service consists only of m_1, m_2, m_3, m_4, and m_5, the service contract may define that the pairs m_1 and m_2, and m_3 and m_4 are mutually exclusive. The expression of such contract would be:

$$S_1 :=!(m_1 + m_2) \mid !(m_3 + m_4) \mid !(m_5) \tag{6.17}$$

6.4.7 Object mobility

We also allow for contracts to represent object mobility. An object may be passed from client to service during an interaction, as an argument. We will present this concept using an example.

On yet another version $C3$ of the contract, the client is be able to attach a stream data mining agent a to the service using the service method m_5. After the service accepts the agent, the service calls the method $a : m_1$ on the agent, which initializes the agent. One of the arguments passed with this method call is a channel c for the agent to initiate an asynchronous communication with the service in which the agent algorithm informs the service that the stream data mining is over. The algorithm may make this decision based on some goal provided by the client.

The agent receives all item sets that are sent to the service (even those item sets that are generated by third party remote nodes) through method $a{:}m_2$. The server hosts the agent for a limited period of time. After this period, the service asks the agent to terminate via the method $a{:}m_3$. This call to $a{:}m_3$ can be used by the agent to send to the client a list of the most frequent item sets that arrived in the service during the time in which the agent was attached. The client can also obtain a partial result of the stream data mining from the agent using the $a{:}m_4$ method, while the agent is attached to the service. The agent can asynchronously send the list of frequent item sets to the client using the client method m_6, which is private between the client and the agent.

$$\begin{aligned} C3_C &\stackrel{\text{def}}{=} !(S1_{C,1} \mid S_{CA,1}) \\ S_{CA,1} &\stackrel{\text{def}}{=} new\{m_6, a{:}m_4\}(\overline{m_5}\langle a\rangle.(!m_6 \mid P_A)) \\ P_A &\stackrel{\text{def}}{=} new\{x_1\}a{:}m_1(c).(\tau.\overline{c}.\overline{m_6}.\overline{x_1} \mid !(a{:}m_2 + x_1.0) \mid \\ &\qquad a{:}m_3.\overline{m_6}.\overline{x_1} \mid !(a{:}m_4.\overline{m_6})) \end{aligned} \tag{6.18}$$

P_A represents the agent's behavior. Since the agent is defined by the client, the client can share with the agent some channels that are made private using restrictions. As we can see in the contract above, m_6 is visible only to client and agent by means of a restriction. The agent may also have its own private channels. In the example above, the bound variable x_1 represents the termination of the agent. The service contract $C3_S$ is given by:

$$\begin{aligned} C3_S &\stackrel{\text{def}}{=} S3_{S,1} \stackrel{\text{def}}{=} m_1.S3_{S,2} \mid m_5(x).S3_{SA,2} \\ S3_{SA,2} &\stackrel{\text{def}}{=} x{:}\overline{m_1}\langle c\rangle.!(m_1.x{:}\overline{m_2} + x{:}\overline{m_2} + c.0 + \tau.0). \\ &\qquad x{:}\overline{m_3}.0 \end{aligned} \tag{6.19}$$

The $S3_{SA,2}$ state represents the beginning of service operation in agent attachment mode. Here is what each term in the sum mean:

- $m_1.x{:}\overline{m_2}$ – The service receives an item set from the client and forwards it to the agent.
- $x{:}\overline{m_2}$ – The service receives an item set from another client and forwards it to the agent.
- $c.0$ – The agent sends a notification to the service to declare that the agent will shut down.

- $\tau.0$ – The service shuts down the agent for a reason not known by the agent. The service may ask the agent to shutdown because the service itself will shutdown, or because the lease time in the service is over, among other reasons.

Programmers should assign a class to play the role of a (which is called x in $C3_S$), so this class should have all methods described in the contract. Also, the service should have a method c available for the agent to call.

Agent mobility raises questions regarding security and context transfer. Although we do not address those questions here, we argue that contracts can be used as a way to describe limitations imposed by the remote environment to address security. Context can also be modeled using contracts to specify what the agent can access.

6.5 Discussion

Other researches [52] [53] propose the use of AOP as a means to create a pre-processor that can make a grid version of a local software. The idea is based on the assumption that programmers find it more natural to program local systems, which is also our assumption here. The main difference between our approach and approaches based on AOP is that we have chosen to concentrate interactions between service and client in a single class on the client side. This allows us to isolate compatibility analysis from possible complexities arising from a multi-threaded client.

Applying AOP makes sense in cases in which a concern is spread across several classes, orthogonally to responsibilities or roles, which are usually mapped into classes. Although it can be argued that adherence to a contract –especially one that is based on parallelism constraints, as in our model– may be a cross-cutting concern, we will be hardly able to come out with any conclusive examination if we analyze several classes in order to extract service usage patterns. That is the reason why we created a separation into layers as in Fig. 6.2.

For the best of our knowledge, this research is the first to propose a DSL specifically to enable formal verification of contracts expressed as a process calculus. We have demonstrated that our method is capable of providing guidelines for programmers to create clients that can be checked to correctly interact with a family of services, grouped by their service contracts. Also, we provided details about our DSL, which was build on top of Java, but could have been implemented based on other similar programming languages. In fact, the limitations we imposed on the references that can be accessed by client layer makes it possible to use one programming language on this layer while choosing other programming languages for other layers.

Although our method is specific to service contracts that are specified in terms of a process calculus equivalent to the π-calculus, our results can be easily used in other contexts. With the help of the formalization we developed, it is possible to create prototypes of complex clients and services and check them formally.

Our proposed architecture exposes service state, which is not compatible with the generally accepted level of opacity for services. It is desirable that contracts are designed in a way that prevents services from having to expose their states. When a service state changes as a result of some internal action inside of the service (for instance, when a thread inside of the process finishes, or as a result of the interaction between the service and a resource within the server), it is a good practice to notify the client using a message from the server, so the client can keep track of the service state without having to explicitly access it.

However, client notifications leave complexity for the client to deal with. The π-calculus, on the other hand, uses the idea of observation. Interaction between complementary channels a and $\overline{a}$, together with invisible actions τ are the way in which process execution advances. Here we use an imperative programming model, in which interaction happens not because an opportunity of interaction presented itself, but as a result of active execution of a coordinator. Therefore, we need an imperative for an active way for active processes to evaluate the possibility of interaction. The Θ context offers such feature.

Also, we do not suggest that all details about the service should be exposed to the client side, but only the states that are part of the contract. So the service is still opaque regarding all information that is not necessary to the client.

6.6 Conclusions

On this paper we presented an outline for service contracts that aims at allowing formal verification of service and client interaction. We also propose a DSL specifically designed to provide a concise programming model

for clients and services. At the same time, features of the DSL (more specifically, the limitations it imposes to programmers and special syntax that refers to service state) make it possible to translate source code into π-calculus expressions for formal verification. A general purpose source code could not be analyzed the same way, as elements that are unrelated to the service contract (such as variables in if clauses and concurrency added by the use of threads) usually disrupts verifications. Our main contribution is an architecture to make these analyzes possible. With little adaptation, our contribution can be directly applied to the general case of distributed services, by allowing developers to reason their distributed systems in terms of the layers we propose on this paper. Although our implementation is based on the Java programming language, if mobile agents are not used, it is possible to adapt our approach to the interaction of heterogeneous systems (in which client and service are based on different platforms).

6.7 Appendix – A formalization of the equivalent expression to verify simulations: Θ

In this appendix we formally define Θ to offer a rationale to support the claim that Θ is an application of π-calculus, rather than a proper extension of it. The π-calculus defines very primitive building blocks, so describing an algorithm using it requires a long expression, which we will define by parts. We will also omit some details when presenting examples of usage of Θ to check compatibility between the pairs $(C_i, C1_C)$ and $(C_i, C2_C)$. C_i was defined in (6.10), $C1_C$ in (6.2), and $C2_C$ in (6.9).

First we define the linking operator that connects two processes P and Q, which is based on the one defined in [71], Section 4.4:

$$P \frown Q \stackrel{\text{def}}{=\!=} \{new\ x\}(\{^{x}/_{right}\}P \mid \{^{x}/_{left}\}Q) \tag{6.20}$$

Channels $right$ and $left$ are made private to the link. We now define a linked list that represents the number of copies of a certain channel available to be called. This list is also based on a construct defined in [71], Section 7.5.

$$\begin{aligned} Empty(c,i,d) &\stackrel{\text{def}}{=\!=} i.(Cell(c,i,d) \frown Empty) + \\ &\quad c(\vartheta_T, \vartheta_F).\overline{\vartheta_F}.Empty(c,i,d) \\ Empty &\stackrel{\text{def}}{=\!=} left(c,i,d).Empty(c,i,d) \\ Cell(c,i,d) &\stackrel{\text{def}}{=\!=} i.(Cell(c,i,d) \frown Cell) + \\ &\quad d.\overline{right}\langle c,i,d\rangle.0 + c(\vartheta_T, \vartheta_F).\overline{\vartheta_T}.Cell(c,i,d) \\ Cell &\stackrel{\text{def}}{=\!=} left(c,i,d).Cell(c,i,d) \end{aligned} \tag{6.21}$$

$Empty(c,i,d)$ is a list that stores the value zero. Calling the channel i causes the list to increase one $Cell$. Calling d removes one $Cell$ until the list is only the $Empty$ element.

The channel c is used to query if the list has any element or not. The list reacts by calling either $\overline{\vartheta_T}$ or $\overline{\vartheta_F}$ to respond with a true or false, respectively. For a short notation, we use:

$$\begin{aligned} \varphi_x^{(0)} &= Empty_x(\vartheta_x, \sigma_{x,INC}, \sigma_{x,DEC}) \\ \varphi_x^{(n)} &= \underbrace{Cell(\vartheta_x, \sigma_{x,INC}, \sigma_{x,DEC}) \frown Cell_x \frown \ldots}_{n\ times} \frown Empty_x \end{aligned} \tag{6.22}$$

Now we can define a context for a contract implementation, which differs from a contract in that an implementation may interact with a reflection of the current service state. An implementation may use the current state of a service to control its execution flow. We need to model this behavior using the π-calculus. A context $\Theta(C_i, C_C)$ for a contract implementation C_i, based on a client contract C_C is given by:

$$\begin{aligned}
\Theta(C_i, C_C) &\stackrel{\text{def}}{=\!=} new\{\vec{m}, \vec{m''}\}\ C_i \mid U(C_C) \mid Trans(C_i) \mid \Phi(C_C) \\
Trans(C_i) &\stackrel{\text{def}}{=\!=} \overline{m1}.m1'.\overline{m1''} \mid \ldots \mid \overline{mn}.mn'.\overline{mn''} \\
\Phi(C_C) &\stackrel{\text{def}}{=\!=} \varphi^{(0)}_{m1} \mid \ldots \mid \varphi^{(0)}_{mn} \\
\vec{m} &\stackrel{\text{def}}{=\!=} m1, \ldots, mn \\
\vec{m'} &\stackrel{\text{def}}{=\!=} m1', \ldots, mn' \\
\vec{m''} &\stackrel{\text{def}}{=\!=} m1'', \ldots, mn''
\end{aligned} \tag{6.23}$$

Where $\vec{m}$ are the channels in C_i or C_C.

$U(C_C)$ is a process that controls the counters φ for each channel availability at any time, according to the changes in C_i. On $U(C_C)$, each channel instance x in C_C is represented by x''. Whenever x'' becomes available a counter increase $\vartheta_{x,INC}$ should be called, and each channel instance that becomes unavailable should result in a call to $\vartheta_{x,DEC}$. Each sum of sequences $x_{1,1}.x_{1,2}\ldots + \ldots + x_{n,1}.x_{n,2}\ldots$ is replaced by:

$$\begin{aligned}
&(\overline{\sigma_{x1,INC}}.\ldots.\overline{\sigma_{xn,INC}}). \\
&(x''_{1,1}.\overline{\sigma_{X,DEC}}.\overline{\sigma_{x12,INC}}.x''_{1,2}.\overline{\sigma_{x12,DEC}}\ldots \mid \\
&\qquad\qquad\vdots \\
&\mid x''_{n,1}.\overline{\sigma_{X,DEC}}.\overline{\sigma_{xn2,INC}}.x''_{n,2}.\overline{\sigma_{xn2,DEC}}\ldots)
\end{aligned} \tag{6.24}$$

where $\overline{\sigma_{X,DEC}} = \overline{\sigma_{x11,DEC}}.\overline{\sigma_{x21,DEC}}\ldots\overline{\sigma_{xn1,DEC}}$, which decreases the counters of all channels that are the first of each sequence in the sum.

$\vec{m'}$ are fresh channels that are observable from anything in parallel with Θ , and $\vec{m''}$ are fresh channels to provide interaction between C_i and $U(C_C)$. $Trans(C_i)$ provides a translation between actions in C_i, C_C, and the external world. Finally, Φ is a set of counters to store the visibility of each channel in $\vec{m}$.

Fig. 6.11 shows how each part of Θ communicate. C_i is free to call whatever channel in $Trans$ through $\vec{m}$. Whenever a channel m_x is called by C_i, $Trans$ calls the equivalent channel m''_x on $U(C_C)$ and m'_x becomes externally observable. When m''_x is called on $U(C_C)$, this call C_i can also call any of ϑ_x in Φ to check if a certain channel x is available to be called. $Trans$ communicates with the outside world through $\vec{m'}$.

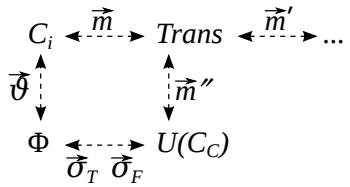

Figure 6.11: Communications between each part of Θ

It is interesting to notice that Θ is define only based on the client portion of the contract, not the service side. We now proceed with two examples of usage of Θ.

6.7.1 Compatibility example: C_i and $S1_C$

We are now ready to show a full example using Θ to check compatibility of the implementation in Fig. 6.7 against versions 1 and 2 of the data mining service.

$$\begin{aligned}
\Theta(C_i, C1_C) = new\{\vec{m}, \vec{m''}\}\ \overline{m_1}.\overline{m_2}.B \mid \\
\overline{m''_1}.\sigma_{m1,F}.U(S1_{C,2}) \mid Trans(C_i) \mid \varphi^{(1)}_{m1} \mid \varphi^{(0)}_{m2} \mid \varphi^{(0)}_{m3} \mid \varphi^{(0)}_{m4}
\end{aligned} \tag{6.25}$$

At follows we list all reactions from $\Theta(C_i, C1_C)$. All reactions are either internal or external ones in terms of $\vec{m'}$, which are the only observable actions from outside of Θ, as we stated above. We omit restrictions, each $\sigma_{x,F}$ after each channel x'', $Trans$, and represent the list of counters $\varphi^{(a)}_{m1} \mid \varphi^{(b)}_{m2} \mid \varphi^{(c)}_{m3} \mid \varphi^{(d)}_{m4}$ as $\varphi^{(a,b,c,d)}$ for a concise representation. We name the states of $\Theta(C_i, C1_C)$ using $S_{1,1}, S_{1,2}, \ldots$.

$$\begin{aligned}
\Theta(C_i, C1_C) &\xrightarrow{m'_1} = S_{1,1} \\
S_{1,2} = \overline{m_2}.B \mid \overline{m''_2}.U(S1_{C,3}) + \overline{m''_1}.U(S1_{C,2}) &\mid \\
&\varphi^{(1,1,0,0)} \xrightarrow{m'_2} \\
S_{1,3} = B \mid \overline{m''_3}.U(S1_{C,2}) + m''_4.0 &\mid \varphi^{(0,0,1,1)} \xrightarrow{(\vartheta_{m1})} \\
&(\vartheta_T.\overline{m_1} + \vartheta_F.\overline{m_3}.\overline{m_1}.\overline{m_2}).B \mid \\
&\overline{m''_3}.U(S1_{C,2}) + m''_4.0 \mid \varphi^{(0,0,1,1)} \xrightarrow{(\vartheta_F)} \\
\overline{m_3}.\overline{m_1}.\overline{m_2}.B \mid \overline{m''_3}.U(S1_{C,2}) + m''_4.0 &\mid \varphi^{(0,0,1,1)} \xrightarrow{m'_3} \\
S_{1,4} = \overline{m_1}.\overline{m_2}.B \mid U(S1_{C,2}) &\mid \varphi^{(1,1,0,0)} \xrightarrow{m'_1} \\
S_{1,5} = \overline{m_2}.B \mid U(S1_{C,2}) &\mid \varphi^{(1,1,0,0)} \xrightarrow{m'_2} \\
S_{1,3} &\xrightarrow{m_4} 0
\end{aligned} \tag{6.26}$$

Therefore we can say that $\Theta(C_i, C1_C) \approx C1_C$ using the relation $\Theta(C_i, C1_C)RC1_C$:

$$\begin{aligned}
R = \{&(S_{1,1}, S1_{C,1}), (S_{1,2}, S1_{C,2}), \\
&(S_{1,3}, S1_{C,3}), (S_{1,4}, S1_{C,2}), (S_{1,5}, S1_{C,2}), (0,0)\}
\end{aligned} \tag{6.27}$$

Note that C_i has the potential to generate more states from S_2 if put in another context, which were avoided by a response ϑ_F from $\varphi^{(0)}_{m1} = Empty_{m1}$.

6.7.2 Compatibility example: C_i and $S2_C$

We can see these states if we try to check the compatibility between C_i and $C2_C$, as follows. The names of states from $\Theta(C_i, C2_C)$ are $S_{2,1}$, $S_{2,2}$, and $S_{2,3}$.

$$\begin{aligned}
\Theta(C_i, C2_C) = S_{2,1} &\xrightarrow{m'_1} \\
S_{2,2} = \overline{m_2}.B \mid \overline{m''_2}.U(S2_{C,3}) + \overline{m''_1}.U(S2_{C,2}) &\mid \\
&\varphi^{(1,1,0,0)} \xrightarrow{m'_2} \\
S_{2,3} = B \mid \overline{m''_3}.U(S2_{C,2}) + m''_4.0 + &\\
\overline{m''_1}.U(S2_{C,3}) \mid \varphi^{(1,0,1,1)} &\xrightarrow{(\vartheta_{m1})} \\
(\vartheta_T.\overline{m_1} + \vartheta_F.\overline{m_3}.\overline{m_1}.\overline{m_2}).B &\mid \\
\overline{m''_3}.U(S2_{C,2}) + m''_4.0 + \overline{m''_1}.U(S2_{C,3}) \mid \varphi^{(1,0,1,1)} &\xrightarrow{(\vartheta_T)} \\
\overline{m_1}.B \mid \overline{m''_3}.U(S2_{C,2}) + m''_4.0 + \overline{m''_1}.U(S2_{C,3}) &\mid \\
\varphi^{(1,0,1,1)} \xrightarrow{m'_1} S_{2,3} &\xrightarrow{m'_4} 0
\end{aligned} \tag{6.28}$$

The relation R to prove that C_i weakly simulates $C2_C$ is:

$$R = \{(S_{2,1}, S2_{C,1}), (S_{2,2}, S2_{C,2}), (S_{2,3}, S2_{C,3}), (0,0)\} \tag{6.29}$$

Both (6.27) and (6.29) contain the $(0,0)$ pair, meaning both relations can be said to reach a simultaneous termination state.

Chapter 7

β channels: a π-calculus extension to one-to-many messages

Abstract

The π-calculus is a well-established formalization to model and reason distributed systems. There are several variants of the π-calculus but none of them was specifically designed to enable communication in a one-to-many fashion. This paper introduces a new kind of channel called β to the π-calculus in which an output channel interacts with a number of input channels. β interactions are useful in a number of applications in which messages are exchanged in multicasting. In this paper we introduce a formalization of β channels in which those channels are translated into standard π-calculus expressions. We also present three examples of usage of β channels that go beyond message multicasting, to model: cyclic barriers, passing references to services, and a MapReduce use case.

7.1 Introduction

Formal methods provide us frameworks to reason about events that we observe in systems, and the structures we create. Although π-calculus is a well-established formalization for distributed computing, for the best of our knowledge, the proposed extensions of the π-calculus do not represent one-to-many communication. In this paper we introduce a new construct we call β that tries to fill this gap.

One-to-many communication is one of the main building blocks of network applications such as streaming and is at the core of non-centralized agent topologies, such as peer-to-peer, grid, or cloud computing. A single message reaching multiple destinations in a transparent way for the sender is useful whenever an agent sends data not to another single agent, but to a group of agents.

In a cloud computing environment, for instance, a client may request a service without knowing which service agent, in a list of hundreds of available agents, will in fact respond to service calls. In such a case, a client may have, at the beginning, only a reference (a β) to a group of agents, not to one specific agent.

We also need to discuss the meaning of simultaneity. In the π-calculus there is simultaneity in a channel reaction, but what does that actually mean? In reality, interactions are hardly simultaneous. A message transfer over the network usually does not happen in a way that is completely simultaneous to sender and receiver. Instead, there is a certain difference in timing that is not relevant in any sense, so we model it as a simultaneity. What the π-calculus does not do is to allow for more than two processes to share the same simultaneous-enough event.

In this paper we present a formal definition of the β-calculus Section 7.2. We argue that the β channels as a shorthand for a complex π-calculus process that we describe in details. We want to demonstrate that β channels described this way do not impose new fundamental constructs to the π-calculus, so that all reasoning valid in π-calculus are also valid with the introduction of β. We present a detailed multicasting example of processing a β interaction in Section 7.3.

On Section 7.4 we show three other applications of β channels: to implement a cyclic barrier to synchronize an undefined number of threads; for service references to passed around, allowing for clients to communicate with several instances of a certain service at once; and to model a MapReduce use case.

Finally, on Section 8.8 presents our final remarks and the contributions this paper has made.

7.2 A β notation in pure π-calculus

Introducing the basic concepts of π-calculus is out of the scope of this text. For a detailed introduction to the π-calculus we refer the reader to [21, 54, 71]. Here we use the notation [71]. A broad overview of research related to the π-calculus can be found on [35].

7.2.1 Expected behavior or β channels

Let us begin by defining a reaction rule $REACT_\beta$ for β channels:

Definition 7.1 ***Reaction rule for β channels.*** *Let $P_0, P_1, \ldots, P_N$ and $M_0, M_1, \ldots, M_N$ be processes in standard π-calculus. All reactions in terms of β channels are relations expressed by the rule:*

$$REACT_\beta : (\overline{\beta}.P_0 + M_0) \mid (\beta.P_1 + M_1) \mid \ldots \mid (\beta.P_N + M_N) \rightarrow P_0 \mid P_1 \mid \ldots \mid P_N \tag{7.1}$$

To improve readability, we also add round brackets around channel names to explicitly state which channel was responsible for the reaction. This kind of arrow differs from a labeled transition, represented without round brackets, in that labeled transitions represents the capacity or potential to a reaction, not a reaction that takes place. The following examples illustrate the difference between these two kinds of arrows:

$$\overline{\beta_X}.P_0 + M_0 \xrightarrow{\overline{\beta_X}} P_0 \tag{7.2}$$

$$(\beta_X.P_1 + M_1) \mid \ldots \mid (\beta_X.P_N + M_N) \xrightarrow{\beta_X} P_1 \mid \ldots \mid P_N \tag{7.3}$$

$$(\overline{\beta_X}.P_0 + \overline{\beta_Y}Q_0) \mid (\beta_X.P_1 + \beta_Y.Q_1) \mid \ldots \mid (\beta_X.P_N + \beta_Y.Q_N) \xrightarrow{(\beta_Y)} Q_0 \mid Q_1 \mid \ldots \mid Q_N \tag{7.4}$$

Equations (7.2) and (7.3) are labeled transitions by means of β_X and $\overline{\beta_X}$ respectively. Both represent the potential of a process to interact with another process put in parallel and change as a result of such interaction. Equation (7.2) means that $\overline{\beta_X}.P_0 + M_0$ is able to send data to at least one β at the same time, while equation (7.3) means that $(\beta_X.P_1 + M_1) \mid \ldots \mid (\beta_X.P_N + M_N)$ is able to receive data from exactly one $\overline{\beta}$.

Equation (7.4) is an actual reaction by means of the complementary pair of channels β_Y and $\overline{\beta_Y}$. We represent the pair using (β_Y). In this example the label on top of the arrow is not mandatory but present only for clarity, since another possible reaction would be in terms of the pair (β_X).

β channels can also be restricted. The restriction rule is almost the same as the one introduced for the π-calculus in [71]. Here we represent the restricted channels within curly brackets for readability.

Definition 7.2 ***Restriction rule for β channels*** *Let P and P' be processes. We define the restriction rule for β channels as:*

$$RES_\beta : \frac{P \rightarrow P'}{new\{\beta\}P \rightarrow new\{\beta\}P'} \tag{7.5}$$

We can also use reactions by means of β channels to pass values from an output channel $\overline{\beta_X}$ to an input channel β_X. In the equation bellow a single interaction sends the polyadic $\vec{d_4}$ to three receiving processes. Each of these three processes performs its own alpha conversion after receiving $\vec{d_4}$.

$$\beta(\vec{x_1}).Q_1 \mid \beta(\vec{x_2}).Q_2 \mid \beta(\vec{x_3}).Q_3 \mid \overline{\beta}\langle\vec{d_4}\rangle.Q_4 \rightarrow \{^{\vec{d_4}}/_{\vec{x_1}}\}Q_1 \mid \{^{\vec{d_4}}/_{\vec{x_2}}\}Q_2 \mid \{^{\vec{d_4}}/_{\vec{x_3}}\}Q_3 \mid Q_4 \tag{7.6}$$

β channels are such that the reaction above is possible in a single step or, as we will see later, in several steps. Before we present the equivalent expression to a β channel we need to introduce a few building blocks.

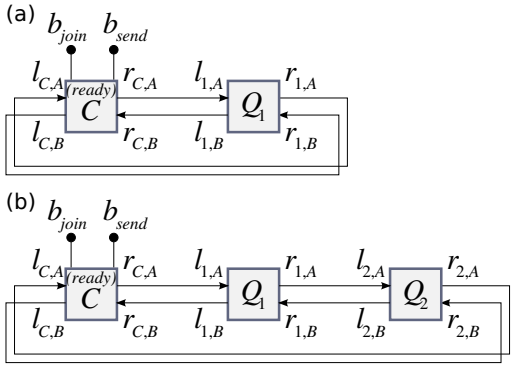

Figure 7.1: The ring structure of a β channel. (a) A situation in which only the process Q_1 is waiting for messages from the β channel. The controller C manages the ring and the channels $b_{x,join}$ and $b_{x,send}$. (b) A state in which processes Q_1 and Q_2 are waiting for messages.

7.2.2 Expressing β channels using π-calculus

The structure of a β channel that we introduce in this section is derived from the linking operator in [71], and the structure of a scheduler (Section 7.3). Figure 7.1 illustrates the structure of a β channel.

Figure 7.1 (a) bla bla 7.1 is the state of a β channel with just one process listening to messages. This is equivalent to the expression $\beta.Q_1$. We need an extra process $C^{(ready)}$, a controlling node, to manage channel subscription (the b_{join} channel) and sending messages to the channel (the b_{send} channel).

It would be also possible for nodes $\beta.Q_1, \ldots, \beta.Q_n$ to organize themselves without the need of an extra process C. One of the nodes would need to become the controller node, doing what C does in the solution we present in this paper. We refrain from using such solution, which arguably can in fact be implemented in real systems, because it imposes the creation of a protocol to avoid problems that arise from parallelism between processes. For instance, the lack of a previously existing central control allows for more than one node to mistakenly take the role of C at the same time. So we would need to create a protocol to avoid this from happening. Such algorithm implementation in π-calculus can be done but, without extending the π-calculus, it can only be expressed using very long expressions, which is beyond the purpose of this paper.

Here we want to prove the feasibility of β channels in π-calculus, so we will avoid the non-centralized approach. Besides, the approach we present here, one that relies on a dedicated process, is a realistic one as it can be directly mapped into message queue services that are usually at the core of scalable enterprise architectures.

Before we present the expressions of each kind of node, we define the following vectors which we use to keep our expressions concise:

$$\begin{aligned} \vec{b} &= (b_{join}, b_{send}) \\ \vec{l} &= (l_A, l_B) \\ \vec{r} &= (r_A, r_B) \end{aligned} \tag{7.7}$$

$\vec{b}$ are the channels used to communicate with the external world. b_{join} is used by external process to join the β channel as a listener process. There is no subscription process for senders since senders typically connect to the β channels only for the time it takes to send the message and leaves the channel right after that. Senders use the b_{send} channel to transfer data, which represent with $\vec{x}$.

Channels $\vec{l}$ and $\vec{r}$ represent links between a certain node (be it the controller C node of the participant nodes Q in Fig. 7.1) and its left and right neighbors respectively. We define two parallel layers A and B in the ring. Layer A is used to transfer data $\vec{x}$ between nodes in the clockwise direction. Layer B is used by nodes to reconfigure the ring, allowing for nodes to be added or removed from it. Messages in the B layer travel in the counter-clockwise direction, as represented by the arrows in Fig. 7.1. We are now ready to define the controller and participant nodes.

Definition 7.3 ***Controller node.*** *A controller node C of a β channel is defined by the equations bellow, in which we initially set the state to $C(\vec{b}) = C^{(init)}(\vec{b})$:*

$$
\begin{aligned}
C^{(init)}(\vec{b}) &\stackrel{def}{=\!=} new\{\vec{l_C}\}\ b_{join}(\vec{l},\rho).\overline{\rho}\langle\vec{l_C}\rangle.C^{(ready)}\langle\vec{b},\vec{l}\rangle \\
C^{(ready)}(\vec{b},\vec{r}) &\stackrel{def}{=\!=} new\{\vec{l_C}\}\ r_B(\vec{r'}).C^{(ready)}\langle\vec{b},\vec{r'}\rangle + b_{join}(\vec{l},\rho).\overline{l_{C,B}}\langle\vec{l}\rangle.\overline{\rho}\langle\vec{l_C}\rangle.C^{(ready)}\langle\vec{b},\vec{r}\rangle\ + \\
&\qquad b_{send}(\vec{x}).C^{(busy)}(\vec{b},\vec{r}) \\
C^{(busy)}(\vec{b},\vec{r}) &\stackrel{def}{=\!=} \overline{r_A}\langle\vec{x}\rangle.C^{(busy)}\langle\vec{b},\vec{r}\rangle + l_{C,A}(\vec{x},\vec{l'}). \\
&\quad if(\vec{l'}=\vec{r})then(r_b(\vec{r'}).C^{(init)}\langle\vec{b}\rangle)else(\overline{r_A}\langle\vec{x},\vec{l_C}\rangle.C^{(busy)}\langle\vec{b},\vec{r}\rangle)
\end{aligned}
\tag{7.8}
$$

The $C^{(init)}$ state is one in which the controller only allows for channel subscription via the b_{join} channel. After at least one listener participant joined the β channel, the b_{send} channel becomes available for a sender participant node to send messages. Interactions by means of the b_{send} and $\overline{b_{send}}$ pair of channels are possible only on the $C^{(ready)}$ state, after channel initialization.

The controller node goes to the $C^{(busy)}$ state after an interaction with a sender node. After the message was delivered to all listener participants, the β channel ring becomes empty again and the controller node goes back to the $C^{(init)}$ state.

The specific β channel is represented by $\vec{b}$. Therefore, at a given time, there should never be two controller nodes bound to the same $\vec{b}$ name. We need then to define constraints for the usage of C.

Definition 7.4 ***Well-formed system in terms of controller nodes.*** *Let S be a system in the π-calculus in the form $Q_1 \mid \ldots \mid Q_n$, where $Q_1,\ldots,Q_n$ are called components and are summations. A well-formed system in terms of C is one that observes the following restrictions:*

- *Controller nodes C can only appear in S if they are one of the components.*
- *No two controller nodes should be parametrized using the same $\vec{b}$.*

The first rule states that controller nodes can not be part of a component. This is important to make sure that a controller node actually behaves as an independent node. Therefore, its expression is not hidden by another channel, as in "$\overline{a}.C\langle\vec{b}\rangle$". Also, container node processes never die, so it does not make sense to have a component whose expression follows the patterns "$C\langle\vec{b}\rangle.a$" or "$C\langle\vec{b}\rangle + P$", where a is a channel and P is a sequence.

The second rule ensures that each β channel is represented by a single controller node. So "$C\langle\overline{b_1}\rangle \mid C\langle\overline{b_2}\rangle$" is a well-formed system, while "$C\langle\vec{b_1}\rangle \mid C\langle\vec{b_1}\rangle$" is not.

Definition 7.5 ***Listener participant node.*** *A listener participant node N to a β channel is a node that intends to receive messages from this channel. Listener participants are defined by the following equations, in which we set the initial state to be $N(\vec{b},\vec{x}) = N^{(init)}(\vec{b},\vec{x})$:*

$$
\begin{aligned}
N^{(init)}(\vec{b},\vec{x}) &\stackrel{def}{=\!=} new\{\rho,\vec{l}\}\ \overline{b_{join}}\langle\vec{l},\rho\rangle.\rho(\vec{r}).N^{(ready)}\langle\vec{l},\vec{r},\vec{x}\rangle \\
N^{(ready)}(\vec{l},\vec{r},\vec{x}) &\stackrel{def}{=\!=} r_B(\vec{r'}).N^{(ready)}\langle\vec{l},\vec{r'},\vec{x}\rangle + l_A(\vec{x},\vec{l'}).\overline{r_A}\langle\vec{x},\vec{l}\rangle.(\overline{l_B}\langle\vec{r}\rangle + N^{(ready)}\langle\vec{l},\vec{r},\vec{x}\rangle)
\end{aligned}
\tag{7.9}
$$

Listener participant nodes start in the $N^{(init)}$ state, in which it requests to join the β channel as a listener. ρ is a private channel that a listener participant sends to the controller node in order to get a reference to the its new right neighbor. When a listener is on the $N^{(ready)}$ state, it is capable of either receive a reconfiguration signal or receive data. Reconfiguration comes from $r_B(\vec{r'})$, where $\vec{r'}$ is the new right neighbor, while data comes from $l_A(\vec{x},\vec{l'})$.

Definition 7.6 ***Equivalent expression to a listener participant node process.*** *Let a listener participant process P have the form $P = \beta(\vec{x}).P'$ where P' is a process in the π-calculus, including the empty process. Also, let the controller node of the β channel be $C\langle\vec{b}\rangle$. The equivalent expression to P in the π-calculus is $N^{(ready)}\langle\vec{l},\vec{r},\vec{x}\rangle.P'$, where $N^{(ready)}\langle\vec{l},\vec{r},\vec{x}\rangle$ is obtained as the series of reactions $N(\vec{b},\vec{x}) \mid C\langle\vec{b}\rangle \rightarrow^* N^{(ready)}\langle\vec{l},\vec{r},\vec{x}\rangle$, and $\vec{r}$ is the right neighbor's left channel received from the controller node.*

This definition states that a listener participant node can be considered in a reaction as specified in equation (7.1) only after the controller node caused the listener participant node to reach the $N^{(ready)}$ state. Between the $N^{(init)}$ and the $N^{(ready)}$ states listener participant nodes go through the intermediate state $\text{new}\{\rho,\vec{l}\}\ \rho(\vec{r}).N^{(ready)}\langle\vec{l},\vec{r},\vec{x}\rangle$, but we should not be concerned about such state since this is a transitory state that will necessarily lead the listener node to the $N^{(ready)}$ state.

Definition 7.7 ***Sender participant node.*** *A sender participant node to a β channel is a node that intends to send messages to a β channel. Sender participant nodes are defined by an output channel $\overline{b_{send}}$ followed by a process P, which can be empty: $\overline{b_{send}}.P$. Sender participant nodes are represented using $\overline{\beta}.P$.*

All complexity is on the controller and listener participant nodes. That is why the sender participant node does not require any especial treatment.

Proposition – On a well-formed system having a controller node for a β channel called β_X, listener and sender participant nodes observe $REACT_\beta$ for β_X.

7.3 Example

In this section we show an example of our proposal. We will simulate a system that begins with a controller node. Figure 7.2 shows the setup process from a state in which only the controller exists (Fig. 7.2-a) until three participant nodes were added (Fig. 7.2-d). Listener participant nodes $\overline{\beta}\langle\vec{x}\rangle.Q_1$, $\overline{\beta}\langle\vec{x}\rangle.Q_2$, and $\overline{\beta}\langle\vec{x}\rangle.Q_3$ are then added to the system one by one.

In Fig. 7.2 we chose to use only names of channels on the left of processes ($\vec{l}$, l_A, and so on). We will use this convention when we describe processes in this example. Because our structure is of two rings, we do not need to use names referring to channels on the right as all channels are also represented as a channel on the left of some process.

After this setup phase, all three listener nodes are ready to receive a single message from the β channel. We will use the following equivalence from β-calculus to standard π-calculus:

$$\overline{\beta}\langle\vec{x}\rangle.Q_1 \mid \overline{\beta}\langle\vec{x}\rangle.Q_2 \mid \overline{\beta}\langle\vec{x}\rangle.Q_3 \equiv C\langle\vec{b}\rangle \mid N\langle\vec{b},\vec{x}\rangle.Q_1 \mid N\langle\vec{b},\vec{x}\rangle.Q_2 \mid N\langle\vec{b},\vec{x}\rangle.Q_3 \tag{7.10}$$

We then calculate the chain of reactions that take place in case the system is put to interact with a process that exposes a $\overline{b_{send}}$ output channel. In other words, we calculate the chain of reactions from the following labeled reaction:

$$C^{(ready)} \mid Q_1 \mid Q_2 \mid Q_3 \xrightarrow{b_{send}} \tag{7.11}$$

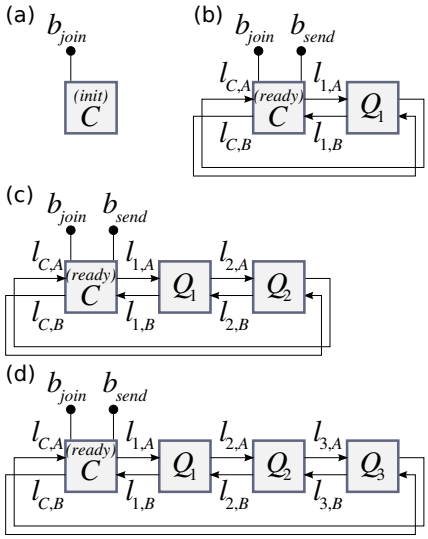

Figure 7.2: A simplified representation of the ring channels in which only the left channels are represented. (a) Initial state, with no processes attached. (b) $\beta.Q_1$ (c) $\beta.Q_1 \mid \beta.Q_2$ (d) $\beta.Q_1 \mid \beta.Q_2 \mid \beta.Q_3$

We will use four kinds of arrows on the equations bellow. The first, without any label $\rightarrow$, is a non-observable reaction that takes place because of an option (a plus sign). The second kind of arrow has a label on top of it, as in $\xrightarrow{x}$, and marks a labeled transition, or a hypothesis of what would take place if the current system was put to interact with a process that exposes the complementary channel $\overline{x}$. The third kind of arrow has a label between parenthesis, as in $\xrightarrow{(x)}$, and means a non-observable reaction by means of a pair of

complementary channels x and $\overline{x}$. The fourth kind of arrow represents a non-arbitrary decision that results from processing an if-then-else block, and is written $\rightarrow (*)$.

We start with C and a process Q_1:

$$C\langle\vec{b}\rangle \mid N^{(init)}\langle\vec{b},\vec{x}\rangle.Q_1 \xrightarrow{(b_{join})} \text{new}\{\vec{l_C}\}\ \overline{\rho_1}\langle\vec{l_C}\rangle.C\langle\vec{b},\vec{l_1}\rangle \mid \text{new}\{\rho_1,\vec{l_1}\}\ \rho_1(\vec{r}).N^{(ready)}\langle\vec{l_1},\vec{r},\vec{x}\rangle.Q_1 \xrightarrow{(\rho_1)} C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_C},\vec{x}\rangle.Q_1$$

Then we add Q_2:

$$\begin{aligned} C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_C},\vec{x}\rangle.Q_1 \mid N^{(init)}\langle\vec{b},\vec{x}\rangle &\xrightarrow{(b_{join})} \\ .Q_2\text{new}\{\vec{l_C}\}\ \overline{l_{C,B}}\langle\vec{l_2}\rangle.\overline{\rho_2}\langle\vec{l_C}\rangle.C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_C},\vec{x}\rangle.Q_1 \mid & \\ \text{new}\{\rho_2,\vec{l_2}\}\ \rho_2(\vec{r}).N^{(ready)}\langle\vec{l_2},\vec{r},\vec{x}\rangle.Q_2 &\xrightarrow{(l_{C,B})} \\ \text{new}\{\vec{l_C}\}\ \overline{\rho_2}\langle\vec{l_C}\rangle.C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q_1 \mid & \\ \text{new}\{\rho_2,\vec{l_2}\}\ \rho_2(\vec{r}).N^{(ready)}\langle\vec{l_2},\vec{r},\vec{x}\rangle.Q_2 &\xrightarrow{(\rho_2)} \\ C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q_1 \mid N^{(ready)}\langle\vec{l_2},\vec{l_C},\vec{x}\rangle.Q_2 & \end{aligned}$$

Finally, we finish the setup by adding Q_3:

$$\begin{aligned} C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q_1 \mid N^{(ready)}\langle\vec{l_2},\vec{l_C},\vec{x}\rangle.Q_2 \mid N^{(init)}\langle\vec{b},\vec{x}\rangle.Q_3 &\xrightarrow{(b_{join})} \\ \text{new}\{\vec{l_C}\}\ \overline{l_{C,B}}\langle\vec{l_3}\rangle.\overline{\rho_3}\langle\vec{l_C}\rangle.C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q_1 \mid N^{(ready)}\langle\vec{l_2},\vec{l_C},\vec{x}\rangle.Q_2 \mid & \\ \text{new}\{\rho_3,\vec{l_3}\}\ \rho_3(\vec{r}).N^{(ready)}\langle\vec{l_3},\vec{r},\vec{x}\rangle.Q_3 &\xrightarrow{(l_{C,B})} \\ \text{new}\{\vec{l_C}\}\ \overline{\rho_3}\langle\vec{l_C}\rangle.C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q_1 \mid & \\ N^{(ready)}\langle\vec{l_2},\vec{l_3},\vec{x}\rangle.Q_2 \mid \text{new}\{\rho_3,\vec{l_3}\}\ \rho_3(\vec{r}).N^{(ready)}\langle\vec{l_3},\vec{r},\vec{x}\rangle.Q_3 &\xrightarrow{(\rho_3)} \\ C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q_1 \mid N^{(ready)}\langle\vec{l_2},\vec{l_3},\vec{x}\rangle.Q_2 \mid & \\ N^{(ready)}\langle\vec{l_3},\vec{l_C},\vec{x}\rangle.Q_3 & \end{aligned}$$

At this point the system is at a state in which there is a controller node and three listener participant nodes. We only need a sender participant node in order to complete a system in which a β transition. We then calculate a labeled transition by means of b_{send}. In other words, we evaluate what would take place if a process such as $\overline{\beta}\langle\vec{x}\rangle$ was put in parallel with the system at the current state. To keep our expressions short we will use Q'_1, Q'_2, and Q'_3 to represent Q_1, Q_2, and Q_3 after an alpha conversion to receive $\vec{x}$ from the β channel.

$$C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q_1 \mid N^{(ready)}\langle\vec{l_2},\vec{l_3},\vec{x}\rangle.Q_2 \mid N^{(ready)}\langle\vec{l_3},\vec{l_C},\vec{x}\rangle.Q_3 \xrightarrow{b_{send}}$$
$$\overline{l_{1,A}}\langle\vec{x},\vec{l_C}\rangle.C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q'_1 \mid N^{(ready)}\langle\vec{l_2},\vec{l_3},\vec{x}\rangle.Q_2 \mid N^{(ready)}\langle\vec{l_3},\vec{l_C},\vec{x}\rangle.Q_3 \xrightarrow{(l_{1,A})}$$
$$C\langle\vec{b},\vec{l_1}\rangle \mid \overline{l_{2,A}}\langle\vec{x},\vec{l_1}\rangle.(\overline{l_{1,B}}\langle\vec{l_2}\rangle + N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle).Q'_1 \mid N^{(ready)}\langle\vec{l_2},\vec{l_3},\vec{x}\rangle.Q_2 \mid N^{(ready)}\langle\vec{l_3},\vec{l_C},\vec{x}\rangle.Q_3 \xrightarrow{(l_{2,A})}$$
$$C\langle\vec{b},\vec{l_1}\rangle \mid (\overline{l_{1,B}}\langle\vec{l_2}\rangle + N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle).Q'_1 \mid \overline{l_{3,A}}\langle\vec{x},\vec{l_2}\rangle.(\overline{l_{2,B}}\langle\vec{l_3}\rangle + N^{(ready)}\langle\vec{l_2},\vec{l_3},\vec{x}\rangle).Q'_2 \mid$$
$$N^{(ready)}\langle\vec{l_3},\vec{l_C},\vec{x}\rangle.Q_3 \rightarrow$$
$$C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q'_1 \mid \overline{l_{3,A}}\langle\vec{x},\vec{l_2}\rangle.(\overline{l_{2,B}}\langle\vec{l_3}\rangle + N^{(ready)}\langle\vec{l_2},\vec{l_3},\vec{x}\rangle).Q'_2 \mid N^{(ready)}\langle\vec{l_3},\vec{l_C},\vec{x}\rangle.Q_3 \xrightarrow{(l_{3,A})}$$
$$C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q'_1 \mid (\overline{l_{2,B}}\langle\vec{l_3}\rangle + N^{(ready)}\langle\vec{l_2},\vec{l_3},\vec{x}\rangle).Q'_2 \mid$$
$$\overline{l_{C,A}}\langle\vec{x},\vec{l_3}\rangle.(\overline{l_{3,B}}\langle\vec{r},\vec{l_C}\rangle + N^{(ready)}\langle\vec{l_3},\vec{l_C},\vec{x}\rangle).Q'_3 \rightarrow$$
$$C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q'_1 \mid N^{(ready)}\langle\vec{l_2},\vec{l_3},\vec{x}\rangle.Q'_2 \mid$$
$$\overline{l_{C,A}}\langle\vec{x},\vec{l_3}\rangle.(\overline{l_{3,B}}\langle\vec{l_C}\rangle + N^{(ready)}\langle\vec{l_3},\vec{l_C},\vec{x}\rangle).Q'_3 \xrightarrow{(l_{C,A})}$$
$$\text{if}(\vec{l_3}=\vec{l_1})\text{then}(l_1(\vec{r'}).C\langle\vec{b}\rangle)\text{else}(\overline{l_{1,A}}\langle\vec{x},\vec{l_C}\rangle.C\langle\vec{b},\vec{l_1}\rangle) \mid$$
$$N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q'_1 \mid N^{(ready)}\langle\vec{l_2},\vec{l_3},\vec{x}\rangle.Q'_2 \mid$$
$$(\overline{l_{3,B}}\langle\vec{l_C}\rangle + N^{(ready)}\langle\vec{l_3},\vec{l_C},\vec{x}\rangle).Q'_3 \rightarrow (*)$$

$$\overline{l_{1,A}}\langle\vec{x},\vec{l_C}\rangle.C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q'_1 \mid N^{(ready)}\langle\vec{l_2},\vec{l_3},\vec{x}\rangle.Q'_2 \mid$$
$$(\overline{l_{3,B}}\langle\vec{l_C}\rangle + N^{(ready)}\langle\vec{l_3},\vec{l_C},\vec{x}\rangle).Q'_3 \rightarrow (*)$$
$$\overline{l_{1,A}}\langle\vec{x},\vec{l_C}\rangle.C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q'_1 \mid N^{(ready)}\langle\vec{l_2},\vec{l_3},\vec{x}\rangle.Q'_2 \mid$$
$$\overline{l_{3,B}}\langle\vec{l_C}\rangle.Q'_3 \xrightarrow{(l_{3,B})}$$
$$\overline{l_{1,A}}\langle\vec{x},\vec{l_C}\rangle.C\langle\vec{b},\vec{l_1}\rangle \mid N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle.Q'_1 \mid$$
$$N^{(ready)}\langle\vec{l_2},\vec{l_C},\vec{x}\rangle.Q'_2 \mid Q'_3 \xrightarrow{(l_{1,A})}$$
$$C\langle\vec{b},\vec{l_1}\rangle \mid \overline{l_{2,A}}\langle\vec{x}\rangle.(\overline{l_{1,B}}\langle\vec{l_2}\rangle + N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle).Q'_1 \mid$$
$$N^{(ready)}\langle\vec{l_2},\vec{l_C},\vec{x}\rangle.Q'_2 \mid Q'_3 \xrightarrow{(l_{2,A})}$$
$$C\langle\vec{b},\vec{l_1}\rangle \mid (\overline{l_{1,B}}\langle\vec{l_2}\rangle + N^{(ready)}\langle\vec{l_1},\vec{l_2},\vec{x}\rangle).Q'_1 \mid$$
$$\overline{l_{C,A}}\langle\vec{x},\vec{l_2}\rangle.(\overline{l_{2,B}}\langle\vec{l_C}\rangle + N^{(ready)}\langle\vec{l_2},\vec{l_C},\vec{x}\rangle).Q'_2 \mid Q'_3 \rightarrow (*)$$
$$C\langle\vec{b},\vec{l_1}\rangle \mid \overline{l_{1,B}}\langle\vec{l_2}\rangle.Q'_1 \mid$$
$$\overline{l_{C,A}}\langle\vec{x},\vec{l_2}\rangle.(\overline{l_{2,B}}\langle\vec{l_C}\rangle + N^{(ready)}\langle\vec{l_2},\vec{l_C},\vec{x}\rangle).Q'_2 \mid Q'_3 \xrightarrow{(l_{1,B})}$$
$$C\langle\vec{b},\vec{l_2}\rangle \mid Q'_1 \mid \overline{l_{C,A}}\langle\vec{x},\vec{l_2}\rangle.(\overline{l_{2,B}}\langle\vec{l_C}\rangle +$$
$$N^{(ready)}\langle\vec{l_2},\vec{l_C},\vec{x}\rangle).Q'_2 \mid Q'_3 \xrightarrow{(l_{C,A})}$$
$$\text{if}(\vec{l_2}=\vec{l_2})\text{then}(l_{2,B}(\vec{r'}).C\langle\vec{b}\rangle)\text{else}(\overline{l_{2,A}}\langle\vec{x},\vec{l_C}\rangle.C\langle\vec{b},\vec{l_2}\rangle) \mid$$
$$Q'_1 \mid (\overline{l_{2,B}}\langle\vec{l_C}\rangle + N^{(ready)}\langle\vec{l_2},\vec{l_C},\vec{x}\rangle).Q'_2 \mid Q'_3 \rightarrow (*)$$
$$l_{2,B}(\vec{r'}).C\langle\vec{b}\rangle \mid Q'_1 \mid (\overline{l_{2,B}}\langle\vec{l_C}\rangle + N^{(ready)}\langle\vec{l_2},\vec{l_C},\vec{x}\rangle).Q'_2 \mid Q'_3 \xrightarrow{(l_{2,B})}$$
$$C\langle\vec{b}\rangle \mid Q'_1 \mid Q'_2 \mid Q'_3$$

7.4 Applications of the β-calculus

In this section we present some applications of the β-calculus. Besides the obvious usage to model message multicasting, β channels can also be used to synchronize processes as it can be easily used to create a point in process execution in which many processes wait for a single signal. Another application is to allow for easy expression of the lottery example in [71].

7.4.1 Cyclic barrier

One of the ways to use β channels is to create a synchronization point for parallel executing processes. Given processes $P_1^{(1)}, \ldots, P_n^{(1)}$ and $P_1^{(2)}, \ldots, P_n^{(2)}$, a cyclic barrier for n processes can be build by selecting any of the processes to resume all others by calling a β channel.

$$P_1^{(1)}.a_2.\ldots.a_n.\overline{\beta}.P_1^{(2)} \mid P_2^{(1)}.\overline{a_2}.\beta.P_2^{(2)} \mid \ldots \mid P_n^{(1)}.\overline{a_n}.\beta.P_n^{(2)} \tag{7.12}$$

Here we assume that $P_1^{(1)}, \ldots, P_n^{(1)}$ are all processes that eventually become equivalent to a "zero" [1]. When these processes finish, they expose $a_2, \overline{a_2}, \ldots, \overline{a_n}$ respectively.

The sequence of input channels $a_2.\ldots.a_n$ ensures that the output channel $\overline{\beta}$ will only be active after all processes $P_1^{(1)}, \ldots, P_n^{(1)}$ terminated. This simple example shows that β channels can concisely represent parallel process synchronization. Our β channels can also easily implement the equivalent to Java's $notifyAll$ method.

7.4.2 Service reference

In this example we consider the problem to pass a reference to a service class, not to a specific service instance to a client. The client intends to contact all available service instances to ask one of them to process a certain service. We can model such system using:

$$\begin{aligned} Client &\stackrel{\text{def}}{=\!=} \text{new}\{r\}\ receiveRef(\beta_{ref}).\overline{\beta_{ref}}\langle r\rangle.r(inst).\overline{inst} \\ Service(\beta_X) &\stackrel{\text{def}}{=\!=} \text{new}\{ref\}\ \beta_X(r).\overline{r}\langle ref\rangle.ref \end{aligned} \tag{7.13}$$

The client waits until it receives a reference to a service. This reference may come from another process that is put to interact with $Client$. Then the client sends a message to all available service instances via the output channel $\overline{\beta_{ref}}\langle r\rangle$. The first service instance to reply through r sends its own address, or reference, $inst$ along r. The client then calls the service on this reference using the output channel $\overline{inst}$.

Service instances are parametrized using the β channel that will group instances of this service. When a service instance receives a call for a reference from a client, the service instance will try to reply using the received channel r. If the service instance is not the first to reply, it will block forever.

This example is not realistic in this sense, but the point we want to stress here is the implications of passing β through other channels.

7.4.3 MapReduce

MapReduce [22] is a strategy for computation distribution in which a certain task is sent to multiple servers to be executed (the map phase). After servers have executed its portion of the task using its local resources, each server sends back the partial results obtained to the caller. All partial results are reduced into a single final result (this is the reduce phase).

Let us consider a specific case of MapReduce consisting of a process P that receives tasks and n worker threads $W_1, \ldots, W_n$. when P receives a task, it sends it to the n workers to execute it and, after a timeout, P asks all workers to interrupt task execution. An outline of such system may be:

$$\begin{aligned} P &\stackrel{\text{def}}{=\!=} \text{new}\{r\}\ run(task).\overline{\beta_{start}}\langle task, r\rangle.P^{(running)}\langle r\rangle \\ P^{(running)}(r) &\stackrel{\text{def}}{=\!=} (r.P^{(running)}\langle r\rangle + \tau.\overline{\beta_{interrupt}}) \\ W &\stackrel{\text{def}}{=\!=} \beta_{start}(task, r).(R\langle task, r\rangle \mid \beta_{interrupt}.\overline{finish}) \\ R(task, r) &\stackrel{\text{def}}{=\!=} \tau.\overline{r}.(R\langle task, r\rangle + finish) + finish \end{aligned} \tag{7.14}$$

Channel β_{start} is used by P to start the same task on all worker nodes W. r is a channel for worker processes to send partial results back to P whenever a partial result is available. P goes to the $P^{(running)}$ state after all available workers were activated by the $\overline{\beta_{start}}\langle task, r\rangle$ output channel. P is a simple thread,

[1] A zero is also known as a "stop", using the nomenclature in [54] among others

so while on the $P^{(running)}$ state all calls to r are not concurrent. The especial channel τ on the $P^{(running)}$ state models the timeout until asking all nodes to interrupt processing the task.

Workers, on the other hand, consist of two threads. One, represented by R, is in charge of executing the task in cycles modeled by $\tau.\overline{r}$. R consists of a loop until the task is over, in which case the second option in the first plus of R is chosen. R then blocks waiting for a call to the $finish$ channel. The $finish$ channel may also be called any time before R decides that the task is over. In this case (the second plus sign in the expression of R), the R thread does not go to another $\tau.\overline{r}$ loop. The second thread in W simply waits for a call in $\beta_{interrupt}$ and sends a message to the first thread to either unblock it if it already finished the task, or to prevent another $\tau.\overline{r}$ loop to be executed.

7.5 Conclusions

On this paper we introduced the β channel, which models one-to-many channels as an extension of the π-calculus. Such fact, established using our definitions of β channels, allows us to carry the concept of simulation (strong, or weak) to the β-calculus as well. By doing so, one can easily draw conclusions regarding behavioral equivalence when modeling systems using the β-calculus.

We also contributed to the field by providing three non-trivial examples of applications of the β calculus. The first example demonstrates that the β calculus can enhance the expressiveness of the π-calculus, by providing a concise way to express certain sorts of process synchronizations. The second shows that when β channels are passed along channels, they allow for processes to reach a group of other processes. The third example demonstrates how more than one β channel can be used in combination to command an indefinite number of worker threads.

Part IV

Distributed middleware prototype

Chapter 8

A process calculus approach to a distributed middleware

abstract

In a distributed computing environment, the middleware is the concretion of distribution and the programming models. We propose a distributed middleware architecture built around the concept of a contract-centric approach to distributed computing. Our definition of a contract is one based on a process calculus in which channels are first class citizens. The most distinctive feature of our proposed architecture for a middleware is that it reconfigures communication channels in real time, following the creation of channels or passing references to channels as part of the process calculus. In order to adapt calculus to a non-centralized environment we introduce a new construct called β that represents a one-to-many channel. This paper also describes our experience trying to use a general-purpose message exchange specification, the JMS, as the basis for the underlying communication that our middleware uses. Although in the end we could implement all features of our proposed middleware using JMS, we found many shortcomings with the JMS specification that we present in this paper. We also present how our approach simplifies unit testing based on messages exchanged between nodes. This section is based on [60].

8.1 Introduction

A distributed middleware is not only the piece of software that makes it possible to build an infrastructure of distributed computing, but also is a concretion of a programming model. Trying to adapt new programming models to existing middleware implementations may lead to bloated solutions that hinders our ability to exploit those new programming models. This is the case of programming models in which mobile agents and dynamic interconnections play a central role.

On this paper we describe the Distributed Service Objects (DSO), a proof-of-concept distributed middleware we built to implement a programming model in which services and clients have dynamic interconnections and whose behavior is specified using a process calculus. These contracts are based on the π-calculus [71] and describe not only interactions between clients and services (the sort of messages they exchange) but also object mobility and dynamic channels allowing communication between software agents.

Our distribution model features a set of message exchange domains on which processes communicate. A process may connect to each message domain with access to read or write messages according to a policy extracted from the contracts. Each message domain represents a message bus for a particular purpose. There are message domains that are private to the middleware infrastructure, while other domains serve for clients to communicate with services.

From the point of view of application developers, the complexity of message domains is completely hidden under an OOP façade and advanced features are presented using a DSL which is pre-compiled into standard Java code. This DSL has two simultaneous purposes. First, to create objects that can be moved across the network. Second, to allow for the adherence to contracts to be formally verified.

We propose a decentralized network of message exchange nodes and that new connections can be created following the service contract. Messages are not only from software agent to a single agent, but also from agent to groups of agents. We chose Java to be the programming language for our implementation and JMS as the technology to implement the overlay network that transfers messages between processes. The choice

of JMS is both a natural one in Java and one that allows us to implement the restrictions we needed in a convenient way. Nevertheless, we found some shortcomings in the JMS standard that forced us create adaptations that we discuss here.

The rest of this paper is organized as follows. On Section 8.2 we enumerate the state of the art research on this field. On Section 8.3 we will present our programming model. Section 8.4 explains message domains. Section 8.5 presents details about our middleware implementation. Next, on Section 8.6, we discuss a service project life cycle and how our proposed solution can ease this process. We discuss our contributions on Section 8.7 and conclude this paper on Section 8.8. We also provide three appendices to offer details about our proposal: 8.9 presents an execution of a system described in terms of an expanded π-calculus; 8.10 outlines classes that implement agents of such system; and finally 8.11 presents a unit test that exploits the testing capabilities of our middleware.

8.2 Related research

The importance of message exchange in enterprise systems was already well-established. For instance, [40] proposes patterns for enterpise integration in which messages play a central role to coordinate distributed agents.

Benatallah et al. [12] propose that business protocols could be used to guide the development of clients. Services should comply with protocols and this would allow for advanced compatibility verifications. Their model is similar to ours, but in our case we focus on the dynamic creation of private channels between client and service, which is equivalent to giving the client new capabilities based on previous interactions with the service.

A study on the application of AOP to simplify the structure of middlewares and increase extensibility was presented on [82]. AOP was also proposed to be used in the structure of middlewares in [10,52,74]. The idea behind quantification is from [27]. We also applied AOP on an early state of our research but decided to replace AOP with a DSL due to shortcomings we found in AOP point cut models. Besides, our use of manipulation of generated code aims at extracting behavior patterns from the client source code and check it against a service specification.

The service composition problem is also addressed in [81] . The proposed solution is based on a catalog of previously tested patterns that can be applied to solve some design problems in the middleware.

Ning et al. [64] propose a business process management based on an Enterprise Service Bus (a message bus that coordinates an enterprise infrastructure of distributed services). The proposal also uses π-calculus to formally model the environment. There are important differences with our approach. Firstly, enterprise infrastructures have a traditionally centralized control. Here we propose a distributed and decentralized approach. Secondly, here we not only use π-calculus to model the structure of the middleware itself, but we also apply the π-calculus to model contracts between services and clients.

Wu and Li [80] propose a language to describe the architecture of middlewares for ubiquitous computing. This work also uses π-calculus to model the structure of the middleware, but as a language or pattern, in contrast with the approach in [64]. Our approach is similar, but not restricted to the middleware itself. In fact, we do not make a great distinction between elements that are part of the structure of the middleware and the services that are provided on top of the middleware. We could blur the limits between those two words using message domains, or dynamic message topics.

On [62], Mishra and Misra propose that interactions between components in a multi-threaded system should not be analyzed from source code or structure decoupling techniques such as AOP, but rather the relations between services should be detected using .NET Common Language Runtime (CLR), a commonality between languages based on the .NET platform. The approach is a valid one and, although designed for local applications, could be adapted to our model. A π-calculus formalism is also used in [62].

Ponnekanti and Fox [68] addresses the problem of how to allow interoperation between services that evolve independently. The article makes the case for web services and does not take into account implementations. But part of the proposed solution is the creation of semi-automated middleware components that can resolve incompatibilities as they are detected. Our approach, on the other hand, is one that does not try to resolve incompatibilities at run time, but that uses formal specifications to find incompatibilities earlier.

Finally, it is important to justify the choice of the variation of π-calculus we used. The one we selected for this paper was defined by Milner [71]. There are several variations of this original π-calculus. An extensive list can be found on [21]. For instance,the Asynchronous Distributed π-Calculus (ADPI) [39] defines a grammar in which option operators (a plus sign, roughly equivalent to an if block) are absent and recursion is used to add expressive power to the grammar. Here we chose the variation presented on [71] because it is the most straightforward when we try to extract behavior from structured imperative languages such as

Java. Nevertheless, here we provide an extension of the π-calculus defined in [71], which we call β and that represents a multicasting channel.

8.3 Programming model

Our programming model aims at allowing programmers to define distributed components (which are called clients or services depending on the context), but also provide the underlying distributed middleware with parameters that tell the middleware which channels to create and which rights to give to components. We separate the programming model into different levels of abstraction.

Figure 8.1 illustrates this separation. On the top, the Contract Design is the most abstract level in which contracts are defined using π-calculus expressions. After a contract was defined, developers can work on source code that implements such a contract. Source code should be written using a DSL, that resembles a general purpose programming language, such as Java or C. Such source code is then analyzed by a π-expression tool and checked for adherence to the service contract, which means that the implementation behaves the same way that the abstraction presented by the service contract does. Finally, during deployment, artifacts generated using a combination of pre-processors and compilers will generate artifacts that can be deployed in the middleware we introduce here. We will come back to this figure later, when we introduce our proposed project life cycle.

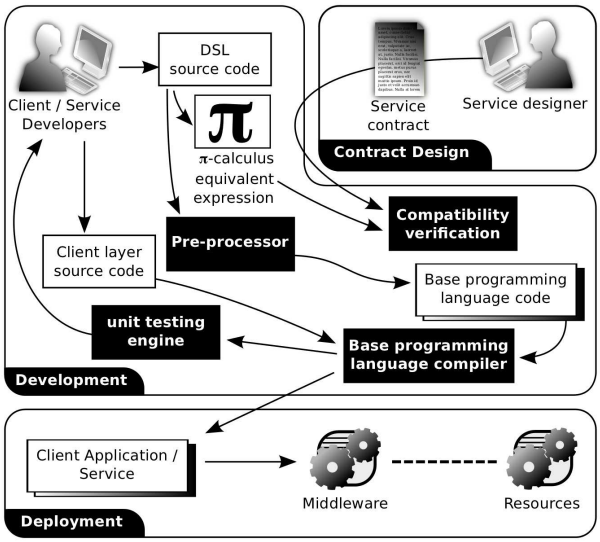

Figure 8.1: Outline of the software development process

An introduction to the π-calculus is beyond the scope of this paper, so we refer the reader to [71] for an in-depth introduction to the π-calculus. Instead of providing explanation about the syntax for the lack of available space on this paper, we will describe in words some of the π-calculus expressions we will use as soon as new aspects of π-calculus are used.

Our aim is to create a strategy in which programmers could have their clients and services checked against defects earlier, before their programs are put into work. The sort of defects we target are those that prevent client and service to interact because they follow different state models. For instance, a service may require an initial setup before starting to serve a client, and a client may not be prepared to conduct this initial setup. On another example, client and service may not agree upon the same termination protocol, making it hard for the service to tell if a client shares the same information about service termination.

We also want to propose a model for communication channels that can dynamically change during the interaction between client and service. Our goal is that such a model should be integrated into the service contracts so that they can also be used as parameters for the formal verifications of source code.

The central piece of our programming model is the service contract, which describes how both client and service should behave. The service behavior alone is not enough for a service contract since it cannot be used to represent situations in which client and service change states asynchronously.

To demonstrate this let us analyze an example. Let us consider a service in which a client requests the service to execute certain tasks. The service takes some time to execute the task and then delivers the

results to the client. While the task is executing the client is free to cancel the task any time, by calling a method on the service. When the task finishes the service sends the result of the task execution to the client asynchronously, which is done by calling a method on the client.

Instead of specifying the format of functions exposed by both service and client, the π-calculus expression specifies the behavior of each party. For instance, let us consider the following set of expressions:

$$\begin{aligned} C &\stackrel{\text{def}}{=\!=} (C_C, C_S) = C_C \mid C_S \\ C_C &\stackrel{\text{def}}{=\!=} \overline{m_1} + \overline{m_2}.m_3(a).P_1 \\ C_S &\stackrel{\text{def}}{=\!=} new\{x,y\}(m_1 \mid m_2.(\overline{m_3}\langle x\rangle + \overline{m_3}\langle y\rangle) \mid P_2 \mid P_3) \end{aligned} \tag{8.1}$$

C stands for the contract, which consists of a pair of interacting contracts: the client contract C_C and the service contract C_S. We will use Latin capital letters to represent processes. When a process is parameterized, we represent its declaration using round brackets (for instance, "$X(p)$") and its instantiation using angle brackets (for instance, "$X\langle p\rangle$"). C is the same as C_C put in parallel with C_S (indicated by the pipe character). Parallelism is the way, in the π-calculus, to represent processes put to interact. P_1 is a process that interacts with the channel a, P_2 is a process that interacts with the channel x, and P_3 interacts with channel y.

In the example above, the client may first chose arbitrarily between calling the method m_1 or m_2. Arbitrary choices are represented by a plus sign. If the client chooses to call m_1, C_C makes a transition to the zero state [1] and the process C_C ceases to exit. If, on the other hand, the client chooses to call m_2, then the service becomes able to call m_3 on the client. Now the service has the opportunity to make an arbitrary choice between passing a reference to the x channel or the y channel to the client. To make this description clear, let us see an example of execution of such contract:

$$\begin{aligned} C_C \mid C_S &\xrightarrow{(m_2)} \\ m_3(a).P_1 \mid new\{x,y\}(m_1 \mid (\overline{m_3}\langle x\rangle + \overline{m_3}\langle y\rangle) \mid P_2 \mid P_3) &\xrightarrow{(m_3)} \\ \{y/a\}P_1 \mid new\{x,y\}(m_1 \mid P_2 \mid P_3) &\xrightarrow{(y)} \cdots \end{aligned} \tag{8.2}$$

We use the notation $\xrightarrow{(m_2)}$ to denote an interaction by means of the pair $(m_2, \overline{m_2})$. The expression $\{y/a\}P_1$ means an alpha conversion of P_1 in which all names a in P_1 are replaced by y. The channels x and y are private to the service until they are passed through m_3.

We denote a visibility restriction using the syntax $new\{x,y\}(W)$, meaning that channels x and y can only be seen within the context of the process named W. For instance, in $new\{x,y\}(W) \mid Z$, it is impossible to have a transition in terms of x and y which are restricted by the *new* keyword. Figure 8.2 shows the interactions in (8.2).

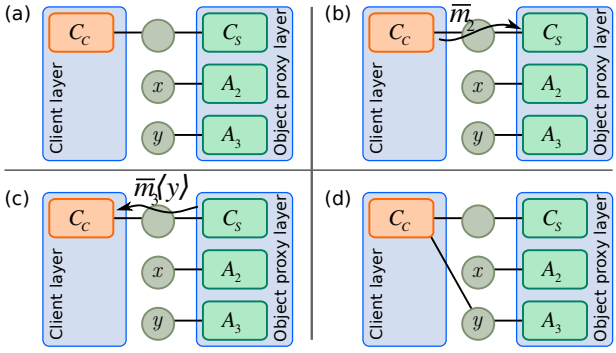

Figure 8.2: Dynamics of domain binding. Circles are message domains. (a) initial state; (b) C_C calls m_2 on C_S; (c) C_S passes a reference to y using $m_3\langle y\rangle$; (d) C_C becomes connected with A_3

[1] The zero state, also called the stop state is usually represented as a trailing 0 (zero) or the word "stop". This state represents a termination of a process. In this paper we will omit these states at the end of sequences for simplicity. For instance, the full form of any sequence would be $m_1.m_2.\ldots.m_N.0$ or $m_1.m_2.\ldots.m_N.\text{stop}$.

Figure 8.2 also illustrates the dynamics of domain connections. Initially, in Fig. 8.2 (a), the client C_C is connected only to the message domain of C_S. C_C and all objects at the client side of the contract are located in the client layer. The object proxy layer contains object proxies for remote services, as in [75].

On Fig. 8.2 (b), C_C calls m_2 on C_S. Then on Fig. 8.2 (c), C_S calls m_3 on C_C and passes a reference to the channel y as part of this call. Finally on Fig. 8.2 (d), C_C becomes connected with A_3 too through y. At this state C_C can interact with both C_S and A_3. There may be other clients connected to y at the same time, but C_C cannot see them and vice-versa. The programming model assumes that the underlying run time environment guarantees that this effect takes place.

Several clients can be connected to the server that provides C_S at a time and each client will see its own instance of C_C, so that the behavior specified in the service contract is not violated.

The equations (8.2) lead the system to a state in which P_1 can interact with P_3 through channel y. Such interaction was not possible before the server decided to send the channel y through m_3.

But our middleware is a message oriented one. So how does this process translates into messages and agents? First, before y was passed to the client process (on the m_3 transition above) this channel does not even need to exist. So the service asks for the underlying message exchange system to create such channel as soon as the channel is needed, right before passing a reference to the channel to the client.

We do not literally need a new domain in this case. In fact, even passing a reference to a channel is not necessary when only two peers are communicating, since communication between service and client is already established. The same behavior of (8.1) could be emulated if P_1 , P_2 , and P_3 were specified in terms of the channel a, and if we replace C_C and C_S by the following:

$$\begin{aligned} C_C &\stackrel{\text{def}}{=\!=} \overline{m_1} + \overline{m_2}.m_3.P_1 \\ C_S &\stackrel{\text{def}}{=\!=} new\{x_1, x_2\}(m_1 \mid m_2.\overline{m_3}.(\overline{x_1} + \overline{x_2}) \mid x_1.P_2 \mid x_2.P_3) \end{aligned} \tag{8.3}$$

The difference here is that the private channels x_1 and x_2 are guarding P_2 and P_3 from access. When m_3 is called, the service makes an arbitrary choice to enable either P_2 or P_3 by calling either x_1 or x_2 respectively. Therefore, by "passing a channel to a client", we mean connecting the client with a third agent. Let us go back to equation (8.1) and forget about (8.3), which was just a counter-example.

Passing a channel to either P_2 or P_3 means that both P_2 and P_3 are agents other than the client and the service. Let us call those agents A_2 and A_3 respectively.

So the expressions of P_2 and P_3 represent the behaviors of A_2 and A_3 from the perspective of the client, which at this point is restricted to the expression P_1.

It is also possible that A_2 and A_3 are already instantiated and serving other clients when the server C_S gives access to them to yet another client, here represented by C_C. So P_2 and P_3 should make reasonable assumptions about in which states the agents can be when C_C starts to interact with them. The simplest case is of services that do not change states and simply expose a set of service methods $m_{S1}, m_{S2}, \ldots$:

$$P_2 \stackrel{\text{def}}{=\!=} !(m_{S1} + m_{S2} + \ldots) \tag{8.4}$$

The exclamation point means a replication, or an infinite repetition of the term on the right of it. The equation above means that this service can serve one method at a time and that after processing any method, all of its methods become available to be called again.

8.4 Message domains

A message domain is a logical entity in which participants exchange messages. Each agent subscribed to a domain may have reading, writing, or reading and writing access to messages. We use the read-only access to model auditing and logging services, that only observe the activity on a message domain and cannot interfere with this activity. Reading and writing accesses represents a bidirectional communication, as in a service coordinator and participant relationship. A write-only access can be used to model situations in which agents send notifications or reports.

Unfortunately, the π-calculus only defines a one-to-one channel interaction. So we introduce a syntax shorthand called β, which is a channel in which communication is one-to-many. We define a reaction rule involving β as:

$$\begin{aligned} REACT_\beta : (\overline{\beta}.P_0 + M_0) \mid (\beta.P_1 + M_1) \mid \ \ldots \ \mid \\ (\beta.P_N + M_N) \xrightarrow{(\beta)} P_0 \mid P_1 \mid \ \ldots \ \mid P_N \end{aligned} \tag{8.5}$$

Where $(\overline{\beta}.P_0 + M_0)$ is the sender process, and $(\beta.P_1 + M_1)$, ..., $(\beta.P_N + M_N)$ are the receiver processes of a message β. There is no guarantee over the order in which each of the receivers get the message. Without parameters, β interactions can be used to emulate notification in a service level (similar to a "notify all" monitor operation). Here we use β channels primarily to multicast messages to a group of agents. This sort of special channel can be generalized using a polyadic calculus:

$$\begin{aligned} (\overline{\beta}\langle\vec{x_0}\rangle.P_0 + M_0) \mid (\beta(\vec{x_1}).P_1 + M_1) \mid \ \ldots \ \mid \\ (\beta(\vec{x_N}).P_N + M_N) \xrightarrow{(\beta)} P_0 \mid \{x_0/x_1\}P_1 \mid \ \ldots \ \mid \{x_0/x_N\}P_N \end{aligned} \tag{8.6}$$

We also use the notation $c{:}m$ to mean an input channel m associated to a group of channels c that can be transferred at once. Output usage of channels is written $c{:}\overline{m}$. We use this notation to show a clear parallel between passing a reference to a set of channels and passing a reference to an object in OOP languages. We extend the reaction above to all channels $\beta{:}x$ grouped under β.

We want to make a clear distinction between one-to-many and one-to-one transmission modes. So we will represent sets of channels that are one-to-one, as in the standard π-calculus, using the letter α. Their reaction rules will be straight forward. The example bellow shows two possible outcomes from the same original system under the same kind of reaction:

$$\begin{aligned} \alpha{:}\overline{g}.P \mid \alpha{:}g.Q \mid \alpha{:}g.R \xrightarrow{(\alpha:g)} P \mid Q \mid \alpha{:}g.R \\ \alpha{:}\overline{g}.P \mid \alpha{:}g.Q \mid \alpha{:}g.R \xrightarrow{(\alpha:g)} P \mid \alpha{:}g.Q \mid R \end{aligned} \tag{8.7}$$

8.4.1 Passing a reference to a message domain

The following is an example of passing a message domain in a channel interaction:

$$\begin{aligned} V_1 = m_1(x).!(x{:}c_1(message).\overline{consume}\langle message\rangle) \\ V_2 = new\{\beta_A\}\ \overline{m_1}\langle\beta_A\rangle \\ V_1 \mid V_2 \xrightarrow{(m_1)} V_3 = !(\beta_A{:}c_1(message).\overline{consume}\langle message\rangle) \end{aligned} \tag{8.8}$$

where m_1 is a channel to send the message domain β_A. Note that V_1 was not made specifically to interact with β_A but becomes capable of doing so after interaction with V_2. The process V_3 keeps receiving messages from $\beta_A{:}c_1$ (one of the channels in β_A) and forward them to a channel that consumes it. V_3 is not a complete system since there is nothing sending messages or consuming them. So we need to put the process V_3 in parallel with another process that has a call like $\beta_A{:}\overline{c_1}$. Let us consider a scenario in which there is an undefined number of copies of V_3 put to interact with such a process. Let us represent each copy as $V_3^{(N)}$:

$$!(T.\beta_A{:}\overline{c_1}\langle\vec{x}\rangle) \mid V_3^{(1)} \mid \ \ldots \ \mid V_3^{(n)} \tag{8.9}$$

Where T is an expression that generates different values of $\vec{x}$ on each iteration. We want the message sender $!(T.\beta_A{:}\overline{c_1}\langle\vec{x}\rangle)$ to have its structure independent from the number of replicas of T_3. That is why we need β here since the alternative to a β would be a complex π-calculus expression that would add listeners to the messages and had to iterate over all registered listeners to replicate the message. It is important to note that when several processes receive a β there is no way to determine in which order the processes received β.

8.4.2 Example: a distributed data mining service

Now let us analyze an example of usage of our approach on a distributed data mining service. Here is a list of agents in this example:

- Q_1 – Client Application – The piece of software that calls the data mining service.

- Q_2 – Resource Broker – An agent that provides a reference to a data mining service. [8]
- Q_3 – Monitoring Agent – Keeps track of all messages exchanged between agents. Data collected by this agent can be used to optimize performance, for instance.
- Q_4 – Data Mining Service – Processes a data mining algorithm with the help of a set of databases.
- Q_5 – Database – Database agents.

These agents have their communication organized using a set of message domains, or channels. Each channel is expected to dynamically accept or reject any attempt a client makes to join it. Here is the list of message domain:

- β_{DM} – Data Mining – For a client to call a data mining service and receive data mining results.
- β_{DB} – Data Base – For a data mining service to access a set of data base services.
- β_{DS} – Data Synchronization – For database agents to communicate with each other and synchronize their data. This channel can be used by databases to form a cluster.
- β_{RB} – Resource Brokerage – For a client to obtain a reference to a data mining service from a resource broker.

Table 8.1 shows read and write access to each message domain. A client Q_1 should first request a reference to a data mining service channel β_{DM}. Obtaining a reference to a service channel means that from this moment, there will be a message domain ready for the client to connect. When the client connects to this domain, the client should send a message to the domain to request whatever is listening on this domain to create an instance of the service. The client only knows the address of the message domain, not the actual address of the data mining service agent Q_4.

Table 8.1: Example of a message domain structure. 'r' and 'w' stand for read and write rights respectively.

Channel	Q_1	Q_2	Q_3	Q_4	Q_5
β_{DM}	rw	-	-	rw	-
β_{DB}	-	-	-	rw	rw
β_{DS}	-	-	r	-	rw
β_{RB}	rw	rw	r	-	-

Requests sent by the client will be executed by the data mining service agent with the help of database agents Q_5. Q_1 cannot communicate directly with any of the Q_5 agents because we are trying to hide the internal structure of the data mining service. This internal structure consists of message domains β_{DB} and β_{DS}. The later is specific for database agents to hide complexity from the data mining service. Q_3 monitors channels β_{DS} and β_{RB}.

Agents do not interact directly. Instead, they exchange stimuli using channels, which is the only artifact in π-calculus. The π-calculus expression of each process is given by the following:

$$
\begin{aligned}
Q_1 &= \text{new}\{p_1\}(\alpha_{RB}{:}\overline{getRef}\langle p_1\rangle.p_1(c_{DM}).Q_1^{(init)}\langle c_{DM}\rangle)\\
Q_1^{(init)}(b) &= \text{new}\{q_1\}(b{:}\overline{startDM}\langle q_1\rangle.q_1.(Q_1^{(init)}\langle b\rangle + \tau.0))\\
Q_2 &= \beta_{RB}{:}declare(ref).Q_2^{(init)}\\
Q_2^{(init)} &= (\alpha_{RB}{:}getRef(a).\overline{a}\langle r\rangle + \beta_{RB}{:}declare(ref)).Q_2^{(init)}\\
Q_3 &= (\beta_{RB}{:}declare(a) + \beta_{DS}{:}sync(a)).Q_3\\
Q_4 &= \text{new}\{p_4\}(\beta_{RB}{:}\overline{declare}\langle p_4\rangle.Q_4^{(waiting)})\\
Q_4^{(waiting)} &= \text{new}\{p_4\}(p_4{:}startDM(a).Q_4^{(running)}\langle a\rangle)\\
Q_4^{(running)}(a) &= \text{new}\{p_4, q_4\}(\alpha_{DB}{:}\overline{read}\langle q_4\rangle.q_4.\\
&\qquad Q_4^{(running)}\langle a\rangle + \tau.\overline{a}.Q_4^{(waiting)})\\
Q_5 &= (\alpha_{DB}{:}read(p).\overline{p} + \beta_{DS}{:}\overline{sync} + \beta_{DS}{:}sync).Q_5
\end{aligned}
\tag{8.10}
$$

The letter τ is used to represent an internal reaction, not externally observable. Nevertheless, the consequences of a τ may be observed by another process as τ represents processing. Here we use τ to represent a computation that, after an unknown or undefined time, puts an agent into another state without the need to an interaction with another process.

The channel $getRefReq$ is used to request a reference to a data mining channel. The argument to this channel call is a channel through which the response should be sent to Q_1. The agent Q_1 becomes initialized after it received c_{DM}, a reference to a remote data mining service agent. Resource brokers Q_2 are initially unable to give references to services and remain on this state until they receive a reference through the β_{RB}:$declare$ channel. Q_3 monitors some multicast conversations and does not interfere with these interactions. Agents Q_4 receive requests for data mining through β_{DB}:$startDM$ and remain on the running state until the data mining is over. Here we do not specify what exactly causes the data mining to finish. We simply present the data mining termination as an arbitrary option (represented by the plus sign in the expression of $Q_4^{(running)}$).

Finally, Q_5 receives requests for data through the α:$read$ channel. Here we use an α channel instead of a β one because we want only one instance of Q_5 to respond to requests, leaving the other instances free to serve parallel requests that may arrive from other clients.

A system consists of at least one instance of each of those five kinds of agents, except for Q_3 which is not absolutely necessary for the system to work. More than one copies of Q_2, Q_4, and Q_5 may exist in order to provide high availability services. We present a detailed example of a system and its execution on Appendix 8.9.

We can also make the expressions in (8.10) more generic by passing all α and β channels as parameters in an initial setup. For instance, Q_5 could be replaced by the equations bellow, and initialized by putting it in parallel with $\overline{setup_{DB}}\langle\alpha_{DB}, \beta_{DB}\rangle$:

$$\begin{aligned} Q_6 &= setup_{DB}(c_{DB}, c_{DS}).Q_6^{(init)}\langle c_{DB}, c_{DS}\rangle \\ Q_6^{(init)}(a,b) &= (a{:}read(p).\overline{p} + b{:}\overline{sync} + \\ &\qquad b{:}sync).Q_6^{(init)}\langle a, b\rangle \end{aligned} \tag{8.11}$$

8.5 Middleware

Mainly because of the way we propose to handle message domains and mediate services using contracts, we decided to implement our own proof of concept middleware. Figure 8.3 shows the layered structure of our middleware. Each node is a copy of the container in this figure. Each container is partitioned into five subsystems: two user subsystems and three infrastructure subsystems in which a container interact with other containers or systems.

The two user subsystems are the client and service subsystems. They consist of business logic and services for the business logic respectively. This is where classes created by the user will be located.

The other three subsystems serve as the infrastructure for the user ones. At follows we will describe each part of the middleware.

8.5.1 Business Layer

Objects in the business layer are general purpose business logic that uses service indirectly. The business layer contains classes that interface with the end user or that represent the application that ultimately uses remote services.

The only layer visible to the business layer is the client layer. References to the objects in the client layer are provided to the business layer objects through inversion of control. Business layer objects can also explicitly call an API to get those references.

The separation between the business layer and the client layer is necessary to insulate the potential complexity of the business layer from formal verification. For instance, the business layer may interface with a number of external systems or interact directly with the user. All those factors should not interfere with the operation of services.

8.5.2 Client Layer

While the business layer contains logic with a general structure that is not checked to adhere with any specification, the client layer is checked against the expected client behavior. This layer represents a bridge

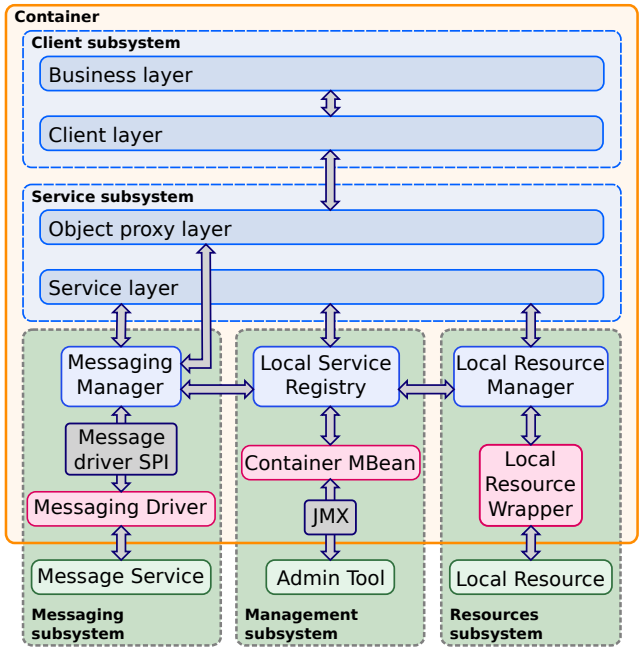

Figure 8.3: Layered structure of the middleware

between business classes and services, remote or local. Together with the business layer, the client layer makes up the client subsystem.

Objects in this layer are subject to analysis over service contract adherence. Also, objects in this layer are declared using a special DSL, which has two purposes: to reinforce restrictions in the design of objects in this layer and to allow for objects in this layer to refer to elements in the contract, such as names of service states.

The client layer can interact both with remote services (through the object proxy layer) and with local services, which are in the service layer.

8.5.3 Object Proxy Layer

This layer consists of local objects that represent remote services. The goals of objects in this layer is to translate local calls to methods into a set of message calls that cause a remote method to execute. Behind the scenes, synchronous and asynchronous decoupling is done using an implementation of the active object design pattern [24, 44], which on the one hand allows for callers of a function or method to synchronously block waiting for a response (which we model as the service client). On the other hand, a group of threads work in the background to perform the task until the result is finally delivered.

In fact, all referred objects are dynamic proxies [75], local objects that are proxies for remote ones. Whenever an object on this layer receives a request to process a method, the request is forwarded to a remote node using the messaging manager, which we will describe later.

8.5.4 Service Layer

This layer contains objects that actually process requests. Requests that originate from the client layer and may reach the client layer through two different paths:

- from the client layer through the object proxy layer, through the messaging manager, and finally to the service layer if both client and service reside on the same node, or
- from a similar path, but using the network to deliver the request to a remote node.

In the second case, an object in the client layer invokes a method in an object in the object proxy layer, which in turn converts the request into a message and asks the messaging manager to deliver this message through the network. Next, the messaging manager delivers such message, as we will describe on the next subsection. The node that receives the message is also an instance of the container depicted in Fig. 8.3. The message, then arrives at this remote node through the message service, reaches the messaging manager,

which places a request to the service layer where the service object is located. After the request is processed by the service object, the response travels the same path on the opposite direction.

Objects in the service layer are instantiated by the local service registry based on demand coming from the messaging manager .

8.5.5 Messaging Manager

As introduced on the previous subsection, the goal of the messaging manager is to deliver and receive messages on behalf local and remote object proxy layers and the service layers. The messaging manager also provides messaging capabilities to the local service registry, which will be introduced in Section 8.5.6.

Actual message sending and receiving is delegated to a messaging driver, as illustrated in Fig. 8.3. This allows for the messaging subsystem to be independent from the actual message delivery service in use. Decoupling is provided by a message driver Service Provider Interface (SPI), which we defined for our middleware. Internally, the messaging manager implements the active object design pattern [45] to decouple synchronous message delivery requests from asynchronous responses. Synchronous requests come from the object proxy and service layer of the same container instance, while asynchronous responses come from remote nodes.

A detailed structure of the messaging manager is shown in Fig. 8.4. This figure depicts three different approaches for messaging drivers: provider-specific messaging, unit testing messaging, and JMS messaging. At follows we explain each approach.

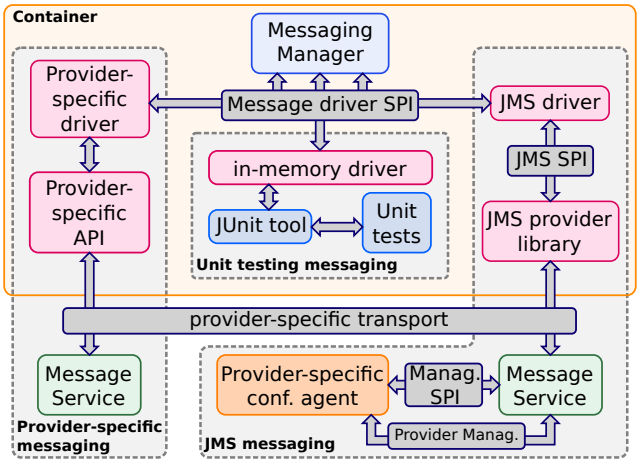

Figure 8.4: The messaging subsystem in details

Provider-specific messaging driver

The first approach is called provider-specific because the whole stack (driver, API, transport, and message service) is specific to the message service in use. This is the most generic way to implement communication.

The main problem about this approach is that creating a driver for another message service implicates creating the whole stack again. There is almost no reuse between the implementation of two drivers. The advantage of this approach is that it can use whatever feature the message service has to offer. For instance, the HornetQ [2] message service has a management API that expose methods to create topics on-the-fly.

In-memory messaging driver

The second approach is the in-memory messaging driver, or unit testing messaging. On this approach the messaging driver works completely in memory and the network is simply not used. Messages are not actually sent over the network but instead are transferred to objects within the same Java Virtual Machine (JVM) for the purpose of allowing for test automation. Each message that passes the in-memory driver is registered in a log for later verification by unit tests, which are based on JUnit [5, 11].

Such technique of replacing an actual class implementation with one that logs activity for later verification is also known as mock objects [51].

[2] http://www.jboss.org/hornetq/

After the messages were exchanged between objects a test script can check if the messages were delivered correctly. This method has its side effects for situations in which objects in the client and service subsystems compete for resources when they are on the same JVM but this competition is avoided when objects are in different JVMs. So development should be test-driven in order for tests to be possible. The same sort of problem is faced in any test environment.

JMS-based messaging

The third approach is based on the JMS and it is the most complex one because the structure was designed to allow for the replacement of JMS implementation with minimum adaptation. The JMS driver is common to any JMS implementation because it connects to the message service through the JMS SPI.

The main reason to use JMS is its flexibility. One of the main merits of using JMS is that the actual implementation of message exchange can be easily replace to meet different needs without changing the source code of the classes that use messaging. JMS specifies a SPI to message delivery services instead of a specific protocol to message exchange. In other words, it is up to each implementation of the JMS standard to specify how messages are delivered. As a result, JMS implementations are free to deliver messages using any underlying method. Many of the implementations can use multiple protocols and allow for optimizations based on the topology of message subscribers. For example, the ActiveMQ [3] implementation allows for several instances of the broker to be connected to each other and clients may connect to any of these instances. The network of brokers can optimize network usage based on the filters in each client subscription.

JMS is also a natural option to partition messages between nodes following the ideas in [56]. A message domain is a virtual space in which nodes can exchange messages freely. Nodes outside the message domain cannot send or receive messages. We used partitioning in terms of message domains to allow for a network of services to act as a single multithreaded server. Certain domains can be allocated to coordination of service objects, while others serve to allow for certain service objects to serve as infrastructure to other services. Our message domain partitions were also modeled in terms of π-calculus.

There is, however, a downside on JMS. Because in our programming model channels are created at run time and passed to agents, the messaging manager needs to ask the underlying messaging service to provide such dynamic control over connections between agents. Unfortunately, the JMS specification does not provide this capability. As a result, any attempt to implement such service on top of JMS will end up as being completely reliant on the capabilities of the message service in use. It may even be the case that a certain implementation of JMS does not support reconfiguration during run time (on-the-fly).

We created a provide-specific configuration agent that is in charge of filling this gap. This agent is a process that is in charge of receiving and processing messages asking for the dynamic reconfiguration of the message topics. Each implementation of the message exchange service or protocol requires its own implementation of the message administration agent.

Whenever the JMS driver receives from the messaging manager a request to create a new channel, the JMS driver translates this request to a call to some methods in the JMS SPI. These methods are implemented by some objects in the JMS provider library. Objects from this library then use a provider-specific transport to send the message through the network. The configuration agent receives the message and then uses a management tool that is specific to the message service in order to have the message service create a new topic.

8.5.6 Local Service Registry

Functions provided by the service registry are:

- creating new instances of service objects following the demands from the messaging manager
- providing information about which services the container is ready to provide
- finding references to service objects that were instantiated in the past
- destroying a service object that will not be used anymore
- terminate the container when asked to do so

If the service needs a resource to run, the local service registry will request a resource from the local resource manager and give the service object access to the resource using dependency injection (an inversion of control mechanism since the service receives the resource, instead of searching for it).

[3] http://activemq.apache.org/

In our current implementation, requests for containers that provide a certain service can be broadcasted. Each node that receives such requests consults its own local service registry to check if it can provide the requested service. A more sophisticated implementation would allow for an infrastructure with a better performance and would require, for instance, resource brokers to keep track of service providers. Here we did not try to address such issues, as we wanted our implementation to be only a proof of concept of a supporting infrastructure for our programming model.

The local service registry also interacts with the container MBean, which is a Java Management Extensions (JMX) service that can receive queries about the current status of the container or commands such as a request to terminate the container.

8.5.7 Local Resource Manager

Most services are based on resources outside of the container, such as a database, a programming library, a storage space, or even a sensor. The local resource manager is in charge of providing necessary resources to service objects in the service layer. As we already described, the local resource manager receives requests for resource wrappers from the local service registry.

A resource wrapper is a Java object that encapsulates primitive routines to access a resource. For instance, instead of providing a file to a service object, the local resource manager may provide an input stream to a file. Having references to resources controlled this way allows for managing resource allocation.

8.6 Source code manipulation and project life cycle

In order to manipulate Java source code we created a source code pre-processor build using the Antlr tool [67] [4]. Antlr generates a lexer and parser from an EBNF grammar. The parser creates an AST from the source code written by a programmer and this AST is then transformed into one that is compatible with the standard Java grammar. Finally, this generated source code compiled into Java classes and interfaces.

An example of transformation is when a method call contains an EST. In this case, the AST processing module translate the method call into source code that creates a message containing the expected transition along with the arguments. The AST obtained from the source code is also used to extract an equivalent π-calculus expression, which is later used to check compatibility with the service contract.

The main benefit of our approach is a programming model in which correctness of an implementation can be checked early. Our aim is to come out with a programming model in which produced artifacts are correct by design. A developer does not need to wait until a real test in order to find structural problems that prevent a client from interacting with a service. Also, as shown on Fig. 8.4, we provide an in-memory messaging driver that enables unit testing before deployment. We separate this model into three distinct phases: contract design phase, development phase, and deployment phase.

8.6.1 Contract design phase

On this phase a service designer describes a set of service contracts. Each contract consists of a set of operations in client and service, and relationships between operations. Each operation is described using both a list of arguments, which can be of two distinct types: data arguments passed by value, and references to channels.

In our prototype implementation we used the Java programming language as the underlying platform, therefore, arguments passed by value are in fact serialized Java objects in which all data is contained within the object. Such self-contained object can also be represented using some general purpose format, such as JSON [5] or XML.

Channels are modeled as references to Java agents. In other words, a channel is equivalent to the capability of interacting with a remote object. Those objects may live in a complex context, with many dependencies to other Java objects, some of them representing resources.

In terms of π-calculus expressions. Each operation is modeled as an input channel. Agent mobility is not represented directly. Instead, agents are, in principle, free to migrate from peer to peer without disrupting visibility. The contract design phase does not need to describe systems in such details. It is during the deployment phase that distribution strategies can be applied to locate computation close to data, or where resources (such as CPUs) are more powerful, for instance.

[4] http://www.antlr.org/
[5] http://www.json.org/

8.6.2 Development phase

This is the phase in which developers spend much of the programming effort to create services and clients. On this phase, developers are in charge of preparing the artifacts of client and service subsystems, as shown in Fig. 8.3.

As we saw, artifacts on those two subsystems are created either using standard Java or our DSL that expands it. The DSL source code will be subject of formal verification and translation.

Firstly, as we already explained, the DSL source code is converted into an AST representation. Next, π-calculus expressions are extracted from this AST. Those expressions are then checked against one or more target contracts to verify compatibility. If lack of compatibility is found, the developer can see an error report and correct a behavior immediately, avoiding going to the deployment phase to find the error.

On the next step, the AST is translated into one compatible with Java. During this translation, the pre-processor adds dependencies with supporting libraries and routines that perform dependency injection.

All this source code is compiled to generate byte codes that can be executed by the middleware. The developer can then submit the assembled artifacts to unit testing. Unit testing is of two types: message exchange verifications and testing based on mock objects [51].

Figure 8.5 shows an example of expected messages. In this figure there are three message verifications. Each message verification consists of a source, a channel (represented by the arrows), and a destination.

```
1  #
2  # A message from admin to the DBS group through channel C1
3  #
4  admin : 0-0-0-0-1 ===[ C1 ]===> <DBS> ( SERVICE_PROVIDER_QUERY )
5
6  #
7  # A message from any user on 6 to all nodes in C1
8  #
9  0-0-0-0-6 ===[ C1 ]===> * ( SERVICE_PROVIDER_QUERY_RESPONSE )
10
11 #
12 # A message from admin to 6
13 #
14 admin : 0-0-0-0-1 ===[ C1 ]===> 0-0-0-0-6 ( ↩
       REMOTE_SERVICE_OBJECT_REFERENCE_REQUEST / org.dso.messaging.dtos.↩
       ServiceObjectReferenceRequestDTO )
```

Figure 8.5: Example of expected message exchange asserts

The expression on line 3 checks if a message was sent from UUID 0-0-0-0-1 using the channel C1, addressed to all nodes in the DBS group. The message type is expected to be a query for a service provider. The second expression, on line 5, is checking if a node whose UUID is 0-0-0-0-6 sent a reply, and the third assert statement checks if the client asked for a reference to a service object.

Expression of expected messages should follow the EBNF grammar in Fig. 8.6. It is optional to specify which user should send the message which is represented by the question mark on the first element of "assert". The origin of the message can be expressed both using a Universally Unique Identifier (UUID) or a wildcard. Channels should be specified either using a channel name or a wildcard. The destination of the message can be a message group, between angle brackets. Finally, the message type is optional and should be within round brackets. The type of the message can be specified using a constant from a fixed list of message types, and the name of a Java class that encapsulates the data. Appendix 8.11 shows an example of unit test.

8.7 Discussion

8.7.1 A critique on JMS and centralization

As already stated, JMS is independent on the actual transport mechanism in place. By means of simple configuration, the actual topology of message exchange can be completely altered without any change in the

```
test := statement*
statement := (comment | assert) '\n'
comment := '#' text
assert := (user ':')? origin channel destination type?
user := userName | '*'
origin := UUID | '*'
channel := '===[' (channelName | '*') ']===>'
destination := UUID | '*' | '<' groupName '>'
type := '(' messageType ('/' javaType )? ')'
messageType := 'PING' | 'PONG' |
  | 'SERVICE_PROVIDER_QUERY'
  | 'SERVICE_PROVIDER_QUERY_RESPONSE'
  | 'REMOTE_SERVICE_OBJECT_REFERENCE_REQUEST'
  | 'REMOTE_SERVICE_OBJECT_REFERENCE_RESPONSE'
  | 'REMOTE_METHOD_SERVER_ALIVE'
  | 'REMOTE_METHOD_SERVER_ALIVE_RESPONSE'
  | 'REMOTE_METHOD_EXECUTION_REQUEST'
  | 'REMOTE_METHOD_EXECUTION_RESPONSE'
  | 'REMOTE_METHOD_EXECUTION_RESPONSE_EXCEPTION'
  | 'REMOTE_METHOD_EXECUTION_ACCESS_DENIED'
javaType := javaNameToken ('.' javaNameToken)*
```

Figure 8.6: EBNF grammar for expected messages asserts

way the DSO middleware is structured. Also, DSO can be configured to use a completely different JMS implementation, which can provide its own topologies for message delivery.

As a consequence, the actual message exchange in use can range from a completely centralized one (for instance, a single message service receiving and dispatching all messages) to a completely distributed one (for instance, using a reliable multicast).

What is interesting about JMS is that it allows for users to decide whether message exchange should be distributed or centralized. The final performance of a system created using our middleware will be highly dependent on the JMS in use.

8.7.2 A critique on the JMS model for message selector establishment

In the JMS programming model, it is up to each client to declare its own message filters when a connection is made to the JMS server. This design is based on the assumptions that all message consumers connected to a certain message topic should be able to consume any message sent to the topic and that it is up the consumer to correctly specify its own message filters, which reflects which sort of messages it expects to receive. The decision to receive or not a message is on the consumer, while the sender's role is to provide correct message metadata so that the filters created by message consumers work as expected by the consumers.

Such model is a problem for us since a message topic is used to transfer both sensitive data (for instance, the parameters of a method call), and data that should not reach all nodes connected to the message topic.

We need to have strict control over which node receives which message. It is the service's responsibility to specify which agent will be able to see which message. One way to solve this problem is by managing public cryptography keys in order to sign messages exchanged.

So while using JMS we cannot rely on the selections made by each client when connecting to a message topic. We cannot prevent, by means of JMS alone, that a malicious client will connect to a certain message topic and apply no filter, which allows this client to read all messages on this topic. Instead, we were forced to implement such mechanism on top of JMS using cryptography. With this mechanism in place we could avoid having a malicious client reading non-authorized messages, but a maliciously defective filter causes an overuse of network, sending messages to nodes that are not able to read them.

Nonetheless, the JMS model is still useful since it allows for broadcasts within the message topic, which we use for service nodes to perform non-functional tasks: to warn all clients about certain conditions such as service shutdown, or to send a query to all clients requesting them to send their current state. JMS is also just a reference model we used to implement our middleware.

8.7.3 A critique on the message admin agent

The main problem we found with JMS was that this standard does not define an interface to reconfigure message topics or message queues. As a result, we had to propose two sorts of adaptations: implementing a message admin agent for each JMS implementation, or using extensions of the JMS SPI provided by some JMS implementations. These two alternatives are illustrated in Fig. 8.4, respectively as "JMS messaging" and "provider-specific messaging".

A third way to implement the dynamic permissions to read and write messages would be to embed filtering of messages in the JMS driver. The problem with this approach is that it uses a JMS system only as a way to implement message multicasting, so each container receives all messages sent and is in charge of discarding messages that it should not receive. Such approach would not use message selectors, which are one of the main parameters to allow for messaging systems to optimize network usage.

8.8 Conclusions

In this paper we presented our experience developing the DSO, a distributed middleware based on a programming model in agents (or services) are developed to comply with a contract specified in terms of the π-calculus. The most distinctive feature of our contracts is the ability of calls to methods to pass references to channels using both one-to-one and one-to-many messages. The main challenge of our middleware was to come out with a way to adapt the JMS model to allow for the dynamic reconfiguration of message topics to emulate channel passing.

We also presented a brief outline of the programming capabilities we developed to enhance the expressiveness of source code, which allows for advanced formal verification even in face of channel exchange. Among the features we developed is a unit testing tool. Finally, we also propose two simple extension points to the standard π-calculus: (a) passing a set of channels not as a polyadic $\vec{c}$, but having subchannels, as in an object; and (b) allowing one-to-many messages using β channels.

Our model can also be used outside of our programming model in the general case in which interconnections are dynamic.

8.9 Appendix – An example of system based on the data mining service contract

On this appendix we analyze an example of system build using the contract in equation (8.10). On this example, there are one instance of Q_1, two instances of Q_2, two of Q_4, and two of Q_5. We won't add instances of Q_3 since this agent neither sends messages to the other agents nor changes its state in a way that is externally observable. In π-calculus terms, we would simply have to repeat the expression of Q_3 on all equations.

8.9.1 Infrastructure set up

We begin by setting up the service infrastructure, which requires service instances to declare themselves to resource brokers. The final product of this set up is two instances of Q_4 ready to be used by a client, and all resource brokers storing references to them. Because we will deal with more than one instance of certain types of agents, we will name each instance using Roman numerals $I, II, \ldots, VII$, and will identify each channel k of a certain instance X using $k^{(X)}$. Also, we will represent a state $S^{(name)}$ of the instance X using $S^{(name,X)}$.

$$Q_1^{(I)} \mid Q_2^{(II)} \mid Q_2^{(III)} \mid Q_4^{(IV)} \mid Q_4^{(V)} \mid Q_5^{(VI)} \mid Q_5^{(VII)}$$

$$\xrightarrow{(\beta_{RB}:declare)}$$

$$Q_1^{(I)} \mid Q_2^{(init,II)} \mid Q_2^{(init,III)} \mid Q_4^{(waiting,IV)} \mid Q_4^{(V)} \mid Q_5^{(VI)} \mid Q_5^{(VII)}$$

$$\xrightarrow{(\beta_{RB}:declare)}$$

$$Q_1^{(I)} \mid Q_2^{(init,II)} \mid Q_2^{(init,III)} \mid Q_4^{(waiting,IV)} \mid Q_4^{(waiting,V)} \mid Q_5^{(VI)} \mid Q_5^{(VII)}$$

On the first reaction, the channel $p_4^{(IV)}$ is transmitted from $Q_4^{(IV)}$ simultaneously to $Q_2^{(II)}$ and $Q_2^{(III)}$. If we exclude the elements that do not participate on the reaction and expand those that are involved, this reaction can be represented in more details as:

$$\begin{aligned}\beta_{RB}{:}declare(ref).Q_2^{(init,II)} \mid \beta_{RB}{:}declare(ref).Q_2^{(init,III)} \mid \\ \text{new}\{p_4^{(IV)}\}(\beta_{RB}{:}\overline{declare}\langle p_4^{(IV)}\rangle.Q_4^{(waiting,IV)}) \xrightarrow{(\beta_{RB}:declare)} \\ Q_2^{(init,II)} \mid Q_2^{(init,III)} \mid Q_4^{(waiting,IV)}\end{aligned}$$

The second reaction is almost equivalent to the first one. The channel $p_4^{(V)}$ is also transmitted to $Q_2^{(II)}$ and $Q_2^{(III)}$, which are now both on the "init" state:

$$\begin{aligned}(\alpha_{RB}{:}getRef(a).\overline{a}\langle r\rangle + \beta_{RB}{:}declare(ref)).Q_2^{(init,II)} \mid \\ (\alpha_{RB}{:}getRef(a).\overline{a}\langle r\rangle + \beta_{RB}{:}declare(ref)).Q_2^{(init,III)} \mid \\ \text{new}\{p_4^{(V)}\}(\beta_{RB}{:}\overline{declare}\langle p_4^{(V)}\rangle.Q_4^{(waiting,V)}) \xrightarrow{(\beta_{RB}:declare)} \\ Q_2^{(init,II)} \mid Q_2^{(init,III)} \mid Q_4^{(waiting,V)}\end{aligned}$$

8.9.2 Client obtaining a reference to the service

Now the client $Q_1^{(I)}$ is able to communicate with one of the resource brokers to obtain a reference to the data mining service. Bellow we show how $Q_1^{(I)}$ gets a reference of an instance of Q_4. In this example $Q_2^{(III)}$ responds to the request sent by $Q_1^{(I)}$. The response of $Q_2^{(III)}$ is a reference to the instance $Q_4^{(V)}$, which is reachable through the channel $p_4^{(V)}$.

$$\begin{aligned}Q_1^{(I)} \mid Q_2^{(init,II)} \mid Q_2^{(init,III)} \mid Q_4^{(waiting,IV)} \mid Q_4^{(waiting,V)} \mid Q_5^{(VI)} \mid Q_5^{(VII)} \\ \xrightarrow{\alpha_{RB}:getRef} \\ \text{new}\{p_1^{(I)}\}(p_1^{(I)}(c_{DM}).Q_1^{(init,I)}\langle c_{DM}\rangle) \mid Q_2^{(init,II)} \mid \overline{p_1^{(I)}}\langle p_4^{(V)}\rangle.Q_2^{(init,III)} \mid \\ Q_4^{(waiting,IV)} \mid Q_4^{(waiting,V)} \mid Q_5^{(VI)} \mid Q_5^{(VII)} \\ \xrightarrow{p_1^{(I)}} \\ Q_1^{(init,I)}\langle p_4^{(V)}\rangle \mid Q_2^{(init,II)} \mid Q_2^{(init,III)} \mid \\ Q_4^{(waiting,IV)} \mid Q_4^{(waiting,V)} \mid Q_5^{(VI)} \mid Q_5^{(VII)}\end{aligned}$$

On the reactions above, $Q_2^{(init,III)}$ was randomly chosen to respond to the request from $Q_1^{(I)}$. The presence of $\alpha_{RB}{:}getRef(a)$ in the expression of $Q_2^{(init)}$ in 8.10 is responsible for such non-determinism. When $Q_2^{(init,II)}$ and $Q_2^{(init,III)}$ are in parallel, only one of these processes will interact with a call like $\alpha_{RB}{:}\overline{getRef}$.

8.9.3 Service call

At this point the client already has a reference to the service instance and the two service instances are already instantiated. So the client can now call the service. Because neither $Q_2^{(II)}$ nor $Q_2^{(III)}$ nor $Q_4^{(waiting,IV)}$ participate in the following interactions, we use the replacement $W = Q_2^{(II)} \mid Q_2^{(III)} \mid Q_4^{(waiting,IV)}$ to simplify notation and will expand it back in the last equation:

$$
\begin{aligned}
& Q_1^{(init,I)}\langle p_4^{(V)}\rangle \mid W \mid Q_4^{(waiting,V)} \mid Q_5^{(VI)} \mid Q_5^{(VII)} \\
& \xrightarrow{p_4^{(V)}:startDM} \\
S^{(running)} = \text{new}\{q_1^{(I)}\}(q_1^{(I)}.(Q_1^{(init,I)}\langle p_4^{(V)}\rangle + \tau.0)) \mid W \mid & \\
& Q_4^{(running,V)}\langle q_1^{(I)}\rangle \mid Q_5^{(VI)} \mid Q_5^{(VII)} \\
& \xrightarrow{\alpha_{DB}:read} \\
& \text{new}\{q_1^{(I)}\}(q_1^{(I)}.(Q_1^{(init,I)}\langle p_4^{(V)}\rangle + \tau.0)) \mid W \mid \\
& \text{new}\{p_4^{(V)}, q_4^{(V)}\}(q_4^{(V)}.Q_4^{(running)}\langle q_1^{(I)}\rangle) \mid \overline{q_4^{(V)}}.Q_5^{(VI)} \mid Q_5^{(VII)} \\
\xrightarrow{q_4^{(V)}} S^{(running)} \xrightarrow{\alpha_{DB}:read} \xrightarrow{q_4^{(V)}} \ldots \xrightarrow{\alpha_{DB}:read} \xrightarrow{q_4^{(V)}} S^{(running)} \xrightarrow{\tau} & \\
& \text{new}\{q_1^{(I)}\}(q_1^{(I)}.(Q_1^{(init,I)}\langle p_4^{(V)}\rangle + \tau.0)) \mid W \mid \\
& \overline{q_1^{(I)}}.Q_4^{(waiting,V)} \mid Q_5^{(VI)} \mid Q_5^{(VII)} \\
& \xrightarrow{q_1^{(I)}} \\
\text{new}\{q_1^{(I)}\}(Q_1^{(init,I)}\langle p_4^{(V)}\rangle + \tau.0) \mid W \mid Q_4^{(waiting,V)} \mid Q_5^{(VI)} \mid Q_5^{(VII)} & \\
& \xrightarrow{\tau} \\
Q_2^{(II)} \mid Q_2^{(III)} \mid Q_4^{(waiting,IV)} \mid Q_4^{(waiting,V)} \mid Q_5^{(VI)} \mid Q_5^{(VII)} &
\end{aligned}
$$

On the first reaction, $Q_1^{(I)}$ starts the data mining process on the $Q_4^{(V)}$ service. We call this state $S^{(running)}$. On the following two reactions, $Q_4^{(V)}$ requests data from the database and gets data from it, which puts the system back to the $S^{(running)}$ state. Reactions $\xrightarrow{\alpha_{DB}:read}\xrightarrow{p^{(V)}}$ may repeat an indefinite number of times until $Q_4^{(V)}$ retrieved all data it needs. $Q_4^{(V)}$ can then perform an internal reaction τ which causes it to send the result back to $Q_1^{(I)}$ through $q_1^{(I)}$. At the end, $Q_1^{(I)}$ becomes the zero (or stop) process and disappears. On the final state the system is ready to serve another client.

More rigorously, there is no guarantee over which of the database agents will reply a request for data. On the equations above, $Q_5^{(VI)}$ is answering the requests, but $Q_5^{(VII)}$ could have replied as well. The contract was designed in a way that this choice is random. Therefore, a more rigorous representation would be one in which we define $S_A^{(running)}$ and $S_B^{(running)}$ as:

$$
\begin{aligned}
S_A^{(running)} = \text{new}\{q_1^{(I)}\}(q_1^{(I)}.(Q_1^{(init)}\langle p_4^{(V)}\rangle + \tau.0)) \mid Q_2^{(II)} \mid Q_2^{(III)} \mid & \\
Q_4^{(waiting,IV)} \mid Q_4^{(running,V)}\langle q_1^{(I)}\rangle \mid Q_5^{(VI)} \mid Q_5^{(VII)} & \\
S_B^{(running)} = \text{new}\{q_1^{(I)}\}(q_1^{(I)}.(Q_1^{(init)}\langle p_4^{(V)}\rangle + \tau.0)) \mid Q_2^{(II)} \mid Q_2^{(III)} \mid & \\
Q_4^{(running,IV)}\langle q_1^{(I)}\rangle \mid Q_4^{(waiting,V)} \mid Q_5^{(VI)} \mid Q_5^{(VII)} &
\end{aligned}
$$

We should also state that reactions by means of the channels $\alpha_{DB}{:}read$ and $p^{(V)}$ lead the system to either $S_A^{(running)}$ or $S_B^{(running)}$ randomly, until $Q_4^{(V)}$ stops calling $\alpha_{DB}{:}\overline{read}$.

8.10 Appendix – Outline of a data mining service implementation

On this appendix we present and example of artifacts that developers are expected to create on top of DSO: objects for the business, client, object proxy, and service layers.

In order to illustrate the artifacts we will present a possible implementation of the distributed data mining application that we outlined on equation (8.10). We will omit the monitoring agent Q_3 and database agents Q_5 in order to keep this appendix short and because their structure can be inferred from the description we make of Q_4.

8.10.1 Business layer

Figure 8.7 shows an example of method that belongs to a business layer object. The DMReceiver class only exists in the context of the business and client layers and is used to send data mining results to the business layer when results become available. Since only the client layer will be checked against the service contract, the business layer can establish whatever protocol with the client layer as long as this protocol does not interfere with the adherence of the client layer to the contract.

```
1  // Uploading data to client layer
2  client.uploadData(data);
3
4  DMReceiver receiver = new DMReceiver() {
5      public void asyncResult(DMResult result) {
6          // Consume data mining result
7      }
8  };
9
10 // Starting a new thread to receive results
11 // asynchronously
12 new Thread() {
13     public void run() {
14     client.startDM(receiver);
15     }
16 }.start();
17
18 // Do something else in parallel while waiting
```

Figure 8.7: Business layer source code – Example of part of a method that uses a client layer object.

The source code in Fig. 8.7 is completely unrelated to the contract in equation (8.10), which allows us to have the business layer as complex as we want without compromising formal verification. The example shows that we are even free to create threads and perform parallel tasks and receive results asynchronously.

8.10.2 Client layer

Figure 8.8 is the outline of a class in the client layer. The @InitialState annotation states that objects created from this class start in the $Q1$ state. We named the states using the same names of states in (8.10) only for readability. Such match is not mandatory. What is mandatory is that the behavior of the implementation should not violate the limits imposed by the contract. On this example, there are two new states A and B that are not named in the contract.

The @State annotation prevents a method to be called if the current state is different from the one specified in the annotation. State control is part of the DSL and allows the verification tools to extract precise π-calculus expressions from the source code.

The @Scope annotation defines which component can call a certain method. For instance, only the resource broker is capable of calling the getRefResponse method, which is how the resource broker sends a reference to the client.

We can see that while the signature of the π-calculus channel getRef accepts only a single channel as argument, the signature in the DSL accepts two arguments: the service class and the method that will be called. The service class is not on the π-calculus expression because we want to limit the π-calculus to channels (methods), or sets of channels (objects). On this source code a reference to a Java method is written (`this.getRefResponse`). The actual byte codes check if the method exists via reflection and passes a reference to the method using a string.

In this case the automatically extracted equivalent expression would be:

$$\begin{aligned} Q_1 &= \alpha_{RB}{:}\overline{getRef}\langle p\rangle.A \\ A &= p(service).Q_1^{(init)} \\ Q_1^{(init)} &= service{:}\overline{startDM}\langle q\rangle.B \\ B &= q.0 \end{aligned} \tag{8.12}$$

It is easy to see that the behavior of A is almost the same as Q_1 in equation (8.10). The main difference is that A does not evolve into a process that accepts more than one repetition of $service{:}\overline{startDM}\langle q\rangle.q$. This difference means that the client implementation is not using all capabilities of the service. Such a difference is not enough for the client implementation to be declared as not being compatible with the contract.

Note that the class in Fig. 8.8 the complexity of finding a reference to the DMService through the resource broker is hidden from the business layer.

In this example when the client object receives the data mining result, it applies a filter to the result and delivers the filtered result in several calls to the `asyncResult` method. It is important to note that the local filter is not part of the contract nor is known by the service, but the existence of such filter is not relevant in the extracted π-calculus equivalent expression since it does not change any execution flow related to communication channels defined in the contract.

8.10.3 Object proxy layer

Figure 8.9 is an object proxy interface. There are two implementations of this interface in each client-service relation. On the client side this interface is implemented by a dynamic proxy, which translates method requests to communication with the messaging manager. On the service side, this interface is implemented by the service itself, as we will see next. During pre-processing, this interface is extended to allow for channels to be transferred as an extra argument, as specified in the contract.

8.10.4 Service layer

Figure 8.10 illustrates an implementation of the service. All details concerning the data mining algorithm were omitted since we only want to discuss the structure of the service. The `init()` method is called by the container in order to allow for the service object to initialize. This method is also taken into account wen extracting the equivalent π expression. It is worth noting that the both the client and the service call procedures on the resource broker. While the client calls an α channel, the service calls a β one. Despite of this difference, there is difference between calling each kind of channel.

This service implementation also contrasts with the client implementation in that the service does not pass a reference to itself in order to implement a call to a database function and receiving the result: $\alpha_{DB}{:}\overline{read}\langle q_4\rangle.q_4$. Instead, the service simply calls a method on the database and collects the result as an ordinary Java local method: `Data d = db.read()`. Behind the scenes, the calls are equivalent, but the structure used in the client is more flexible. It allows for a call to pass a reference to a third agent that will receive the results of processing.

8.11 Appendix – Example of unit test

Figure 8.11 shows parts of a unit test that checks the basic functioning of messages. First, on line 02, the unit test creates an in-memory container. Lines 05 until 10 show the creation of a configuration object. Actual configuration is much more detailed than the one shown on Fig. 8.11. Only the relevant configuration is shown: the messaging driver to be used is the mock messaging driver, user name and password for the messaging service, the operating system integration is also a mock one, and the service provided is represented by the MyService class. So all messages transferred and all interactions with the operating system (such as attempts to create new processes) will be trapped by mock implementations that can be later analyzed by unit tests. The container is started on line 11. The first argument is a working directory in which the container will keep all temporary files and logs. The second argument is the properties that were set before.

Next, from lines 15 until 19, we create another properties object to represent the configuration used by a client. On line 22 the unit test obtains a reference to a dummy service. Having a reference to a service is equivalent to the unit testing working as the client layer.

The test verifies on line 25 that the total number of messages exchanged so far was four: one message in which the client searches for a service reference, another for the service to reply declaring itself as a service

```
1  @InitialState(Q1)
2  public class Client {
3      // Field set via inversion of control injection
4      ResourceBroker rb;
5      // Field initialized by the refResp() call back
6      DMService service;
7      // Field initialized with data received
8      Data data;
9
10     @State(Q1) // Can only be called when state is Q1
11     public void uploadData(Data data) {
12         rb.getRef(DMService.class, this.getRefResponse);
13         this.data = data;
14         to(A); // State transition
15     }
16
17     @State(A) @Scope(rb)
18     public void getRefResponse(DMService service) {
19         this.service = service;
20         to(Q1_init);
21     }
22
23     @State(Q1_init) public synchronized void
24     startDM(DMReceiver receiver, LocalFilter filter) {
25         service.startDM(this.data, this.dmResult);
26         to(B);
27     }
28
29     @State(B) @Scope(service)
30     public void dmResult(DMResult result) {
31         for (/* loop around the local filter */) {
32             if (/* local filter logic */) {
33                 receiver.asyncResult(result);
34             }
35         }
36     }
37 }
```

Figure 8.8: Client layer source code – A class that implements part of the client contract.

provider, a third message in which the client requests a service instance, and finally a fourth message in which the server replies with a service reference.

The test calls a method on line 27, and on line 29 it checks that the number of messages transferred now changed to 8. Finally, on line 31, the unit test checks if the messages transferred so far using a script called "messages.expected", similar to the one on Fig. 8.5. The MockMessagingJUnitTool class can also be used to reset the memory log of activities. This is useful for us to check only part of the messages transferred and to start each unit test with an empty log of messages, which reduces the chances of a test ending up interfering with a subsequent one. It is also possible to start several containers to run in parallel using the approach in Fig. 8.11, so we can easily simulate a network of containers and agents.

We want to come out with a programming environment in which a client can be developed and checked independently from the service implementation. To achieve this goal we replace the service with a mock version, which also keeps a log of calls and emulates the behavior of the actual service. Unit tests allow us to check aspects of software other than the interactions based on interaction models. Therefore the unit testing capabilities are complementary to the formal verification done using π-calculus.

```
interface DMService {

    void startDM(Data data);
}
```

Figure 8.9: Proxy layer source code

```
@InitialState(Q4)
public class Service implements DMService {

  // Inversion of control injection
  Database db;
  // Inversion of control injection
  ResourceBroker rb;
  // Field set with value from argument
  Method receiver;

  // Hook method to initialize the service
  @State(Q4)
  public init() {
      rb.declare(this);
      to(Q4_WAITING);
  }

  @State(Q4_WAITING)
  public void startDM(Data data,
                      Method receiver) {
    to(Q4_RUNNING);

    Result results = new Result();

    while(/*stop condition*/) {
      Data d = db.read();
      // Using d to create the result
    }

    receiver(results);
    to(Q4_WAITING);

  }

}
```

Figure 8.10: Service layer source code

```
01. // Creating a virtual container
02. Container container1 = new Container();
03.
04. // Configuring and starting this container
05. Properties containerProps = new Properties();
06. containerProps.setProperty(MESSAGING_DRIVER,
  .    MockMessagingFactory.class.getName());
07. containerProps.setProperty(
  .    CONFIGURATION_VAR_USERNAME, "agent");
08. containerProps.setProperty(
  .    CONFIGURATION_VAR_PASSWORD, "secret");
09. containerProps.setProperty(OS_INTEGRATION_DRIVER,
  .    MockOSIntegration.class.getName());
10. containerProps.setProperty(SERVICE + "1",
  .    MyService.class.getName());
11. container1.start(workingDir,
  .    new DSOSystemProperties(containerProps));
12.
13. // Creating a client configuration with a
14. // different username and password pair
15. Properties clientProps = new Properties();
16. clientProps.setProperty(MESSAGING_DRIVER,
  .    MockMessagingFactory.class.getName());
17. clientProps.setProperty(OS_INTEGRATION_DRIVER,
  .    MockOSIntegration.class.getName());
18. clientProps.setProperty(
  .    CONFIGURATION_VAR_USERNAME, "client");
19. clientProps.setProperty(
  .    CONFIGURATION_VAR_PASSWORD, "1234");
20.
21. // Obtaining a service reference
22. MyService service = (MyService) new
  .    LocalServiceDirectory(clientProperties).
  .    getServiceManager(MyService.class.getName()).
  .    getRemoteServiceObjectProxy();
23.
24. // Checking that 4 messages were exchanged
25. junit.framework.Assert.assertEquals(4,
  .    MockMessaging.getTransferredMessages().size());
26. // Calling a service method
27. service.serviceMethod();
28. // Checking that 8 messages were exchanged
29. junit.framework.Assert.assertEquals("",
 .     8, MockMessaging.getTransferredMessages().size());
30. // Checking message metadata
31. MockMessagingJUnitTool.
  .    assertMessages("messages.expected");
```

Figure 8.11: Example of unit testing using the in-memory driver

Part V

Evaluation

Chapter 9

Evaluation

Abstract

We have described our approach for a new DSL that allows for programmers to develop implementations of distributed systems specified by means of the π-calculus. The next natural step is then to evaluate our proposed approach, which is the topic of this chapter. In order to do so we present one example and present the outline of its implementation.

9.1 Introduction

On the previous chapters we described the problem we are addressing in this research and our proposed solution to it. In this section we evaluate our solution. Because the ultimate goal of our research is to provide a more convenient way for developers to specify, implement, and check software artifacts, evaluating our contribution in an experimental setting should necessarily involve measurements of human factors in system development. In the end, there would be no complete experiment without actual programmers trying to use our proposed method and tools. Such experiment should be multidisciplinary and include fields such as psychology.

Instead of going this way, our evaluation will be based on the quantitative analysis of a partial implementation of the IRC. This example also illustrates the concepts we introduced on previous chapters. Section 9.2 introduces the chat program. Section 9.3 summarizes the conclusions we can draw from the examples we presented.

9.2 Chat program

Our first example is a chat system. The design of this chat is based on the IRC protocol [42, 66]. We are not interested in implementing the whole protocol but just to demonstrate how the main functions of this protocol (namely, user connection, message exchange, and basic channel management) can be done using our proposed architecture.

The chat example is interesting from the point of view of compatibility verification since

We begin by describing the basic elements of the IRC protocol that we will implement. Then we present two different implementations of these functionality using our proposed architecture of middleware and we demonstrate how contracts could be used to control compatibility.

On the first implementation, we will try to replicate the IRC network by using agents to represent servers IRC spanning tree topology. Channels will be virtual entities collectively managed by all server agents.

On the second implementation, we model each IRC channel as a JMS message topic. This implementation is perhaps more closely related to the original IRC idea since it delegates client exchange of messages to the underlying JMS transport. On the other hand, this solution imposes some additional complexities related to channel management.

9.2.1 The IRC protocol

Topology

The IRC connection topology is illustrated by Figure 9.1. The network is such that each client should be connected to a single server. Servers are connected to each other in a spanning tree, which is based on the graph defined by all possible direct links between servers.

All servers should be aware of the whole system topology (the sets of clients, servers, and connections between them) at any given time. So a server that receives a connection from a client process is supposed to inform such topology change to the servers it is connected with.

Such information should be propagated to all servers. In the example of Figure 9.1, a new client C6 connecting to S3 would make S3 send a message to S2 to tell S2 about the new client. S2 would propagate such data to S1.

Because all servers share the same information about the network topology, any server can create a route plan for messages. In the scenario of Figure 9.1, a message from client C1 to client C5 would traverse servers S1, S2, and S3 in this order. But a message from client C3 to C5 would go through S2 and S3 only, since S2 is able to determine that the message does not need to pass through S1 to reach its destination.

Messages can also be sent to a group of clients. Routing in such a case is done similarly but when a server receives a request to send a message to a group of clients, this server should forward such request to all branches that have at least one node that should receive the message.

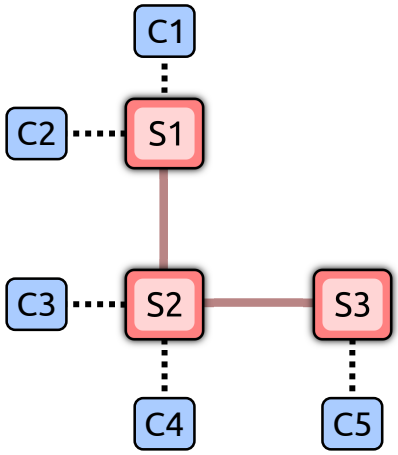

Figure 9.1: Connection structure of an IRC network. S1, S2, and S3 are servers and C1, C2, C3, C4, and C5 are clients. Servers are interconnected in a spanning tree.

To connect to an IRC system, the client should know beforehand the IP of one of the servers and its TCP/IP port.

Channels

Clients communicate through channels. The same client may join one or more channel at the same time. The original IRC specification [66] states that there is no limit on the number of simultaneous channels that a single client should join at the same time, but also suggests that it is reasonable to impose a limit on such number. For simplicity, here we will assume that there are no limitations on the number of simultaneous channels or subscriptions that servers can handle.

There are three possible situations when a client joins a channel:

- The channel does not exist – In this case the client becomes the channel owner, which is also known as a 'channel operator'. We describe the capabilities of a channel operator later.
- The channel exists and it is free to join – The client automatically joins this channel and becomes able to exchange message with other clients currently subscribed to the channel.
- The channel exists, but was marked as private by its channel operator – The client can only join the channel if, prior to the attempt to join the channel, the client received an invitation from the channel operator.

Operators

For each IRC channel, a channel operator is a client owns the channel and that is able to perform some management tasks in the channel, such as removing other users, banning users, or inviting users to join the channel, if the channel was created so that only invited users can join the channel. Channels are created by the first user who tries to connect to it, and this user automatically becomes the channel operator.

Security

In 1993 the IRC protocol was described with little concern about message privacy. The protocol was built around the assumption that the operating system user plays an important role in the distributed chat system. So one of the problems with the security is that identity verification uses ident to query for the actual user that is connecting to an IRC server. When a service receives a connection from a client, the client is required to issue a login command on the server. Such command consists of a user name in the IRC system and the user name in the local operating system, among other data. The IRC server passes the TCP/IP client port to an ident daemon (usually identd) to confirm with the client operating system that the user name passed in the login command matches with the user that is running the process that is connecting with the IRC server.

Another issue is that the protocol does not describe any sort of support for cryptography. There is a handshake between servers but not between server and clients. So the protocol does not allow for a negotiation step in which cryptography could be added to communication. The lack of cryptography can be fixed by wrapping sockets, as in Secure Shell (SSH) tunneling. But this solution requires the usage of a second protocol, which uses an independent login process.

9.2.2 Server contract

Table 9.1 lists all functions that our parcial implementation of IRC servers provide to clients. Among the simplifiations we imposed to this implementation is the restriction that no client can be simultaneously connected to more than one channel at the same time. Instead, we model a client subscription as one in which clients have to necessarily chose only one channel to be connected to at a time and clients receive messages only from the channel it is currently connected to. Such restriction greatly simplifies the design of our contract, as the complete specification would require us to create a complex data structure in π-calculus, which is completely possible, but out of the scope of this text.

Table 9.1: List of IRC agents and functions

Agent	Function name	Short name	Data arguments
Server	login	login	user/password
Connection ($Conn$)	join private channel	jpc	channel name
	join public channel	jc	channel name
Channel (Ch)	send message	sm	message
	receive message	rm	–
Channel with Public Admin (CA)	send message	sm	message
	receive message	rm	–
	remove user	ru	user name
	destroy channel	dc	–
Channel with Private Admin (CPA)	send message	sm	message
	receive message	rm	–
	remove user	ru	user name
	destroy channel	dc	–
	invite user	iu	user name

$$
\begin{aligned}
Server &\stackrel{\text{def}}{=} \text{new}\{x\}\ login(r,e).(\overline{r}\langle x\rangle.(Conn\langle x\rangle \mid Server) + \overline{e}.Server) \\
Conn(ref) &\stackrel{\text{def}}{=} \text{new}\{x\} \\
ref{:}jc(r,s,e).(\overline{r}\langle x\rangle.(Ch\langle x\rangle \mid Conn\langle ref\rangle) + \overline{s}\langle x\rangle.(CA\langle x\rangle \mid Conn\langle ref\rangle) + \overline{e}.Conn\langle ref\rangle) + \\
ref{:}jpc(r,s,e).(\overline{r}\langle x\rangle.(Ch\langle x\rangle \mid Conn\langle ref\rangle) + \overline{s}\langle x\rangle.(CPA\langle x\rangle \mid Conn\langle ref\rangle) + \overline{e}.Conn\langle ref\rangle) \quad (9.1) \\
Ch(ref) &\stackrel{\text{def}}{=} (ref{:}sm + ref{:}\overline{rm}).Ch\langle ref\rangle \\
CA(ref) &\stackrel{\text{def}}{=} (ref{:}sm + ref{:}\overline{rm} + ref{:}ru + ref{:}dc).CA\langle ref\rangle \\
CPA(ref) &\stackrel{\text{def}}{=} (ref{:}sm + ref{:}\overline{rm} + ref{:}ru + ref{:}dc + ref{:}iu).CPA\langle ref\rangle
\end{aligned}
$$

Equation (9.1) shows the server side of the system, which consists of many agents that are created on demand as they are made needed by clients. Here we rely on inversion of control to set the correct channels to the client. For instance, if the client asks to join a channel that does not exist, the connection agent will assign the caller as the channel operator. The way we designed the system is such that channel operators receive references to PCA or CA agents through inversion of control, via the s channel.

We abstract away from the actual topology of agents that make up the system because we are assuming that the contract should reflect the client perspective towards the system behavior. As we saw on Section 9.2.1, an IRC network is based on a number of server agents connected in a spanning tree, but such organization is not visible to clients.

Channels named x are those through which clients can communicate with newly created agents. Right after sending x to a client, the server creates the corresponding agent and passes x as a parameter.

Agents in this context are the expression of services from the point of view of clients. For instance, a subscription to a public channel is represented by a connection with a C agent. But several clients should be simultaneously connected to the same IRC channel for them to communicate. So each server-side agent is actually the proxy, on the server, representing a remote client.

At follows we explain the functioning of each kind of agent in equation (9.1).

Server agent – *Server*

The server agent is in charge of checking if clients are allowed to join an IRC network and providing connections to authorized clients. If a client is authorized, the server creates a connection agent using $\overline{Conn}\langle x\rangle$ and passes the channel x to the client. This channel x is used by clients to exchange messages with a connection agent. If the client is not authorized (for instance, because the password provided does not match with the expected one) the server will not create a new connection and pass a null channel to the client. Null channels will be understood as messages from server-side agents to inform clients about a failure.

The π-calculus expression $login(r)$ actually does not represent all data transferred from client to server, but only the channels exchanged between these two processes. Actual calls to the login method, as shown in Table 9.1, should carry user name and password arguments.

Argument r is the way to translate into π-calculus the capacity of the server to send something back to the caller. In concrete programming language terms, a call to r such as $\overline{r}\langle x\rangle$ is translated into a callback method in the client for dependency injection purposes.

Connection agent – *Conn*

As Table 9.1 shows, the *Conn* agent has two functions: one that allows for clients to join a private channel, and another that allows clients to join a public channel. If the channel already exists, the connection agent will decide whether the client will be allowed to connect to the channel or not. If the client is not allowed to connect to the channel, the *Conn* agent will reply by sending a null name through the r channel: $\overline{r}\langle null\rangle$. If joining the channel is successful, the *Conn* agent will send back to the client a reference to a PC or C, and decide if the client should be a channel operator or not.

If the client becomes the channel operator, in addition to sending the reference to the channel agent PC or C, the *Conn* agent will also send a reference to the admin agent PCA or CA to the client. For the passing of admin channels to be possible, clients are expected to wait for those channels to be sent through a channel that the *Conn* agent call s in equation (9.1).

IRC Channel agent – Ch

The IRC channel agent represents a channel subscription in which the connected client can only send and receive messages. A client should send messages using the $ch{:}sm$ channel and receive messages asynchronously through the $ch{:}\overline{rm}$ channel.

Public channel admin agent – CA

Represents a channel subscription that allows for the client to send messages, destroy the channel, and remove users.

Private channel admin agent – CPA

Represents a channel subscription that allows for the client to send message, destroy the channel, remove users, and invite users. By default, no user can join a private channel unless the user was already invited to join the channel.

9.2.3 Client contract

A client starts in the $C^{(start)}$ state. On this state the client can only try to log in to the server. If login is successful, the client receives a reference to a connection from the server. If login is not successful, the client process should simply die. The server will not accept any further command.

At the $C^{(connected)}$ state, the client has a connection with the server and can use such connection to request to join a channel. If a channel is private and the client did not receive an invitation to join the channel, the client receives an error report via the e channel. When this happens, the client remains at the connected state, in which it can try again to join a channel.

We assume that channel invitations do not generate messages that are sent to the client. Instead, a channel invitation simply adds the invited user to the list of users allowed to join a certain channel. Again, this decision was motivated to keep this implementation as simple as possible.

We also did not model termination of any sort for simplicity. In an IRC network the service is interactive and therefore dependent on the end user whish to continue the service, in contrast with task oriented services in which the termination of the task should mark the end of the service, and in this case a more complex termination protocol, such as the WS-BA [65] should take place.

Before presenting the client contract we define three auxiliary functions: $F^{(t)}$, $F^{(t\text{-}a)}$, and $F^{(t\text{-}p\text{-}a)}$. A supercript (t) stands for "talk", $(t\text{-}a)$ stands for "talk and administration", and $(t\text{-}p\text{-}a)$ stands for talk and administration in a private channel. A more restrictive kind of subscription is used to define a less restrictive one, in which extra functions are added:

$$\begin{aligned} F^{(t)}(ch) &\stackrel{\text{def}}{=\!=} ch{:}\overline{sm} + ch{:}rm \\ F^{(t\text{-}a)}(ch) &\stackrel{\text{def}}{=\!=} F^{(t)}\langle ch\rangle + ch{:}\overline{ru} + ch{:}\overline{dc} \\ F^{(t\text{-}p\text{-}a)}(ch) &\stackrel{\text{def}}{=\!=} F^{(t\text{-}a)}\langle ch\rangle + ch{:}\overline{iu} \end{aligned} \tag{9.2}$$

Given those functions, the client contract will be:

$$\begin{aligned} C^{(s)} &\stackrel{\text{def}}{=\!=} \text{new}\{r,e\}\ \overline{login}\langle r,e\rangle.(r(c).C^{(c)}\langle c\rangle + e.0) \\ C^{(c)}(c) &\stackrel{\text{def}}{=\!=} \text{new}\{r,s,e\} \\ c{:}\overline{jc}\langle r,s,e\rangle.(r(ch).(C^{(t)}\langle c,ch\rangle \mid C^{(c)}\langle c\rangle) + s(ch).(C^{(t\text{-}a)}\langle c,ch\rangle \mid C^{(c)}\langle c\rangle) + e.C^{(c)}\langle c\rangle) + \\ c{:}\overline{jpc}\langle r,s,e\rangle.(r(ch).(C^{(t)}\langle c,ch\rangle \mid C^{(c)}\langle c\rangle) + s(ch).(C^{(t\text{-}p\text{-}a)}\langle c,ch\rangle \mid C^{(c)}\langle c\rangle) + e.C^{(c)}\langle c\rangle) \\ C^{(t)}(ch) &\stackrel{\text{def}}{=\!=} F^{(t)}\langle ch\rangle.C^{(t)}\langle ch\rangle \\ C^{(t\text{-}a)}(ch) &\stackrel{\text{def}}{=\!=} F^{(t\text{-}a)}\langle ch\rangle.C^{(t\text{-}a)}\langle ch\rangle \\ C^{(t\text{-}p\text{-}a)}(ch) &\stackrel{\text{def}}{=\!=} F^{(t\text{-}p\text{-}a)}\langle ch\rangle.C^{(t\text{-}p\text{-}a)}\langle ch\rangle \end{aligned} \tag{9.3}$$

The superscript (s) and (c) stand for "start" and "connected", respectively. At the $C^{(s)}$ state, the client sends channels r and e to the server, which in turn is responsible for checking the data provided by the client

(user name and password) and call either r or e in case of successful or failing login, respectively. A call to the r channel will make the client go to the $C^{(c)}$ state, in which the client is able to call either $c{:}\overline{jc}$ or $c{:}\overline{jpc}$. Calls to the e channel cause the client to simply terminate, which here is represented by 0, a zero[1].

When calling either of these functions, the client sends three channels to the server: r, s, and e. The server is then in charge of selecting which of these three channels to call, reflecting a decision it made about the request send by the client. This strategy uses the Visitor design pattern [34], freeing the client from having to parse service responses before taking action. Instead, the response from the server activates the correct reaction in the client.

The first channel r should be used by the server to inform the client that the subscription was successful and that the subscription does not have administrative rights over the channel. The second channel s is called by the server to tell the client that the subscription was successful and that the client is now considered the channel operator. Finally, the e channel is used by the server to inform the client that channel subscription was not well-succeeded. Subscription is not allowed when the channel already exists, it is a private one, if the client was not invited to join it, or if the clien already joined the channel.

We should note that the reaction to interactions by means of channels r or s are such that the client is supposed to create a new process to each channel. This has a very strong consequence in the structure of the client.

It is also important to emphasize that a $C^{(c)}$ process is always available in the client. Therefore there is no limit on the number of simultaneous channel subscriptions that a client may have at any given time. The consequence is that it is impossible, by using finite states, to represent such kind of system.

9.2.4 Client implementation

We start by describing the client implementation because the client, by force of the very IRC protocol, is simpler than the server structure. Figure 9.2 shows the source code of an attempt to implement the client. The strategy used is to keep a hash map containing all channels that the client subscribed. When some other class calls the `sendMessage` method, the procedure checks if the required channel is already in the hash map. If it is, then use it, if it is not, then the method attempts to retrieve a new channel object.

Figure 9.4 is the `IRCCHannelWrapper` class. Note that this implementation does not use all available functions of the contract. For instance, this implementation does not allow for the client to create private channels. If this was a GUI-based implementation, it would be equivalent to an interface with a missing button called "create private channel". Such channel would require a "invite user" button, that the implementation does not support neither. Another missing function is the capability of deleting the channel. As we will see, the lack of these functions do not qualify the client as not being a legal implementation of the contract. As we already saw, client implementations are free to implement the contracts only partially.

The π-calculus expression is the following:

$$\begin{aligned} A &\stackrel{\text{def}}{=} \overline{login}\langle r, e\rangle.B \\ B &\stackrel{\text{def}}{=} r(c).C + e.Z \\ C &\stackrel{\text{def}}{=} c{:}\overline{jc}\langle r, s, e\rangle.(r(c).(C'\langle c\rangle \mid C) + s(c).(C''\langle c\rangle \mid C) + e.C) \\ C'(c) &\stackrel{\text{def}}{=} (c{:}\overline{sm} + c{:}rm).C'\langle c\rangle \\ C''(c) &\stackrel{\text{def}}{=} (c{:}\overline{sm} + c{:}rm + c{:}\overline{ru}).C''\langle c\rangle \\ Z &\stackrel{\text{def}}{=} 0 \end{aligned} \tag{9.4}$$

The following relation provides the proof of simulation:

$$\begin{aligned} R = \{&(A, C^{(s)}), (B, r(c).C^{(c)}\langle c\rangle + e.0), (C, C^{(c)}\langle c\rangle), (Z, 0), \\ &(r(c).(C'\langle c\rangle \mid C) + s(c).(C''\langle c\rangle \mid C) + e.C, \\ &r(ch).(C^{(t)}\langle c, ch\rangle \mid C^{(c)}\langle c\rangle) + s(ch).(C^{(t\text{-}a)}\langle c, ch\rangle \mid C^{(c)}\langle c\rangle) + e.C^{(c)}\langle c\rangle), \\ &(C'\langle c\rangle \mid C, C^{(t)}\langle c, ch\rangle \mid C^{(c)}\langle c\rangle), \\ &(C''\langle c\rangle \mid C, C^{(t\text{-}a)}\langle c, ch\rangle \mid C^{(c)}\langle c\rangle), \\ &(C'\langle c\rangle, C^{(t)}\langle c\rangle), (C''\langle c\rangle, C^{(t\text{-}a)}\langle c\rangle)\} \end{aligned} \tag{9.5}$$

[1] As we saw, another way to represent termination is using stop instead of 0.

```
1  import java.util.HashMap;
2
3  @InitialState(A)
4  public class IRCClient {
5      private IRCServer server;
6      private IRCConnection connection;
7      private HashMap<String, ChannelWrapper> channels = new HashMap<↩
           String, ChannelWrapper>();
8      private HashMap<String, ChannelAdminWrapper> channelsAdmin = new ↩
           HashMap<String, ChannelAdminWrapper>();
9      @State(A)
10     public void login(String username, String password) {
11         server.login(username, password, this.ok, this.ko);
12         to(B);
13     }
14     @State(B)
15     @Scope(server)
16     public void ok(IRCConnection connection) {
17         this.connection = connection;
18         to(C);
19     }
20     @State(B)
21     @Scope(server)
22     public void ko() {
23         System.err.println("Login␣invalid.");
24         to(Z); // Prevents any further method call
25     }
26     @State(C)
27     public void sendMessage(final String channelName, final String ↩
           message) {
28         if (channels.containsKey(channelName)) {
29             ChannelWrapper cw = channels.get(channelName);
30             cw.send(message);
31         } else if (channelsAdmin.containsKey(channelName)) {
32             ChannelAdminWrapper cwa = channelsAdmin.get(channelName);
33             cwa.send(message);
34         } else {
35             connection.join(channelName, new Return() {
36                 public void r(ChannelWrapper cw) {
37                     channels.put(channelName, cw);
38                     cw.send(message); }
39                 public void s(ChannelWrapperAdmin cwa) {
40                     channelsAdmin.put(channelName, cwa);
41                     cwa.send(message); }
42                 public void e() {
43                     System.err.println("Error!"); } });
44         }
45     }
46     @State(C)
47     public void removeUser(String channelName, String userName) {
48         if (channelsAdmin.containsKey(channelName)) {
49             ChannelAdminWrapper cw = channelsAdmin.get(channelName);
50             cw.removeUser(userName);
51         }
52     }
53 }
```

Figure 9.2: An IRC client implementation

```
1  public class ChannelWrapper implements IRCMessageListener {
2
3      private IRCChannel channel;
4
5      public ChannelWrapper(IRCChannel channel) {
6          this.channel = channel;
7          channel.setListener(this);
8      }
9
10     public void send(String message) {
11         channel.send(message);
12     }
13
14     @Scope(channel)
15     public void receive(IRCMessage message) {
16         System.out.println(message);
17     }
18 }
```

Figure 9.3: The IRC channel wrapper class

```
1  public class ChannelAdminWrapper extends ChannelWrapper {
2      public ChannelAdminWrapper(IRCChannel channel) {
3          super(channel);
4      }
5
6      public removeUser(String userName) {
7          channel.removeUser(userName);
8      }
9  }
```

Figure 9.4: The IRC channel admin wrapper class

Therefore the client implementation correctly implements the client contract. Equations (9.5) do not present all possible states reachable since each call to $c{:}\overline{jc}$ or $c{:}\overline{jpc}$ adds an extra $C^{(t)}\langle c\rangle$, $C^{(t\text{-}a)}\langle c\rangle$, or $C^{(t\text{-}p\text{-}a)}\langle c\rangle$ process to the system. Therefore, formally the number of states can grow indefinitely. But it can be shown that R proves that the client implementation simulates any state that can be reached by the client contract.

Note that the inverse relation R^{-1} does is not a simulation just take the pair $(C''\langle c\rangle, C^{(t\text{-}a)}\langle c\rangle)$. It is easy to see that the inverse, $(C^{(t\text{-}a)}\langle c\rangle, C''\langle c\rangle)$, cannot be in a simulation relation since $C^{(t\text{-}a)}\langle c\rangle \xrightarrow{ch:\overline{dc}} C^{(t\text{-}a)}\langle c\rangle$, but no transition is possible from $C''\langle c\rangle$ by means of $ch{:}\overline{dc}$. Therefore there no relation that contains $(C^{(t\text{-}a)}\langle c\rangle, C''\langle c\rangle)$ is a simulation. We already expected that R^{-1} would not be a simulation since, as we stated above, this client does not implement all possible functions present in the contract.

9.2.5 First implementation: agents as servers

Our first implementation is one in which agent topology attempts to reproduce the topology of the IRC network. Our implementation is not concerned with the actual bytes transferred through the network, since transport was abstracted and delegated to messaging drivers, as explained in Chapter 8.5.5. The topology of this solution is depicted in Figure 9.5.

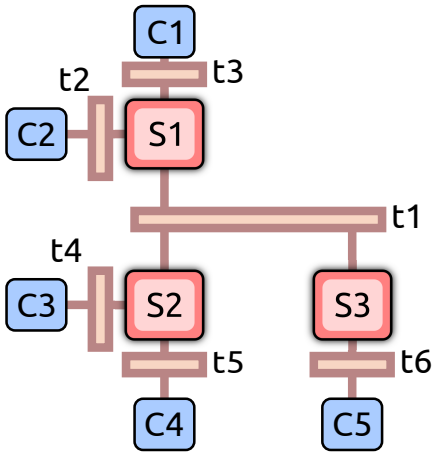

Figure 9.5: Our first implementation of a chat system

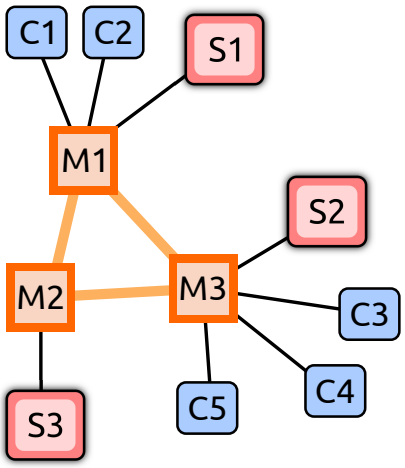

Figure 9.6: Example of actual topology of process connections

Each rectangle (t1, t2, t3, t4, t5, and t6) represents a message topic. Each client connection with a server process requires a specific message topic. Also, all service processes are connected to a single message topic t1 that represents the message domain in which services share information regarding client topology.

While Figure 9.5 depicts the logical way in which agents are interconnected, Figure 9.6 is an example of an actual JMS deployment. Nodes M1, M2, and M3 are JMS messaging processes. All other agents are

directly connected to M1, M2, or M3 through sockets. The actual protocol used for agents and messaging processes to communicate depend on the actual JMS implementation.

Messaging processes are in charge of mediating all communication between all agents. In this example, M1, M2, and M3 are all connected. They use these connections to share data about updates in the topology of the JMS system and to transfer messages that should reach client processes agents connected to other messaging processes. This setup can be used to optimize network usage, or to improve the responsiveness of the system as a whole since each computer can be in charge of processing part of the job of receiving and sending messages to connected nodes. Redundant messaging processes can also provide a high availability of the messaging services.

On this research we propose a network of distributed services in which the underlying message transfer remains transparent to programmers. As we saw in Chapter 8, one of the ways to implement the messaging subsystem is by using JMS. The actual transport can be base on a network of servers that work in cooperation to deliver messages to connected clients. Therefore, the actual topology of servers in DSO may be more sophisticated than the one described in the IRC protocol. For instance, ActiveMQ supports a topology called "network of brokers" in which several ActiveMQ processes, or brokers, are connected to virtually function as a single broker.

While we do not intend to emulate the actual message delivery, we can create an infrastructure of services whose structure reminds of the IRC protocol. Because our aim is to provide enough arguments to support that the whole protocol could be implemented with our methods, we are going to implement a subset of the functions present in the IRC protocol.

```
1  public class IRCServer {
2
3      public void login(String username, String password, r, e) {
4          if ( /* check access */) {
5              r(new IRCConnection(username));
6          } else {
7              e();
8          }
9      }
10 }
```

Figure 9.7: IRC login server

9.2.6 Second implementation: message domains as channels

On this second implementation we propose that each IRC channel should be implemented by a JMS topic. This implementation uses the messaging exchange mechanism in a way that fits better to the JMS model, which means that it surely takes better advantage of optimizations and the routing mechanisms available in the JMS network in use.

While the actual connection configuration is also the one in Figure 9.6, topic usage is depicted in Figure 9.9. ch1 and ch2 are chat channels, while m is a management channel used by channel operators to manage chat channels. In the example of Figure 9.9 channel operators are C2 and C5. S1 is an agent that is in charge of receiving management messages from channel operators and reconfiguring message topics, as explained in Section 8.5.5.

9.3 Conclusions

The example we just presented allows us to clearly see the benefits of our proposal. As we already stated, we could see on the example that a contract the way we are specifying using the π-calculus is not possible if we use a finite number of states. This is because the IRC protocol specifies that there is no limit in the number of simultaneous channels a certain client may be connected to. All channels are expected to send asynchronous messages to the client, which forces us to model such protocol using a structure that generates new processes.

As we saw, an actual implementation of this protocol using our approach does not force one to actually have a process (even a thread) for each channel, but asynchronous events may be channeled through a single

process. In theory, this eliminates the possibility of a true simultaneous reception of messages, but such problem does not represent a serious violation of the IRC protocol.

We also saw that it is possible to implement the same contract using different agent topologies. In other words, the π-calculus contract may be intentionally vague to allow for many sorts of implementations to be created around it.

```
1  public class IRCConnection implements MessageDomainListener {
2
3      public static final String IRC_BUS = "IRCServerBus";
4      public static final long BUS_TIMEOUT = 2000;
5
6      // Injected using inversion of control
7      private MessageDomainFactory dmf;
8      private MessageDomain domain;
9      private String username;
10
11     private HashMap<String, IRCChannel> channelMap = new HashMap<String,↩
           IRCChannel>();
12
13     public IRCConnection(String username) {
14         domain = dmf.get(IRC_BUS);
15         this.username = username
16     }
17
18     public void jc(String channelName, r, s, e) {
19         domain.send(new IRCChannelSearch(channelName));
20         IRCChannelSearchResponse resp = domain.waitFirst(↩
               IRCChannelSearchResponse.class, BUS_TIMEOUT);
21         if (resp == null) { // Should create a new channel
22             IRCChannel c = new IRCChannel(domain);
23             channelMap.put(channelName, c);
24             s(c);
25         } else { // Channel already exists
26             domain.send(new IRCChannelJoinRequest(channelName, username)↩
                   );
27         IRCChannelJoinResponse jr = domain.waitMessage(↩
               IRCChannelJoinResponse.class, BUS_TIMEOUT);
28         if (jr.isAccept()) {
29                 r(CONTINUE HERE);
30             } else {
31                 e();
32             }
33         }
34     }
35
36     public void jpc(String channelName, r, s, e) {
37         // (...)
38     }
39
40     @Scope(domain)
41     public void receiveMessage(Message m) {
42
43     }
44 }
```

Figure 9.8: IRC connection

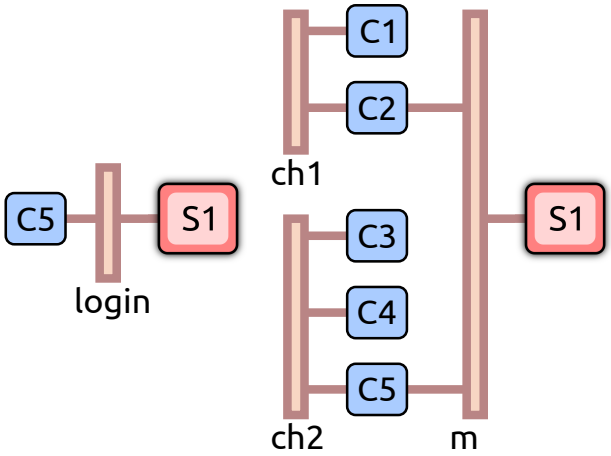

Figure 9.9: Our second implementation of a chat system

Part VI

Conclusions

Chapter 10

Conclusion

Abstract

We could shortly describe our proposal as a model for the development of distributed systems. Our focus was on which working environment a programmer will utilize and which abstractions a distributed system designer will utilize. We started on the basis of an OOP model of distributed objects. Next, we evaluated the design principles of message domains. We also analyzed how we could take advantage of defining our own DSL to create a seamless version evaluation mechanism. Finally, we presented our implementation of a middleware that not only applies those ideas but also has some advanced testing and debugging capabilities. In this chapter we summarize all this discussion, present details about the research process, and state our perspectives for future developments that follow the same research direction.

10.1 Introduction

We start by providing a summary of our proposal. This research has main two parts. On the first one we attempted to solve the problem with finite state machines. On the second part we explain our second approach in which we applied the π-calculus to express contracts of distributed services.

Although the first proposal was for us an important evolutionary step, the second proposal proved to be much more far reaching. We compare both approaches after the summary.

Finally, we provide a brief outline of our main contributions.

10.2 Summary of our proposal

10.2.1 Part I – FSMs

Our first approach showed to be enough for systems in which the number of states is finite. We argue that those systems are the majority of distributed ones, since in an ideal client-server set up the server should be ready to serve any function chosen by the client any time, and the server should expect anything from the client at any time. In terms of number of states, the client may have an infinite number of possible states, issuing any signal to the server at any time. The server, on the other hand, should have a very predictable and easy to guess behavior, so ideally, the server should have as few states as possible, allowing the client to chose whatever function it needs at any time.

Obviously, real systems differ from this simplification. In reality, systems tend to be somewhere in between this ideal (of a pure service serving a pure client) and a complex cooperation between agents in which agents share communication responsibilities.

In terms of process calculus, our approach was successful to deal with the sort of system in which channels are fixed and the number of processes do not change along time. We can call those systems static and say that this first approach was a reasonable solution for those systems, while the second approach solves the same problem for dynamic systems.

10.2.2 Part II – π-calculus

For dynamic systems, the π-calculus provides us with a reasoning that allows us to describe systems in a more rich way. Although this method allows us to model systems of a much greater range, there are still

limitations. We focus on systems S consisting of a set of processes $S \stackrel{\text{def}}{=\!=} P_1 \mid \ldots \mid P_n$. With each of the n processes fixed. But what happens when we put S to interact with a new environment, therefore exposed to a different set of processes? In other words, what if we have a new system S in which $S' \stackrel{\text{def}}{=\!=} S \mid Q$? Such question is harder to answer when we consider that Q interferes with S, changing its behavior. We are yet to address such concerns and, as we have shown in the structure of our middleware and programming model, we avoid such a problem by imposing a control over the involved agents, which we represented using layers (business, client, object proxy, and service).

10.3 Contributions

For the best of our knowledge, we are the first to attempt to use the π-calculus as a means to represent service contracts. We have outlined a programming model and run time environment making realistic assumptions. Our proposal can also be easily adapted to the general case of formally checking distributed systems in other scenarios, possibly using other programming languages, or other ways to express contracts.

Although we still consider our middleware as a tentative implementation, our experience implementing it helped us find differences between a messaging service (JMS in our case) and the requirements for a message system that provides the dynamic interconnection that our proposed architecture needs. We could not provide a solution that is general for any JMS implementation. Instead, we proposed a pattern for adapters to be created. The role of these adapters is to add functions in a JMS implementation that allows for messages to manage the topology of message topics.

Part VII

Appendices

Appendix A

Middleware manual

As we already presented in Section8, we have implemented a simple prototype middleware to test our approach for distribution. In this appendix we present a pragmatic manual for the middleware.

All commands are designed for Linux systems and were tested in Ubuntu 10.10, but as we implemented the middleware in Java, all tools can also be used in Windows and MacOS environments. The Java version utilized was jdk1.6.0_25 and jdk1.7.0_21. We did not test it against older versions of Java, but as we utilized Java 6, one can expect that old JVMs will be not fully compatible.

A.1 Admin tool

The admin tool is a stand alone application to manage containers running in a local machine.

In order to manage containers during their execution, the admin tool makes a local connection to the containers using JMX. For now we are not providing any support for remote administration as it makes setup and security more complex. Nevertheless, it is possible to administer remote agents by using the admin tool in a remote terminal (for instance, using SSH).

A.1.1 Starting a container

A container is the process that will keep a single agent running. Staring a container will create a software agent instance and connect it to the message exchange system. Containers can be simply started as a stand-alone Java application and will require the location of a configuration file.

In all examples in this section, let us assume that the directory /a/b/c contains the following files: dso.properties, dso1.properties, dso2.properties, and dso3.properties. Let us also assume that the contents of these files are presented in Figures A.1, A.2, A.3, and A.4 respectively. Let us also call the containers specified by these files c_1, c_2, and c_3.

```
1  dso.messagingDriver=org.dso.messaging.drivers.activemq.MessagingFactory
2  dso.activemqURL=tcp://localhost:61616
3  dso.osDriver=org.dso.osintegration.linux.LinuxOSIntegration
4
5  dso.service=x.y.z.MyService
6
7  log4j.rootLogger=DEBUG, FILE
8
9  log4j.appender.FILE=org.apache.log4j.FileAppender
10 log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
11 log4j.appender.FILE.layout.ConversionPattern= %d - %-4r [%t] %-5p %c %x ↩
       - %m%n
```

Figure A.1: Contents of /a/b/c/dso.properties

Note that dso.properties does not contain a complete container specification since this file does not have a dso.uuid entry. Therefore, these four files specify how to start only three containers. Also note that

```
1 dso.uuid=0000-00-00-00-000001
2 log4j.appender.FILE.File=/var/log/container1.log
```

Figure A.2: Contents of /a/b/c/dso1.properties

```
1 dso.uuid=0000-00-00-00-000002
2 log4j.appender.FILE.File=/var/log/container2.log
```

Figure A.3: Contents of /a/b/c/dso2.properties

```
1 dso.uuid=0000-00-00-00-000003
2 log4j.appender.FILE.File=/var/log/container3.log
```

Figure A.4: Contents of /a/b/c/dso3.properties

dso1.properties, dso2.properties and dso3.properties define three different files to write log messages. This is important for us to make sure that the logs of the containers don't mix.

Starting all containers in a certain directory

Figure A.5 shows how to start all containers specified in a certain directory. The admin tool will find all properties files and will start all three containers.

```
1 admin start /a/b/c
```

Figure A.5: Starting a container

Starting only one container by passing the configuration file

Figure A.6 shows how to start only the container c_2. The admin tool will read only the files dso2.properties and dso.properties.

```
1 admin start /a/b/c/dso2.properties
```

Figure A.6: Starting a container

Starting only one container by UUID

It is also possible to select only one container based on its UUID . Figure A.7 shows how to start only c_3. The admin tool will read only the files dso3.properties and dso.properties.

Note that UUIDs can be specified using either a short or a long format. For instance, the strings 0000-00-00-00-000011 and 00000000-0000-0000-0000-000000000011 represent the same UUID.

```
1 admin start /a/b/c/ 0000-00-00-00-000003
```

Figure A.7: Starting a container

Starting all containers in the current directory

If no argument is given after the 'start' clause, the admin tool will try to read all containers in the current directory. Figure A.8 shows how to do it with the containers of our example.

```
1 cd /a/b/c
2 admin start
```

Figure A.8: Starting a container

Manually starting a container

Alternatively, you can also start a single container manually using the command in Figure A.9. Running a container this way is exactly what the admin tool does behind the scenes. Running it manually can be useful for debugging purposes.

```
1 java -cp DSO.jar org.dso.service.container.Container ./dso.properties
```

Figure A.9: Starting a container

A.1.2 Listing containers

After starting one or more containers it is possible to list which containers are running using the status clause, as in Figure A.10.

```
1 admin status
```

Figure A.10: Listing containers

The output will be a list of all containers currently running. The list also shows during how many milliseconds the container is running and the Process Identifier (PID) of the container. The PID is useful for us to change the priority of a container in the operating system. Also, the PID can be used to shutdown a container, as we will see at follows.

A more detailed version of the 'status' command is the 'detailedstatus' command. The difference is that 'detailedstatus' also prints several system properties of the remote JVM, such as the working directory and the JVM class path. Some of the system properties variables (especially the class path) can be quite long.

A.1.3 Stopping a container

There are four ways to stop a container using the admin tool: asking the tool to stop all containers, specifying the container to stop by passing the container configuration file, specifying the container to stop by passing the operating system's PID, and specifying the container to stop by passing the container's UUID.

Stopping all containers

Stopping a single container using a file name

Stopping a single container using a PID

Stopping a single container using a UUID

As in the start command, UUIDs can be specified either in the short of long forms.

A.1.4 JMX

All operations performed by the admin tool after the console was started are done using the JMX. Therefore, these operations can all be performed using a JMX such as the JConsole, which is part of the Java Development Kit (JDK).

```
1 admin stop all
```

Figure A.11: Stopping all containers

```
1 admin stop dso2.properties
```

Figure A.12: Stopping a container by passing a configuration file

A.2 Configuration file of a container

Configuration files can become very long which may make them hard to maintain. Yet, most of configuration in configuration files is the same across containers. For instance, if we want to have five containers running on the same host machine, we are likely to have all five containers connecting to the same messaging system and outputting log messages using the same message format. The differences in configuration between containers may be which user name to use to connect to the messaging system, or which log file will receive logs from each container.

For this reason, we have chosen configuration files to be hierarchical. Each container is configured by one configuration file but a root configuration file may contain all configuration that is common to all children configuration files.

Configuration files in the DSO middleware use the standard plain text Java configuration format [1]. Besides configuration files in plain text, the Java API for configuration files can also read an Extensible Markup Language (XML) format. But we opted for using the simplest plain text format because it is less verbose than the XML format. Besides, the XML format does not add any feature to the configuration files in our usage scenario.

Figure A.15 shows an example of configuration file. The sharp character (#) starts a comment line.

All DSO configuration has the 'dso.' prefix, which makes it easy to use a single configuration file for DSO and other systems, when needed. For example, the bottom part of the configuration file is fed to the Log4J tool.

As we already mentioned, each container makes a single connection to the messaging system.

A.2.1 Service properties

Services may have specific configurations as well, which are useful for us to control the behavior of services without having to change its source code. For example, a service that connects to a database may have service properties that provide configuration parameters to the service. Configuration variables are declared in the implementation of a service. They are not available in the service interface since the values of these variables do not interest to the service user, only to the service container.

Service properties should be set in the container configuration file. If the property name is "property1", the configuration should be set using an entry called "dso.service.conf.property1", as shown in Figure A.15.

When creating a service, a programmer can easily specify a service property by decorating a field of an implementation class with the ServiceProperty annotation. Figure A.16 shows an example of a service implementation class with two service properties.

As the example shows, the ServiceProperty annotation can have a defaultValue property set. This property is the default value of the property if the configuration file does not provide any value for the property. The type of a service property should be a Java primitive (int, double, etc), a class equivalent to a primitive (Integer, Double, etc) or the String class.

Instance policy

The instance policy is a configuration variable that controls when the container creates new instances of service objects. This variable may have three different values [2]:

- ONE_PER_CONTAINER – Each container creates only one instance of the service object.

[1] For a detailed reference on the format of configuration files, please refer to http://download.oracle.com/javase/6/docs/api/java/util/Properties.html#load(java.io.Reader)

[2] These values are internally parsed to the org.dso.service.container.content.InstancePolicy enumeration.

```
admin stop 876
```

Figure A.13: Stopping container with PID 876

```
admin stop 0000-00-00-00-000003
```

Figure A.14: Stopping container with UUID 0000-00-00-00-000003

```
dso.messagingDriver=org.dso.messaging.drivers.activemq.MessagingFactory
dso.activemqURL=tcp://localhost:61616

#
# Adding items to the class path
dso.classpath1=${dso.home}/lib.jar

dso.containerHome=${dso.home}/../../containersHome/000001/
dso.uuid=000000-00-00-00-000001

# To be replaced by the shell script
dso.service1.conf.outputDir=OUTPUT_DIR

# Service properties
dso.service1.conf.containerNumber=1

dso.osDriver=org.dso.osintegration.linux.LinuxOSIntegration

dso.username=admin1
dso.password=786

# Will be defined in children files:
# dso.uuid=

dso.customerTopic1=C1

dso.group1=ADMINISTRATORS

dso.service1=example.CounterService
dso.service1.serviceUseMock=no
dso.service1.instancePolicy=ONE_PER_CONTAINER

#
# Log4J configuration
#

log4j.rootLogger=DEBUG, FILE

log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=${dso.home}/x.log
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.MaxFileSize=1MB
log4j.appender.FILE.MaxBackupIndex=0
```

Figure A.15: Example of DSO configuration file

- ONE_PER_CLIENT – Each client will have its own instance of the service object.

```
1  public class MultiplicationServiceImpl extends ServiceImplementation ↩
       implements MultiplicationService {
2
3    @ServiceProperty(defaultValue="Default␣property␣1")
4    private String property1;
5
6    @ServiceProperty(defaultValue="Default␣property␣2")
7    private String property2;
8
9    (...)
10
11 }
```

Figure A.16: Service implementation with service properties

- ONE_PER_CALL – Each call to any function will create a new instance of the service object.

The question is now how these policies relate to the mobility of objects. Let's say that a certain service S_1 is installed on node n_1. A service object $s_{1,1}$ was created using the ONE_PER_CONTAINER policy on node n_1. Calls to S_1 on this server will always be directed to $s_{1,1}$. What happens when another instance $s_{1,2}$ of the same service S_1 arrives at n_1? Suppose this instance was created on a node n_2 in which the policy is ONE_PER_CLIENT. In this case we need to make sure that the behavior of S_1 for all its clients do not change on n_1. In other words, clients that already have a reference to $s_{1,1}$ should be still served by $s_{1,1}$, not $s_{1,2}$.

A.2.2 Log4J configuration

The bottom part of the file in Figure A.15 are properties to configure the behavior of the Log4J logging tool.

The configuration file is also forwarded the Log4J classes [3], which control all the log capabilities of DSO. Log4J reads all configuration variables starting with the 'log4j.' prefix. In the example of Figure A.15, we are forwarding logs to a file. We log everything whose level is at least DEBUG. We chose to utilize Log4J because it is very flexible and modular. For instance, logs can be sent to a remote log server and several log appenders can be utilized simultaneously.

A.3 Jython client

The Jython client is an interactive console application that enables users to use the infrastructure of services in a text-based interface. This client tool is useful when an user wants to call service methods without the need to implement a client application.

All commands entered by the user are processed using the Jython programming language. Jython is the implementation of the Python programming language as a script engine for the Java platform. Jython is almost fully compatible with the Python language specification, therefore most of what can be done using the C implementation of Python can also be done using Jython.

Besides the basic Python programming language environment, the Jython client also provides an easy way to access remote services through the network. All services are presented to the user as Python objects. Calling a remote service is as easy as calling a method in a Python object.

It is also important to stress that the Jython client differs a lot in purpose and structure compared to the admin tool we saw on the previous section. From an architectural perspective, the Jython client does not differ from any service client. The admin tool, on the other hand, does not use the messaging subsystem and relies on JMX

A.3.1 Starting the Jython Client

The Jython client can be started using the jythonclient command in the terminal. Figure A.21 shows a terminal session in a Jython client. The initialization of the Jython client does not take any arguments

[3] For more information on Log4J, please refer to the official website http://logging.apache.org/log4j/

from the user, but is heavily dependent on the contents of its configuration file, located at DSOHOME/conf/jythonclient.properties. We will see details about this configuration file in the following section.

The Jython Client tries to mimic the interaction of the python Read-eval-print loop (REPL). Just like the python REPL, the Jython Client detects when the user starts a block. The REPL understands that the input is starting a block when the last non-space character is a colon. The following lines typed by the user will be all considered as being part of the block until the user types a blank line. Only when a blank line is entered, the Jython client will interpret the whole block.

Also similarly to the python REPL, the Jython client REPL is terminated when the user issues the "exit()" command.

A.3.2 Non-interactive mode

It is also possible to ask the Jython Client to execute a script and exit. This is what we call a non-interactive Jython client session. Scripts should be also written in the Python programming language. There are two ways to pass a script to the Jython client: through Unix pipes (output redirection) and explicitly by passing the name of a Jython script to the Jython client. Each method is illustrated in Figures A.17 and A.18 respectively.

```
1 echo "service.method()␣\n␣for␣i␣in␣range(10):␣\n\tservice.method(i)\n" |↩
      jythonclient
```

Figure A.17: Jython client using pipe

```
1 jythonclient --script script.py
```

Figure A.18: Jython client using script file

In Figure A.17, the -e option in the echo command activates the interpretation of escape characters such as "
n" for the new line character. Python uses line breaks and indentation as part of its syntax to delimit blocks such as class definitions, loops, and function declarations.

A.3.3 Jython Client configuration file

The configuration file will determine how the Jython Client will initialize.

Service connections

A service connection is a local Python object that serves as a proxy to a remote service. Using service connections is actually the most important feature of the Jython Client.

It is possible to create any number of service connections.

Start up script

This is a script that is executed when the Jython Client is initialized. Java properties files allow for properties to have multiple lines. Then although the start up script is a single property in the configuration file, it can contain several Python commands. The source code in Figure A.20 is an example of how to use this property to show a welcome message.

```
1 startupScript=print "Welcome␣to␣the␣Jython␣Client" \
2 from datetime import datetime \
3 print "Current␣time␣is:␣" + str(datetime.now())
```

Figure A.19: Stopping container with UUID 0000-00-00-00-000003

Java properties files allow property values to be declared in many lines using the backslash character. But the location of the backslash character does not add a new line character to the value of the property. The

backslash character is just a way to declare a long variable name using more than one line in the properties file.

Command aliases

The configuration script allows users to define command aliases. Each alias is represented by a name, which can be any string. Users can call the alias by typing two dot characters followed by the alias name. Figure A.20 shows how to declare command aliases.

```
1  command.a=print "this␣is␣a"
2  command.0=print "this␣is␣0"
3  command.1=
4  command.2=
5  command.3=print "This␣is␣command␣3!!"    \
6  for i in range(11):                       \
7      tprint "Calling␣decrease()"           \
8      tprint "xxx"
9  command.4=
10 command.5=
11 command.6=
12 command.7=
13 command.8=
14 command.9=
15 command.10=print "this␣is␣10"
```

Figure A.20: Stopping container with UUID 0000-00-00-00-000003

Log4J configuration

Log4J needs to be configured too and this is done on the same file, just as we saw in the configuration of containers. Here too, all configurations related to Log4J have the prefix 'log4j.'.

A.3.4 Example of Jython Client usage session

First the user is starting the Jython Client by calling the 'jythonclient' command. Then the Jython Client shows a welcome message and shows that it is loading services. After that the client lists all command aliases.

A.4 Variables in configuration files

When we store log messages in files, an immediate problem we need to solve is how to represent the file path in a way that keeps the configuration portable. In other words, we want the configuration to be easily adaptable to other environments. This requirement forces us to use relative paths. So we can assign the property "log4j.appender.FILE.File" with a string that starts with the value of the variable "dso.home". For example: "log4j.appender.FILE.File=$dso.home/../data/

A.4.1 List of variables

At follows we list the available variables.

dso.home - The directory that contains the DSO installation. In the Jython Client, the value of this variable is automatically detected by the shell script that calls the JVM and the value of the variable can be overridden by using the "–basedir" option.

user.dir - The directory from which the JVM was executed. This variable is the same provided by the Java programming language.

user.home - The home directory of the user who is running the JVM. This variable is the same provided by the Java programming language.

```
user@host:~/# jythonclient
################################
#                              #
#         Jython client        #
#                              #
################################

Classes PM, PMTree, VM, VMState, and CommandLineLib are pre-imported.
Type "exit" to leave

Loading services...

Loading: myservice (x.y.z.MyService)

Loading pre-defined command aliases:

..1 :: myservice.x("Testing")
..2 :: UUID.randomUUID()
..3 :: print "This is cool"
..4 :: myservice.draw(20, 20)
..5 :: myservice.draw(21, 21)
..6 :: myservice.draw(22, 22)
..7 :: dir()
..8 :: ping("C1")
..9 :: ping("C2")

Loading the startupScript:

Welcome to the Jython Client

> myservice.a()
Response from myservice
> ..2
This is cool
> exit()
user@host:~/#
```

Figure A.21

A.5 Limitations

There are limitations to our middleware. Here we list the main limitations, which were not yet fully implemented for us to focus on the functionality and our proof of concept.

A.5.1 Inner classes

Inner classes are not supported by the pre-processor. This is because, in the context of defining artifacts for the DSO, inner classes do not add any new way to specify artifacts. Also, supporting the syntax of inner classes requires an extension of our grammar. Visibility of inner classes can be too limited, which does not make those classes interesting for distributed objects. Nevertheless, we are planning to support inner classes in services in the future in order to fully support the standard Java grammar.

Appendix B

Distributed Middleware Implementation Details

In this appendix we explain some implementation details of the middleware. Some of the decisions we made during the course of the research were based on issues we found during the construction of the middleware. Therefore, this appendix sheds light on the causes of those decisions.

B.1 Messaging Subsystem

In our middleware, we present an abstract model that is different from the JMS one. In our model, as it can be seen on Figure B.1, a user object represents a certain user A. From the perspective of this object, a messaging subsystem is essentially an object that exposes some methods to manage message visitors and that can be used to send messsages.

Such messaging object is obtained through a MessagingFactory object, which is obtained through the MessagingSystem class. Messaging factory is part of the driver and is in charge of controlling the instantiation of messaging objects. Control, in this case, means giving messaging object to authorized users and trying to use message exchange resources with austerity. For instance, two requests to establish identical connections to the message exchange system should point to the same messaging object.

For the middleware, messages are Data Transfer Objects (DTOs) related to a message type. A message DTO is an object of a class that extends the MessageDTO class, while a message type is a value of an enumeration.

Details about how messages are delivered depend on the driver and underlying message exchange mechanism. In this section we present the JMS driver, which serves as an illustration.

B.1.1 JMS Driver

The JMS driver is responsible for creating connections to JMS and present this connections to the rest of the middleware. At present, we tested the driver only against the ActiveMQ implementation of JMS.

Figure B.1 is a diagram of the internal functioning of such driver. JMS cannot be used directly by our middleware since JMS model is based on sessions, which manage connections with a message exchange system.

Compatibility

As the driver was not build utilizing any class that is specific to ActiveMQ, it should work with other implementations of JMS but tests are necessary to ensure full compatibility. Nevertheless, we needed to add an ActiveMQ agent to the JMS network in order to provide additional functions to manage topics (create and delete topics, and change access rights to users on-the-fly).

As already stated, JMS is an API specification instead of an actual message delivery mechanism or protocol. That means that clients written to use JMS are, at least in theory, capable of exchanging messages utilizing any implementation of JMS, utilizing any underlying message transport.

Several message queues that offer a JMS API allows for utilization of several message delivery mechanisms. ActiveMQ, for instance, can utilize reliable multicast or in-memory messaging.

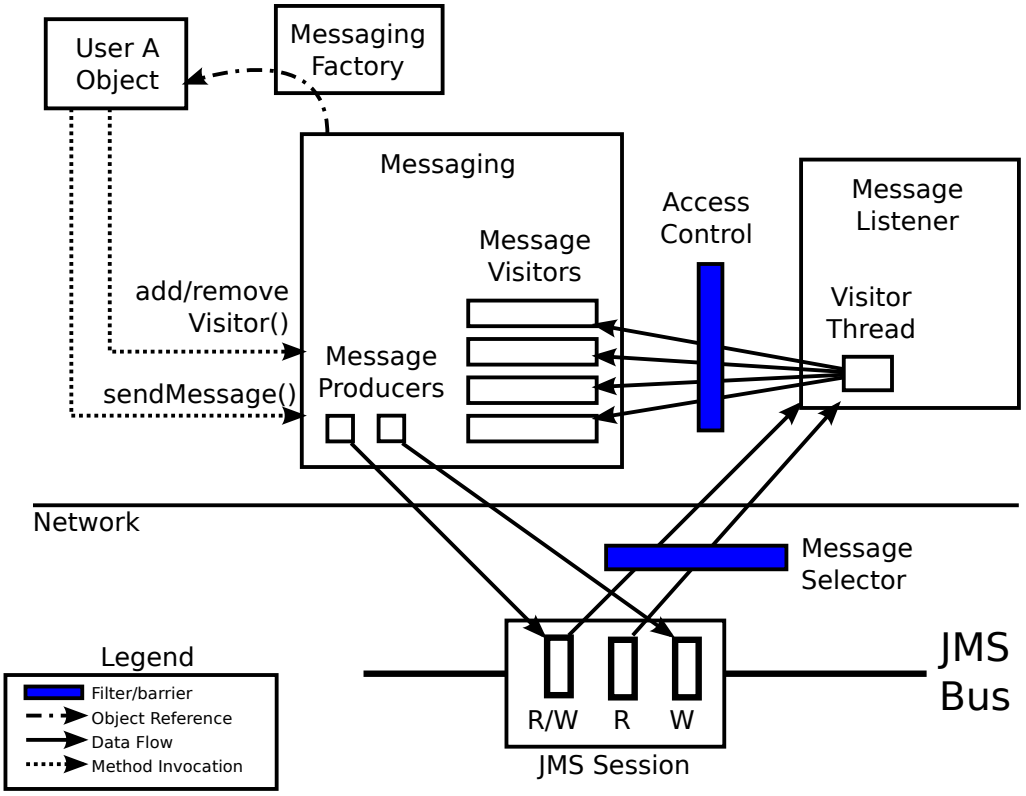

Figure B.1: Structure of the JMS driver.

ActiveMQ-specific Aspects

Each JMS implementation can define its own solutions for common problems such as failure tolerance. High availability is already supported by ActiveMQ utilizing the 'failover:' URL prefix. Adding this prefix to the URL used to connect to ActiveMQ causes the JMS clients to try to reconnect in case the JMS server becomes unreachable for any reason. In these cases, the ActiveMQ driver will try reconnection automatically, which does not impact our JMS driver in Figure B.1.

B.2 Class Loading

Class loading is one of the biggest concerns when trying to migrate classes over the network. In this section we introduce details about a simple dynamic class loader we had to create for the DSO middleware.

B.2.1 Dynamic Class Loader

One needs to replace the standard class loader in order to change the class loading mechanism of a JVM. This can be done either by providing the -Djava.system.class.loader argument to the java command or by calling the java.lang.Thread.setContextClassLoader(ClassLoader) method.

The first method replaces the system class loader. So all threads in the JVM will use this class loader to load classes. The second method changes the class loader of a single thread.

Both methods will still use a parent class loader, which is the class loader Java uses to load the JVM.

This creates a difficulty, since this parent class loader may be automatically activated when a certain class was not found by the child class loader. Our solution was to start the JVM using the first method while giving to the parent class loader no location of any Java Archive (JAR) file.

There are some features missing in our implementation of a class loader. An important missing feature is the ability to load agents during the JVM live phase. We are using the clause "-javaagent:$JAVA_HOME-/jre/lib/management-agent.jar" when starting a JVM in order to avoid a bug triggered by the JMXUtility, on this line: "vm.loadAgent(managementAgentJAR);".

This line asks the remote JVM (or, in our case, a local JVM accessed using the loop back virtual network interface) to load an agent during the live phase of a JVM. The system classloader is supposed to

load the agent and add the agent JAR to the class path dynamically. But our custom class loader (called org.dcl.DynamicClassLoader) is not doing something it should in order to add the agent and yet allow for the JVM to detect this. A workaround we found was adding the following:

```
1 me.equals("sun/jvmstat/perfdata/resources/aliasmap")) {
2
3 management.Agent.agentmain("");
4 h(Exception e) {
```

Figure B.2: A bad fix

But this solution does not work in every case. So instead of solving this problem, we are leaving the actual solution for later and for now we simply load the agent although this agent is not needed for now. The current solution is not that bad, since it avoids having the JMXUtility to have to load remote agents.

B.2.2 Security

It is easy to see that allowing a class obtained remotely to be executed locally can become a security breach. For instance, a remote class may have been build in a malicious way and damage some local resource, or even be exploited to expose sensitive data.

Security was not a concern addressed on this research. Here our focus was on the inter-operation of mobile software agents and its implications in the programming environment. Nevertheless, we can outline how security at this level could be controlled.

First, all exchanged JAR files should necessarily be digitally signed by the sender. The identity of the sender should necessarily be validated by a chain of certificate authorities whose first node should be a root authority trusted by the local node. Each node should have one or more root certificates in order to be able to build chains of certificate authorities.

If a chain can be built, the local node should execute classes in the JAR file in a controlled JVM.

Appendix C

Distributed Middleware Programmers' Manual

This appendix is a manual for programmers interested in participating on the development of the DSO project. Reading this appendix is also useful to understand some of the decisions we made when we developed the DSO middleware.

This appendix differs from the in that this one was written having the programmer in mind. The programmer needs to know all the internals of the system (which is covered on sessions) but also

C.1 Coding Conventions

C.1.1 Class naming

MV suffix - Message Visitor - A class that visits one or more kinds of messages. DTO suffix - Data Transfer Object - Every DTO is a subclass of MessageDTO and should be identified by the "DTO" suffix.

C.1.2 Comments

Comments in Java source code should go until the column 100, despite of the indentation.

The word "TODO" should be placed at the beginning of a comment to mark it as something still to be resolved. The word "FIXME" has the same usage, but points to something of a more pressing priority.

C.2 Important Classes in the API

C.2.1 MessageDTO

A message between two nodes is represented by a org.dso.messaging.dto.MessageDTO object.

Each kind of message exchanged between nodes is an object of a class that extends the MessageDTO. The MessageDTO class has the "validate()" abstract method, which is called by all "init()" in order to verify if the DTO is complete. During the life cycle of a DTO, this object may have its internal state changed several times before delivery. Calling this method is a way to prevent a DTO in an invalid state to be transferred through the network.

C.3 Mock Messaging

The mock messaging (the org.dso.messaging.drivers.mock.MockMessaging class) is the message driver we use during unit testing. Instead of actually transferring messages through the network, the mock messaging transfer messages within the same JVM. By using the mock messaging, we can start several containers on the same JVM and make them interact. This is ideal for unit tests (such as the ones using the JUnit tool).

On the one hand, this prevents us from having to start several JVMs in order to test the interactions between networked elements. On the other hand, tests conducted using the mock messaging can not simulate many situations that occur in a networked set up.

The main limitations of the mock messaging are: sharing of the system class loader and sharing of operating system resources. Actually, the limitations in the mock messaging was the main motivation for us to write the integration tests.

When two or more containers share the same class loader the behavior of the middleware is different from the real usage of containers. In a real scenario, each container starts up with a set of classes it can reach. Because the class loader is the same, the class loader cannot reproduce the effect of containers having access to different sets of classes.

Also, as we pointed out, using the mock messaging causes all containers to share operating system resources. It is not possible, for instance, to precisely estimate the memory footprint of containers when the mock messaging is in use.

C.3.1 Reset

Puts the mock messaging in the initial state, cleaning the log of messages transferred.

C.3.2 Dump

Generates a dump of everything that happened with the mock messaging since the last reset.

C.3.3 Reset Event Log

The resetEventLog() method resets only the event log, leaving the network topology intact. This method is useful when used in combination with the dump() method. The resetEventLog() method can be called to make the next call to the dump() method print only messages that were sent since the last call to the resetEventLog() method. It is then easier to see which messages were transferred in each section of tests.

MockMessaging.resetEventLog(); /* * Perform some actions that uses the messaging system * to send and receive messages. */ System.err.println(MockMessaging.dump());

C.3.4 Get sent messages

Retrieves a list of the messages transferred since the last reset.

C.3.5 Mock Messaging Authorization

Authorizations can be given and revoked at run time using the org.dso.messaging.drivers.mock.MockMessaging-Authorization class. This is done by the org.dso.DSOAllTests.setUpLoggerAndResetEverything() method as we will see in the next section.

C.3.6 JUnit Tool

When we write unit tests, it can easily become complex to check long sequences of messages transferred through the mock messaging. Although the mock messaging driver allows us to retrieve every message object that was transferred, the source code of the test case can easily become long, hard to read, and therefore hard to maintain or expand.

We developed the Mock Messaging JUnit Tool in order to simplify checking such sequences of messages. Instead of programmatically checking each field of a message, which is cumbersome and, as already stated, hard to read, this JUnit Tool allows us to represent each expected message as a string in the same format generated by MessageDTO.toString().

Figure C.2 shows an example of such string representation. All white space characters are simply ignored, making it easy for the programmer to indent the string the way the programmer finds more readable. This is particularly important when a set of string representations of messages are put together in a file, as we will see later.

The syntax of the string representation is simply a series of properties of the message put in an order that makes it readable: the user who sent the message, the UUID of the node that originated the message, the topic through which the message was sent, the recipient of the message, and the message type between parenthesis.

The recipient can be either a single node or a group. If the recipient is a single node, then the UUID of the node should be placed on the right side of the arrow. If the recipient is a group, then the name of the group should be placed between angle brackets on the right side of the arrow.

The type information is optional and can be expressed both as "(<message type >/ <DTO class name >)" or simply as "(<message type >)". The DTO class name is only checked against the expected DTO class, as specified in the MessageType enumeration.

```
1 admin : 0000-00-00-00-000600 ===[ C1 ]===> 0000-00-00-00-001000 (↩
    REMOTE_SERVICE_OBJECT_REFERENCE_RESPONSE/org.dso.service.reference.↩
    ServiceObjectReferenceDTO)
```

Figure C.1: Example of string representation of a message

With the exception of the message type parenthesis (which is optional), every element can be replaced by an asterisk sign to represent a wildcard. No verifications will be made for all location in which asterisks are present. This is useful in situations in which there is uncertainty about the outcome of a message. For example, when node 0000-00-00-00-000600 sent a message to the DBS group asking if any server is alive, the first reply can come from any server in this group. So if we do not want the exact sequence to be checked, we can simply check the sequence in Figure C.2. In this figure we are simply checking that two nodes replied the message from 0000-00-00-00-000600, we do not care what is the UUID of each node or which node sent the first message.

The string representation does not allow us to represent the contents of the message. Checking the contents of the messages can be done by retrieving each message from the Mock Messaging using the getTransferredMessages() method. The focus of the JUnit Tool is not checking the correctness of the data that was transferred, but instead to check if the right kind of message was transferred from the right node to the right destination in the right order.

```
1 #
2 # Checking that a query is followed by two responses
3 #
4 admin : 0000-00-00-00-000600 ===[ C1 ]===> <DBS>                (↩
    SERVICE_PROVIDER_QUERY)
5 *     : *                    ===[ C1 ]===> 0000-00-00-00-000600 (↩
    SERVICE_PROVIDER_RESPONSE)
6 *     : *                    ===[ C1 ]===> 0000-00-00-00-000600 (↩
    SERVICE_PROVIDER_RESPONSE)
```

Figure C.2: Example of string representation of a message

As the tool was specifically designed to check long sequences of messages, it is capable of reading a file containing one message description in each line. Figure C.2 also illustrates the format of input files for the Mock Messaging JUnit Tool. The first three lines are comments, as they start with the sharp character. A unit test may invoke the processing of such a file using the MockMessagingJUnitTool.assertMessages() method. The sole argument should be either the name of the file containing the description of the sequence of messages (as in Figure C.2), or a input stream pointing to bytes in such format.

C.4 Structure of JUnit tests

The Maven build will find all unit tests and execute them automatically. But programmers also need to execute the unit tests from inside of an IDE such as Eclipse. When running from an IDE, the programmer can call all unit tests by calling the org.dso.DSOAllTests test suite.

This test class has a static method called org.dso.DSOAllTests.setUpLoggerAndResetEverything(), that prepares the environment for any unit test. Resetting the environment at each test is useful for us to make sure that we eliminate the influence a test may have in another one.

C.4.1 Logging

Logging should be done by test classes using the Log4J tool. All logs are directed to the standard console. Log4J is configured by the org.dso.DSOAllTests.setUpLoggerAndResetEverything() method to write

all output to the standard console. It is possible to change the log level by changing the setUpLoggerAndResetEverything() method.

Structure of a typical test case class is shown in Figure C.3.

```
1  public class MyTest {
2
3    @Before
4    public void setUp() throws Exception {
5      DSOAllTests.setUpLoggerAndResetEverything();
6    }
7
8    @Test
9    public void test1() {
10     Logger logger = Logger.getLogger(this.getClass());
11     logger.info(MockMessaging.dump());
12     assertEquals(102, MockMessaging.getSentMessages().size());
13   }
14 }
```

Figure C.3: Typical unit test class

C.5 Integration test script

The integration test script is a bash [1] shell script.

For more information on how to call the integration test script, call the script without any argument. The script will print a detailed message explaining all arguments that can be passed to it. You can also check the header of the script's source code.

C.5.1 Interactive session

When a test fails, debugging the test by printing messages on screen can be extremely time consuming as each run of an integration test can take several minutes to complete. If we identified that a certain line n fails to execute, the best way to find the problem is by analyzing the bash process (the process that executes the integration test) while the process is still active.

We provide the "interactive" function to allow for developers to interact with the bash process this way. A call to the "interactive" function causes the integration test script to stop and enables the user to interact with a bash console. In the interactive session, the user can manually check the state of the containers, log files, and output files.

The idea behind the "interactive" function is that this function should only be in the integration test script temporarily while the programmer searches for the causes of some defect in DSO or in the integration tests themselves. As soon as the defect is found, the programmer should remove the calls to the "interactive" function, so subsequent calls to the integration tests finish without interruptions.

If sound was enabled using the –sound argument, the integration script will also play a sound file when a break point is reached. This feature is to avoid making the user wait several minutes for the interactive session to be reached. If the rich output is enabled, the command line is printed in bold to ease identification of the commands entered.

During the interactive session the user actually interacts with the same bash instance that was executing the integration tests. Therefore all functions and variables available during the execution of the integration test script will be available during the interactive session. Also, all changes in the variables' values and function declarations during the interactive session will be available for the integration test script if the script is resumed after the interactive session finishes.

Calls to the "interactive" function can be placed anywhere after the preamble section. This section is identified in the integration test script using some comment lines.

There are three different ways to terminate an interactive session:

[1] Bash is a command line interpreter originally based on the Bourne shell, and is currently maintained by the Free Software Foundation.

- By typing “ok” or “resume”. The interactive session will terminate and the integration test script will resume from where it stopped.
- By typing “kill”. The integration test script will be immediately interrupted and the rest of the script will be not executed.
- When a timeout is reached. If the user does not issue any command before a certain timeout, the interactive session automatically terminates and the integration test script is resumed. Reaching this timeout has the same effect as typing “ok” or “resume”. The INTERACTIVE_TIMEOUT variable in the test.config file sets the length of the timeout. The value of this variable is the number of seconds in the timeout.

C.5.2 Continuous mode

Unfortunately, not all tests are deterministic. Depending on a number of factors, it is possible that some execution errors happen intermittently, making it impossible to completely reproduce the integration tests. The best practice when we have this sort of defect is to find what causes the test to have this random behavior and to eliminate the source of randomness.

But the very source of instability is also a defect. So we cannot expect that the solution for this problem to be easy to address. There are cases in which the only option we have is to execute the integration tests until the random error happens. The integration test script can be executed in an infinite loop to help developers facing this situation.

This is what we call the continuous mode of execution. When the “–continuous” option is passed to the integration test script, the script will start a loop that executes the integration tests until the tests finishes with a return code different from zero or until the user terminates the script (which can be done by sending a kill signal through the kill command, or interactively using Control+C key combination).

The continuous execution attempts to reproduce several independent executions of the integration tests. For the executions to be independent it is important that we minimize the influences one execution will have on the next one. For that reason, the continuous execution of the integration test creates a new Bash process for each test. This way, we avoid having the variables defined during an integration test execution influencing the next integration test execution.

The continuous mode can be prevented from running by passing the “–non-continuous” option as an argument to the integration test script. When both “–continuous” and “–non-continuous” options are available, the integration test script will execute in a non-continuous way. This option has actually two uses.

The first use is for cases in which the user defined an alias to get the integration test script to run in continuous mode by default. In other words, if the user defined a command line alias that maps the “integration_test” script to the “integration_test –continuous” command. The “–non-continuous” option can be used to cancel this default when the user needs a single execution of the integration test.

The second use of this option has to do with the structure of the integration test script. For simplicity, this script calls itself (even if this call is on a new bash instance, as we stated above) passing all received arguments to the new script execution process. The “–non-continuous” argument is added to the list of arguments in order to prevent this other instance to also execute in continuous mode, since the “–continuous” argument was not removed from the argument list.

C.5.3 Logging

Limiting log file sizes

Logging is important during the integration tests. When we generate log files correctly, we can easily find the causes of problems in our tests. DSO uses the Log4J tool to generate logs, as we saw on Section A.

But the log outputs are not intended to be kept long after they were generated, as in a real production environment. Instead, all log files generated during the integration tests are intended to be checked by the user and discarded as soon as the user does not need them anymore. It is common that when a test finishes, some agents or containers remain running. The side effect may be the log files getting bigger and bigger out of control. For example, those pieces of software periodically generate log messages from the ActiveMQ classes, even if there is no activity in the DSO system.

To avoid having log files taking too much hard disk space, we configured all log files to have the limitations shown in Figure C.4. The MaxFileSize variable limits the maximum size of a log file, while the MaxBackupIndex tells Log4J how many backups of old log files to keep when the maximum size of a log file

is reached. When this variable is zero, Log4J will not keep any backup and will simply truncate the log files to fit in the maximum size, discarding the oldest messages.

```
1 log4j.appender.FILE.MaxFileSize=1MB
2 log4j.appender.FILE.MaxBackupIndex=0
```

Figure C.4: Log4J file size limitation

Log output performance

Currently, we are using the pattern conversion in Figure C.5. The meaning of each token in this pattern is documented in the Log4J API documentation[2]. What we want to underline here is the "%L" token, which outputs the line number that generated the log. The "%c" token prints the class that generated the log. Having the line number in the log messages makes it easy for programmers to navigate the source code based on the contents of the log files.

But, according to the Log4J documentation , using the "%L" conversion character is extremely slow. We are using this conversion character despite of this performance problem since in the integration tests we focus on checking the functionality, not the performance. Nevertheless, it is important for the developer to know about this issue.

```
1 log4j.appender.FILE.layout.ConversionPattern= %d - %-4r [%t] %-5p %c(%L)↩
    %x - %m%n
```

Figure C.5: Pattern conversion that formats Log4J messages

Advanced log visualization with Chainsaw

Figure C.6 shows the contents of a configuration file that creates a logger that sends log messages to a log server through the network.

```
1 log4j.rootLogger=DEBUG, SERVER, SOME_OTHER_LOGGER
2 log4j.appender.SERVER=org.apache.log4j.net.SocketAppender
3 log4j.appender.SERVER.Port=4445
4 log4j.appender.SERVER.RemoteHost=localhost
5 log4j.appender.SERVER.ReconnectionDelay=10000
```

Figure C.6: Log4J configuration to use remote log server

C.6 Message debugging

One of the key aspects of DSO is the message exchange subsystem. It is important that the developer knows how to verify if messages are being transferred correctly. In this section we list the main tools a programmer can use to check messages.

C.6.1 HermesJMS

HermesJMS is a free software tool to allow for users to interact with JMS systems. The tool makes it easy for users to create and send JMS messages and to visualize messages exchanged in message topics and message queues. We use HermesJMS mainly for debugging the DSO protocol.

The kind of debugging that HermesJMS provides allows us to check the contents of messages: its fields and payload. For instance, if we are to change the DSO protocol and add a new field to messages for any

[2]Log4J pattern layout: http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/PatternLayout.html

particular reason, the first tests we should conduct would be executed while the programmer can access the history of messages exchanged.

A more high level test of message exchange uses expected messages lists, as in Section 8.6.2. Expected messages lists are concerned with the types of messages, the destination and origin. This differs from the sort of fine-grained data that can be visualized using HermesJMS.

It is also possible to visually inspect the sequence of messages to search for inconsistencies, but such process takes too much time to do manually and is quite error prone.

Appendix D

Detailed compatibility example

In this apendix we will show a complete example of compatibility processing to illustrate the process in details. We believe this apendix may be useful for readers that want to review the algorithm we explained.

During explanation text, long examples break text fluency, so that is the reason why we decided to place a longer example as an apendix.

D.1 Client source code

```
1  method x() {
2    m1();
3    if(/*condition*/) {
4      m2();
5      loop(/*repetition arguments*/) {
6        m3();
7        m4();
8        if(/*condition*/) {
9          m5();
10         m6();
11         m7();
12       } else {
13         m8();
14       }
15       m9();
16     }
17     m10();
18   } else {
19     m11();
20   }
21   m12();
22   m13();
23   loop(/*repetition arguments*/) {
24     m14();
25   }
26   m15();
27 }
```

Figure D.1: Client source code for the detailed example

```
1  (BLOCK A
2  A1: m1
3  A2: (COND
4  CK B
5  B1:          m2
6  B2:          (LOOP
7  CK C
8  C1:                   m3
9  C2:                   m4
10 C3:                   (COND
11 CK D
12 D1:                            m5
13 D2:                            m6
14 D3:                            m7
15
16 CK E
17 E1:                            m8
18
19
20 C4:                   m9
21
22
23 B3:          m10
24
25 CK F
26 F1:          m11
27
28
29 A3: m12
30 A4: m13
31 A5: (LOOP
32 CK G
33 G1:          m14
34
35
36 A6: m15
37 )
```

Figure D.2: Client source code for the detailed example

D.2 Client source code reduced AST

D.3 Step by step solution

A1 means line 1 of subtree A. A! means the solution of the subtree A, meaning that the subtree will be reduced to one or more linear transformations. A means transformation of a LOOP subtree into a BLOCK subtree.

```
A1: (BLOCK A m1)
A2: (BLOCK A (BLOCK B m1) (BLOCK F m1))
A3: (BLOCK A (BLOCK B m1 * m12) (BLOCK F m1 * m12))
A4: (BLOCK A (BLOCK B m1 * m12m13) (BLOCK F m1 * m12m13))
A5: (BLOCK A (LOOP G (BLOCK B m1 * m12m13) (BLOCK F m1 * m12m13)))
A6: (LOOP G (BLOCK B m1 * m12m13) (BLOCK F m1 * m12m13) * m15)
B1: (LOOP G (BLOCK B m1m2 * m12m13) (BLOCK F m1 * m12m13) * m15)
B2: (LOOP G (BLOCK B (LOOP C m1m2) * m12m13) (BLOCK F m1 * m12m13) * m15↩
    )
B3: (LOOP G (BLOCK B (LOOP C m1m2) * m10m12m13) (BLOCK F m1 * m12m13) * ↩
    m15)
F1: (LOOP G (BLOCK B (LOOP C m1m2) * m10m12m13) (BLOCK F m1m11 * m12m13)↩
     * m15)
F!: (LOOP G (BLOCK B (LOOP C m1m2) * m10m12m13) m1m11m12m13 * m15)
C1: (LOOP G (BLOCK B (LOOP C m1m2 : m3) * m10m12m13) m1m11m12m13 * m15)
C2: (LOOP G (BLOCK B (LOOP C m1m2 : m3m4) * m10m12m13) m1m11m12m13 * m15↩
    )
C3: (LOOP G (BLOCK B (LOOP C m1m2 : (BLOCK D m3m4) (BLOCK E m3m4)) * ↩
    m10m12m13) m1m11m12m13 * m15)
C4: (LOOP G (BLOCK B (LOOP C m1m2 : (BLOCK D m3m4 * m9) (BLOCK E m3m4 * ↩
    m9)) * m10m12m13) m1m11m12m13 * m15)
D1: (LOOP G (BLOCK B (LOOP C m1m2 : (BLOCK D m3m4m5 * m9) (BLOCK E m3m4 ↩
    * m9)) * m10m12m13) m1m11m12m13 * m15)
D2: (LOOP G (BLOCK B (LOOP C m1m2 : (BLOCK D m3m4m5m6 * m9) (BLOCK E ↩
    m3m4 * m9)) * m10m12m13) m1m11m12m13 * m15)
D3: (LOOP G (BLOCK B (LOOP C m1m2 : (BLOCK D m3m4m5m6m7 * m9) (BLOCK E ↩
    m3m4 * m9)) * m10m12m13) m1m11m12m13 * m15)
D!: (LOOP G (BLOCK B (LOOP C m1m2 : m3m4m5m6m7m9 (BLOCK E m3m4 * m9)) * ↩
    m10m12m13) m1m11m12m13 * m15)
E1: (LOOP G (BLOCK B (LOOP C m1m2 : m3m4m5m6m7m9 (BLOCK E m3m4m8 * m9)) ↩
    * m10m12m13) m1m11m12m13 * m15)
E!: (LOOP G (BLOCK B (LOOP C m1m2 : m3m4m5m6m7m9 m3m4m8m9) * m10m12m13) ↩
    m1m11m12m13 * m15)
C!: (LOOP G (BLOCK B m'1 m'2 m'3 * m10m12m13) m1m11m12m13 * m15)
B!: (LOOP G m'1m10m12m13 m'2m10m12m13 m'3m10m12m13 m1m11m12m13 * m15)
G1: (LOOP G m'1m10m12m13 m'2m10m12m13 m'3m10m12m13 m1m11m12m13 : 14 *↩
    m15)
G~: (BLOCK G m'4 m'5 m'6 m'7 * m15)
G!: m'4m15 m'5m15 m'6m15 m'7m15
```

Figure D.3: Step by step solution

Appendix E

Building it all

All software we developed is distributed in its source code and therefore we need to document how to build it. Our software is not a monolithic project, but it is made of smaller interdependent modules. A distribution of all pieces of software in this research is organized in a directory tree. Hereinafter we will refer to the root node of this directory tree as ROOT.

Before presenting each module and the dependencies between the modules, we understand that the reader may have reached this appendix only to get guidance about how to start using the deliverables of this project. Therefore, we first present a simple method to build all deliverables and later we explain the deliverables in details.

E.1 General Build

All build is managed by the Maven tool [7]. In order to build all artifacts, one only needs to execute the mvn command at the ROOT directory. Figure E.1 illustrates how to call Maven in a Linux system. The mvn command will automatically recognize a configuration file called pom.xml in the ROOT directory and build all modules according to the instructions in this file.

```
1 user@host-name:/# cd ROOT
2 user@host-name:ROOT# mvn
```

Figure E.1: Execution of the Maven build process

Figure E.2 shows a successful output of the execution of this script. When all modules were built correctly, Maven will print the message "BUILD SUCCESS" as in Figure E.2. As a result of a successful build execution, Maven creates the distribution files:

1. ROOT/dso-00-distribution/target/dso-0.01-bin.tar.gz
2. ROOT/dso-00-distribution/target/dso-0.01-bin.tar.zip

These files contain everything one needs to execute a DSO container. Details about how to start a container can be found in Chapter A.

```
1 [INFO] ------------------------------------------
2 [INFO] BUILD SUCCESS
3 [INFO] ------------------------------------------
4 [INFO] Total time: 15.081s
5 [INFO] Finished at: Sun Jun 24 18:16:23 JPT 2012
6 [INFO] Final Memory: 23M/179M
7 [INFO] ------------------------------------------
```

Figure E.2: End of the output of a successful Maven build

Maven is based on Java and is a free software multi-platform building tool. The execution of mvn will yield zero as the return code if no problem was found. A non-zero will be yielded if any error prevented the Maven script from terminate successfully. This feature makes it easy for an external script to call Maven and to check if the execution was successful.

The build process may take a few minutes. During this process Maven compiles all source code and execute all unit tests. During the build process Maven will check if all dependencies (all JAR files that are needed by DSO classes) are available in the local repository. If any dependency is not available, Maven will download it. Therefore, the first run of the Maven build may take a long time if Maven needs to download a lot of files.

Maven creates all of its artifacts in directories called "target" inside of each sub-project. Calling "mvn clean" will clean these directories. Cleaning these directories enables the user to eliminate the eventual influences of a previous build.

E.1.1 Integration tests

Although we have tried to cover most of the possible sources of defects in our software, it is not possible to fully test our software using JUnit alone. For example, it is not possible to check if our software will behave correctly when it uses the ActiveMQ as the underlying message delivery infrastructure. We need then another kind of test that mimics a real usage of the software. We call these tests the integration tests.

We also provide a script that conducts the integration tests automatically. Calling the integration tests is as simple as calling this script. Figure E.3 shows how to call this script. Calling it without any argument will get the script to print all possible arguments.

```
1 user@host-name:/# ROOT/dso-00-distribution/src/test/bin/integration_test↩
      --test
```

Figure E.3: Starting the integration tests

These tests will check if a DSO distribution was already built by a previous call to "mvn". If such distribution exists, the script will create an instance of the ActiveMQ server, several containers connected to this ActiveMQ server, and will call some services on these containers.

Figure E.4 shows the end of the output of a successful execution of the integration tests. The output shows a message printed by the admin tool. This message shows the details of a currently running container. The integration tests leave an instance of ActiveMQ and an instance of a container, which can be used by the user to manually execute any further test.

```
1  Found 1 container:
2
3  0.       Node UUID: 00000000-0000-0000-0000-000000000002
4  Service name: example.CounterService
5  PID: 31552
6  Up time: 3949 ms
7  Display name: org.dso.service.container.Container ROOT/dso-00-↩
       distribution/target/integration_tests-tmp/data/dso000002.properties
8
9  SUCCESSFUL!
10 Elapsed time: 82 seconds
```

Figure E.4: End of a successful execution of the integration tests

If the user does not want to use the ActiveMQ instance and the container, the user can simply call the integration test with the "clean" argument, as in Figure E.5. This command will stop the DSO containers, stop the ActiveMQ process, and remove all files related with the integration tests, which are located at ROOT/dso-00-distribution/target/integration_tests-tmp/.

```
1 user@host-name:/# ROOT/dso-00-distribution/src/test/bin/integration_test↩
      --clean
```

Figure E.5: Cleaning an integration test

E.1.2 Build and integration tests in a single command

We have also prepared the integration_test command to call the mvn command. This is convenient for cases in which we want to completely build the modules from the source code and perform the integration tests. Doing it all in a single step is particularly useful for developers while changing the source code of our modules. Figure E.6 shows how to print the help of the integration test script. As the help message shows, calling integration_test –mvn-clean-mvn-test is the most complete option. It cleans previous builds and performs the complete build and integration tests.

```
1  user@host-name:/# ROOT/dso-00-distribution/src/test/bin/integration_test↩
       --help
2  This script can perform 4 different procedures. Each procedure is ↩
       activated by a specific argument.
3  Tasks in each procedure are as follows:
4  A    Cleans previous integration tests
5  B    Performs the integration tests
6  C    Calls "mvn␣clean" on the base project
7  D    Calls "mvn" on the base project
8  Usage:
9  ./integration_test --clean                  A
10 ./integration_test --test                   A B
11 ./integration_test --mvn-test               A D B
12 ./integration_test --mvn-clean-mvn-test     A C D B
```

Figure E.6: Cleaning an integration test

E.1.3 Prerequisites

All pieces of software may be built from any terminal in which JDK and Maven are installed. The whole software was written in Java, therefore both build and execution should work in a number of platforms.

Maven dependencies are managed with the use of a repository of JAR files. During the execution of a Maven build script, the execution of the script gathers data to determine the set of dependencies needed for the build. If any dependency of a certain module is not met by another module in the same build process, Maven treats this dependency as external to the build and searches for the JAR file of this dependency in the repository. If the repository already contains this dependency, Maven will proceed with the build process. Otherwise, Maven will download the dependency from a Maven repository on the Internet.

For the user who invokes the build of this project, the consequences of this approach is that the first build may take time, as all dependencies are downloaded from the Internet.

E.2 Modules

The successful execution of the mvn command will generate several JAR files, which are placed in the target directory inside of each module. Maven is built around the concept that each module exports one main artifact. In our modules, all artifacts are JAR files.

In this section we both introduce each module and the location of the JAR file that each module generates as its main artifact. This is the list of all artifacts:

1. ROOT/DMViewer/target/DMViewer-0.1.jar
2. ROOT/DynamicClassLoader/target/DynamicClassLoader-0.1.jar

3. ROOT/hyphenType/target/hyphenType-0.1.jar

When we call the mvn command at the ROOT directory, Maven checks the dependencies between modules before actually start building any module. When we execute the General Build and a certain module a depends other modules b and c, Maven will make sure that the artifacts of b and c are available before starting to build a.

But when we call the mvn command at any specific module (while the current directory is a child of ROOT), Maven will not resolve inter-module dependencies since only the Global Build at ROOT/pom.xml contains the locations of all modules. So if we want to manually build a, we need first to build b and c manually and install b and c in the local Maven repository.

Fortunately, building and installing a module in the local Maven repository can be done simply by calling the "mvn install" command. For instance, assuming that neither b nor c have any dependency on other modules, we can build a using the sequence of commands in Figure E.7. Here we also assume that ROOT is /media/distribution.

```
1 user@host-name:/# cd /media/distribution/b
2 user@host-name:/media/distribution/b# mvn install
3 user@host-name:/# cd /media/distribution/c
4 user@host-name:/media/distribution/c# mvn install
5 user@host-name:/# cd /media/distribution/a
6 user@host-name:/media/distribution/a# mvn
```

Figure E.7: Building the module a manually.

As the general build, the build process of each module is configured to generate the basic artifacts using only the mvn command. But each module may also offer additional artifacts that are built using the mvn command followed by a target name. At follows we present details about each module and their specific targets, when available.

E.2.1 DMViewer

The simulator.

E.2.2 DynamicClassLoader

A class loader that makes it possible to load new items to the class path during run time.

E.2.3 hyphenType

hyphenType is a tool to read command line inputs as in a Linux terminal. We needed to create this tool in order to make it easier for us to create text based programs. Details about this tool are out of the scope of this text and the reader should refer to the documentation generated by Maven.

Site target

The site target will generate a site containing a detailed documentation about the hyphenType tool. The documentation covers not only how to use the tool, but also the Javadocs documentation, and reports on unit test coverage and source code quality. The site target can be called using the command "mvn site:site" and the site is generated as a series of files under the ROOT/hyphenType/target/site directory.

PDF target

The PDF target will generate a PDF file containing the same documentation that is generated in the site target. The PDF target can be called using the command "mvn pdf:pdf" and the PDF is generated at ROOT/hyphenType/target/pdf/maven-pdf-plugin.pdf.

E.2.4 DSO

dso-00-distribution

This project serves two purposes: (1) to create a distribution containing everything a user needs to use the middleware, and (2) to perform an integration test to check, as described before.

dso-01-core

Contains all core classes. All other projects depend on this one, but the dso-01-core project does not depend on any other project.

dso-02-messaging-01-activemq

The ActiveMQ messaging driver. This is a separate project in order to allow for users to chose whether or not they want to use the ActivemQ JMS implementation.

dso-03-client-01-jython

The Jython client project.

dso-04-osintegration-01-linux

An operating system integration ready to create and manage Linux processes.

dso-05-build-01-maven-plugin

The Apache Maven plugin to allow for programmers to use the DSO pre-processor.

dso-06-example-01-counter-service and dso-06-example-02-multiplication-service

Two examples of usage of the DSO middleware. These examples are also used to perform integration tests.

DynamicClassLoader

The dynamic class loader project.

hyphenType

The hyphenType library, which is used to read command line arguments and parse them.

Bibliography

[1] Extensible Markup Language (XML) 1.0 (Third Edition). http://www.w3.org/TR/2004/REC-xml-20040204/#sec-r [Online; accessed 02-February-2014].

[2] Spin - Formal Verification. http://spinroot.com. [Online; accessed 02-February-2014].

[3] GridMPI. http://www.gridmpi.org/, 2010. [Online; accessed 02-February-2014].

[4] UDDI. http://uddi.xml.org/, March 2010. [Online; accessed 02-February-2014].

[5] JUnit. http://www.junit.org/, 2011. [Online; accessed 02-February-2014].

[6] Eiffel. http://www.eiffel.com/, 2013. [Online; accessed 02-February-2014].

[7] Maven. http://maven.apache.org/, 2013. [Online; accessed 02-February-2014].

[8] Enis Afgan. Role of the resource broker in the grid. In *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*, pages 299–300, New York, NY, USA, 2004. ACM.

[9] Stephan Aier, Philipp Offermann, Marten Schönherr, and Christian Schröpfer. Implementing non-functional service descriptions in SOAs. In *TEAA'06: Proceedings of the 2nd international conference on Trends in enterprise application architecture*, pages 40–53, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] Fabien Baligand and Valérie Monfort. A concrete solution for web services adaptability using policies and aspects. In *Proceedings of the 2nd international conference on Service oriented computing*, ICSOC '04, pages 134–142, New York, NY, USA, 2004. ACM.

[11] K. Beck and E. Gamma. Test infected: programmers love writing tests. Technical Report 3(7), 1998.

[12] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Representing, analysing and managing Web service protocols. *Data & Knowledge Engineering*, 58(3):327–357, September 2006.

[13] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic composition of E-services that export their behavior. In Maria E. Orlowska, Sanjiva Weerawarana, Michael P. Papazoglou, and Jian Yang, editors, *Service-Oriented Computing - ICSOC 2003*, volume 2910 of *Lecture Notes in Computer Science*, pages 43–58. Springer Berlin / Heidelberg, 2003.

[14] Lucas Bordeaux, Gwen Salan, Daniela Berardi, and Massimo Mecella. When are two web services compatible? In Ming-Chien Shan, Umeshwar Dayal, and Meichun Hsu, editors, *Technologies for E-Services*, volume 3324 of *Lecture Notes in Computer Science*, pages 15–28. Springer Berlin / Heidelberg, 2005.

[15] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan, M. Leino, and Erik Poll. An overview of {JML} tools and applications. *Electronic Notes in Theoretical Computer Science*, 80(0):75 – 91, 2003. Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS'03).

[16] Rajkumar Buyya, David Abramson, and Srikumar Venugopal. The grid economy, 2005.

[17] Alexey Cherchago and Reiko Heckel. Specification matching of web services using conditional graph transformation rules. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 3256 of *Lecture Notes in Computer Science*, pages 259–262. Springer Berlin / Heidelberg.

[18] Carsten Clauss, Stefan Lankes, and Thomas Bemmerl. Design and implementation of a service-integrated session layer for efficient message passing in grid computing environments. *Parallel and Distributed Computing, International Symposium on*, 0:393–400, 2008.

[19] David Cohen and Michael Fredman. Products of finite state machines with full coverage. In Andrzej Lingas, Rolf Karlsson, and Svante Carlsson, editors, *Automata, Languages and Programming*, volume 700 of *Lecture Notes in Computer Science*, pages 469–477. Springer Berlin / Heidelberg.

[20] K. Czajkowski, D. F. Ferguson, I. Foster, J. Frey, S. Graham, I. Sedukhin, D. Snelling, S. Tuecke, and W. Vambenepe. The ws-resource framework. In *Globus Alliance Whitepaper*, March 2004.

[21] Davide Sangiorgi and David Walker. *The π-calculus: A theory of mobile processes*. Cambridge University Press, 2001.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.

[23] Patricia Derler and Rainer Weinreich. Models and tools for SOA governance. In *TEAA*, pages 112–126, 2006.

[24] Douglas C. Schmidt and Michael Stal and Hans Rohnert and Frank Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, October 2000.

[25] Marlon Dumas, Boualem Benatallah, and Hamid R. Motahari Nezhad. Web service protocols: Compatibility and adaptation. *IEEE Data Eng. Bull*, pages 40–44, 2008.

[26] Catalin Dumitrescu, Ioan Raicu, and Ian Foster. DI-GRUBER: A distributed approach to grid resource brokering. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 38, Washington, DC, USA, 2005. IEEE Computer Society.

[27] Filman R. E. and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Aspect-Oriented Programming is Quantification and Obliviousness, technical report, RIACS*, October 2000.

[28] W. Cirne et al. Grid computing for bag of tasks applications. In *Proceedings of 3rd IFIP Conference on E-Commerce, E-Business and E-Government*, 2003.

[29] International Organization for Standardization. ISO/IEC 14977:1996, 1996.

[30] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. Von Reich. The open grid services architecture, version 1.0. In *Global Grid Forum (GGF), Informational Document*, January 2005.

[31] Ian Foster and Nicholas T. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. ACM Press, 1998.

[32] Ian Foster, Carl Kesselman, Craig Lee, Bob Lindell, Klara Nahrstedt, and Alain Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *In Proceedings of the International Workshop on Quality of Service*, pages 27–36, 1999.

[33] Ian Foster, Yong Zhao, Ioan Raicu, and Shiyong Lu. Cloud computing and grid computing 360-degree compared, 2008.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Elements of reusable object-oriented software. Boston, MA, USA, 1995. Addison-Wesley Longman Publishing Co., Inc.

[35] Philippa Gardner, Robin Milner, and Peter Sewell. Calculi for Interactive Systems: Theory and Experiment. Technical report, 2001.

[36] Xueqiang Gong, Jing Liu, Miaomiao Zhang, and Jueliang Hu. Formal analysis of services compatibility. volume 2, pages 243 –248, July 2009.

[37] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, 29(2):1–12, May 2000.

[38] B. Harbulot and J. R. Gurd. Using AspectJ to separate concerns in parallel scientific java code. In *Proceedings of the 3rd international conference on Aspect-oriented software development, AOSD '04*, 2004.

[39] Matthew Hennessy. *A Distributed Pi-Calculus*. Cambridge University Press, New York, NY, USA, 2007.

[40] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[41] Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 1 edition, September 2003.

[42] C. Kalt. RFC 2810 - IRC architecture description. http://tools.ietf.org/html/rfc2810.html, April 2000. [Online; accessed 18-January-2014].

[43] G. Laszewski, I. Foster, J. Gawor, and P. Lane. A java commodity grid toolkit. In *Concurrency: Practice and Experience 13*, 2001.

[44] R. Greg Lavender and Douglas C. Schmidt. Active object – an object behavioral pattern for concurrent programming, 1996.

[45] R. Greg Lavender and Douglas C. Schmidt. Pattern languages of program design 2. chapter Active object: an object behavioral pattern for concurrent programming, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[46] Gary T. Leavens. Tutorial on jml, the java modeling language. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 573–573, New York, NY, USA, 2007. ACM.

[47] Gary T. Leavens and Yoonsik Cheon. Design by contract with jml, 2006.

[48] K. Lee, A. LaMarca, and C. Chambers. Hydroj: Object-oriented pattern matching for evolvable distributed systems. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA 03)*, October 2003.

[49] P. Leitner, A. Michlmayr, F. Rosenberg, and S. Dustdar. End-to-end versioning support for web services. In *2008 IEEE International Conference on Services Computing, Honolulu, Hawaii, USA*, 2008.

[50] B. Lublinsky. Versioning in SOA. In *Microsoft Architect Journal, Microsoft*, April 2007.

[51] Tim Mackinnon, Steve Freeman, and Philip Craig. Extreme programming examined. chapter Endotesting: unit testing with mock objects, pages 287–301. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[52] M. E. F. Maia, P. H. M. Maia, N. C. Mendonca, and R. M. C. Andrade. An aspect-oriented programming model for bag-of-tasks grid applications. In *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid'07)*, 2007.

[53] P. H. M. Maia, N. C. Mendonca, V. Furtado, W. Cirne, and K. Saikoski. A process for separation of crosscutting grid concerns. In *Proceedings of the 2006 ACM Symposium on Applied Computing (Dijon, France, April 23 - 27, 2006). SAC '06. ACM, New York, NY, 1569-1574*, 2006.

[54] Mathew Hennessy. *A distributed pi-calculus*. Cambridge University Press, 2007.

[55] A. A. M. Matsui and H. Aida. Refinements to task control in the aspect-oriented programming model for computing grids. In *In Proceedings IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, 2009*. IEEE PacRim, 2009.

[56] A. A. M. Matsui and H. Aida. Message domains as an architectural solution to complexity in distributed message-oriented systems. In *2010 International Conference on Future Information Technology - ICFIT 2010*, December 2010.

[57] A. A. M. Matsui and H. Aida. A process to client-service compatibility assessment based on service-side fsms. In *IC4E 2011*, January 2011.

[58] A.A.M. Matsui and H. Aida. Advanced client-service compatibility assessment via analysis of references to service-side fsms. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 69–77, 2011.

[59] A.A.M. Matsui and H. Aida. A contract-centric approach to compatibility assessment in highly distributed services. In *Journal of Information Processing – Information Processing Society of Japan*, 2014 – under review.

[60] A.A.M. Matsui and H. Aida. A process calculus approach to a distributed middleware. In *Journal of Information Processing – Information Processing Society of Japan*, 2014 – under review.

[61] B. Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992.

[62] A. Mishra and A.K. Misra. Formal aspects of specification and validation of dynamic adaptive system by analyzing execution traces. In *Engineering of Autonomic and Autonomous Systems (EASe), 2011 8th IEEE International Conference and Workshops on*, pages 49–58, 2011.

[63] Surya Nepal, John Zic, and Thi Chau. Compatibility of service contracts in service-oriented applications. *Services Computing, IEEE International Conference on*, 0:28–35, 2006.

[64] Fu Ning, Zhou Xingshe, Wang Kaibo, and Zhan Tao. Npuesb: A bpm environment based on enterprise service bus. In *Information Technology and Computer Science, 2009. ITCS 2009. International Conference on*, volume 2, pages 465–469, 2009.

[65] OASIS. Web Services Business Activity Version 1.1. http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec/wstx July 2007. [Online; accessed 02-February-2014].

[66] J. Oikarinen and D. Reed. RFC 1459 - IRC protocol specification. http://tools.ietf.org/html/rfc1459.html, May 1993. [Online; accessed 18-January-2014].

[67] Terence Parr. *The Definitive Antlr Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[68] Ponnekanti S. R. and Fox A. Interoperability among independently evolving web services. In *5th ACM/IFIP/USENIX international conference on Middleware, Toronto, Canada*, 2004.

[69] Arun N. Swami Rakesh Agrawal, Tomasz Imielinski. Mining association rules between sets of items in large databases. In *SIGMOD Conference 1993 : 207-216*, 1993.

[70] Tobias Rho, Gner Kniesel, and Malte Appeltauer. Fine-grained generic aspects. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL'06), AOSD 2006*, 2006.

[71] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[72] Peter Sewell. Modules, abstract types, and distributed versioning. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '01, pages 236–247, 2001.

[73] J. L. Sobral. Pluggable grid services. In *Proceedings of the 8th IEEE/ACM international Conference on Grid Computing (September 19 - 21, 2007). International Conference on Grid Computing. IEEE Computer Society, Washington, DC, 113-120*, 2007.

[74] E. Sonchaiwanich, J. Zhao, C. Dowin, and M. McRoberts. Using AOP to separate SOA security concerns from application implementation. In *Military Communications Conference, 2010 - MILCOM 2010*, pages 470 –474, 31 2010-nov. 3 2010.

[75] Willem van Heiningen, Tim Brecht, and Steve MacDonald. Exploiting dynamic proxies in middleware for distributed, parallel, and mobile java applications. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 231–231, Washington, DC, USA, 2006. IEEE Computer Society.

[76] S. Vinoski. The more things change... In *IEEE Internet Computing, IEEE Computer Society*, 2004.

[77] W. Vogels. Web services are not distributed objects. In *IEEE Internet Computing, IEEE Computer Society*, 2003.

[78] Gregor von Laszewski, Jarek Gawor, Peter Lane, Nell Rehn, Mike Russell, and Keith Jackson. Features of the java commodity grid kit. In *Concurrency and Computation: Practice and Experience 14:10451055*, 2002.

[79] R. Weinreich, T. Ziebermayr, and D. Draheim. A versioning model for enterprise services. In *AINAW'07, Ontario, Canada*, 2007.

[80] Qing Wu and Ying Li. Scudadl: An architecture description language for adaptive middleware in ubiquitous computing environments. In *Computing, Communication, Control, and Management, 2009. CCCM 2009. ISECS International Colloquium on*, volume 4, pages 611–614, 2009.

[81] Uwe Zdun. Pattern-based design of a service-oriented middleware for remote object federations. *ACM Trans. Internet Technol.*, 8(3):15:1–15:38, May 2008.

[82] Charles Zhang and Hans-Arno. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, AOSD '03, pages 130–139, New York, NY, USA, 2003. ACM.

Index