Journal of Information Processing Vol.23 No.1 41–57 (Jan. 2015)

[DOI: 10.2197/ipsjjip.23.41]

Regular Paper

A DSL for Contract-centric Compatibility Assessment in Distributed Services

Aurélio Akira Mello Matsui[1,a)] Hitoshi Aida[1,b)]

Received: April 8, 2014, Accepted: September 12, 2014

Abstract: As distributed computing becomes part of the daily life of an expressive number of people, it becomes important to rethink the way we express compatibility between the components of distributed systems. This paper proposes a mechanism to check service compatibility based on service 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 contracts. This way, we can formally check if they can reach a target state, meaning that 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 examination of behavior in a message-oriented middleware environment. We provide a qualitative evaluation of our proposal through the analysis of an example involving the dynamic creation of interconnections.

Keywords: Domain-specific Language, distributed computing, distributed service, service contract, π-calculus

1. Introduction

In a world where distributed systems are pervasive and each day more relevant, it is important to analyze how developers create clients and services. We are witnessing the appearance of a distributed service market in which developers do not have the luxury of fully understanding the internals of the remote services that they use as part of their applications. So trying to come out with service interfaces that prevent misuse by clients can be tricky. For instance, in cloud computing, a service is usually provided by a company for a fee, which is one of the key differences between computing clouds and computer grids, according to Ref. [15]. 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 and the service provider reputation.

To keep development costs low, client applications should be checked before a remote service is hired. Also, 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 client applications to formally verify if their software can successfully interact with a service specification rather than with a specific service implementation. We 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 [7]), so the actual software composition cannot be determined during coding.

Here we propose a method to verify agent implementations against contracts. We take into account interactions between objects and 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 [41], which we apply to be part of contracts.

But what does it mean for two loosely coupled agents to be compatible? It means that the assumptions under which they were both built will be met during run time. For instance, let P be an agent that was built to interact with an abstraction Q of a remote service. Let us also assume that the model implemented by P states that initially Q should offer a function m_1 and that whenever m_1 is called, another function m_2 becomes available. A contract of Q should, then, impose that Q should be either in a state in which only m_1 can be called or in a state in which only m_2 can be called. Also, the contract should impose that a call to m_1 should be responsible for the state transition. A contract may go as far as explaining what should go on inside of Q but we do not need (and do not encourage) this sort of details since, as we said, we base our model on the assumption that many implementations may be available and too detailed contracts leave no room for adaptation or creativity.

Our main contribution to the field is a Domain-Specific Language (DSL) that was specifically designed for developers to express agent interactions by means of either implicit channels (calls to remote methods), or explicit channels that can be passed from agent to agent the same way that object references are passed. The DSL also makes it possible for an automated process to extract from its source code an equivalent π-calculus expression that is later checked against the contract. Characteristics of the proposed DSL are based on the blueprint of a Message-Oriented Middleware (MOM), which we also outline.

The rest of this paper is organized as follows. The following section introduces related research. Section 3 introduces our model for service contracts and the outline of the MOM model

[1] University of Tokyo, Bunkyo, Tokyo 113–8656, Japan
[a)] akira@aida.t.u-tokyo.ac.jp
[b)] aida@ee.t.u-tokyo.ac.jp

that we use. Section 4 introduces a DSL we have created on top of the Java programming language to support advanced contract verification. Section 5 is an example of our proposal. In Section 6, we discuss our contributions. Finally, we conclude this paper on Section 7. We also provide two appendices at the end of this paper. Appendix A.1 defines a special π-calculus context that we use to check client implementations of contracts. Appendix A.2 describes the grammar of our DSL.

2. Related Research

Interoperability between old and new code based on types was proposed in Ref. [42]. It also uses π-calculus, but as a means to define a type system, instead of trying to check code against a contract.

Reference [30] proposes 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 also proposed in Ref. [10]. Verifications based on finite state machines allow us to identify illegal sequences 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, they do not allow us to easily represent unobservable transitions, and they do not cope with a variable number of states. The π-calculus includes all those missing features.

Service compatibility is an issue addressed both in Refs [5] and [6]. Both tackle 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 [*1]), 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).

Analyzing Web service compatibility using graphs and protocols was addressed in Refs. [4], [9], and [12]. A formalization of compatibility was also proposed in Ref. [18].

OurGrid [28], [29] also proposes using transformations over an Object-Oriented Programming (OOP) language to create distributed systems. The strategy of OurGrid is to allow programmers to mark certain Java threads as points to be exported for the grid to execute. The transformations are performed by AspectJ [1], an Aspect-Oriented Programming (AOP) language for Java. Aspects identify those explicit marks and replace calls to the execution of threads with procedures that will request the execution of the threads in a remote node. AOP is based on the idea that standard OOP design does not allow for correct mapping of crosscutting concerns. Aspects provide for a unique concretion for crosscutting concerns. AOP languages (even the ones that have more expressive point cut grammars such as LogicAJ [40]) limit themselves to manipulate procedures in interaction points between classes: method calls and method declarations. In general, field introductions provide support for additional logic. But the interactions between agents are not only subject to their method structure, but also the result of the structure of algorithms, which AOP ignores.

In contrast, in our design, we propose that we should extract behavioral patterns from a meticulous analysis of a method structure. In our model, direct interactions with each remote service should be restricted to a single client object, so we can predict how each service method will be called. Restricting interactions to those special client objects is what allowed us to check contract compatibility as we will see in the next section.

On Ref. [33] it is proposed that Web Service Business Activity (WS-BA) [34] termination protocols (coordinator initiated or participant initiated completion) should be applied to web services as a set of constraints that allows for formal verification of algorithms to ensure both services reach an acceptance state. This property is used as a compatibility criteria. To avoid deadlocks and race conditions, the paper proposes to use SOAP Service Description Language (SSDL) to express constraints. SSDL constraints are then translated into Process Meta Language (PROMELA) source code, which is in turn executed by the SPIN model checker [21] [*2]. Interaction between services should be grouped into activities that take place sequentially. Each activity should complete in a consistent state. Our compatibility criteria differs from Ref. [33] 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 representing entities that are equivalent to parallel activities.

In this paper, we claim that exposing service state can improve the client-service cooperation and the formal verification of compatibilities. This approach is similar to Design by Contract (DbC) [31], which is the foundation of some programming languages such as Eiffel [*3]. The Java Modeling Language (JML) [20], [26], [27], succeeding iContract [24], is perhaps the most widely used approach for DbC in Java. The idea behind JML (as is the approach of other DbC languages such as Eiffel [2]) is that contracts should be written as part of the source code. In our model, on the other hand, contracts are not specific to one particular source code or implementation. Another difficulty to adapt the DbC to our proposal would be to track interactions between agents in order to extract interaction patterns which, using JML can only take place during run time. We want to check contract adherence before run time. Another dimension in which the classic DbC is not enough for our needs is interactions by means of channels. We need the π-calculus and to use bisimulation in order to describe processes that exchange messages and that have a connection topology that is dynamic by passing channels as message data.

We can state that we also exploit DbC since our contracts can also express pre-conditions and post-conditions. We model channel interaction as method calls. So in our model, a pre-condition is the assumption that the caller will be able to receive results from the called peer. Post-conditions are assumptions that the caller has regarding the state that the called peer will have after interaction. This is expressed by the caller exposing output channels after interaction. Invariants, on the other hand, are not as

[*1] Although a zero state is formally defined in the π-calculus as a state in which further interactions are impossible, it is also reasonable to designate certain states as accepting ones, even if a further interaction is possible from them.

[*2] http://spinroot.com

[*3] http://www.eiffel.com/

straightforward. It is possible to specify asserts that enforce invariants, but they cannot be considered to be a syntactic feature of our model.

Architecture Description Languages (ADLs) are languages to provide specifications of distributed systems. Not only these languages provide a more convenient method to describe distributed systems, but they also produce system descriptions that can be subject to formal verification and simulation. Among the ADLs, at least the ScuADL [45] and the π-ADL [36] use the π-calculus principles to describe systems. Both provide a syntax to described systems as processes that communicate (or interact) over channels, and channels that can be passed as parameters.

The semantic properties of Java in terms of the π-calculus have been investigated in Ref. [22]. The equivalences between Java code and π-calculus that we use here are compatible with the ones in Ref. [22], but we study a special case of objects in a controlled environment and expressed using a DSL. We only consider those elements in the source code that are relevant to interactions between agents in the contract.

Agent mobility by means of process passing, instead of channel passing, is the core of the HOπ (Higher-order π-calculus) and has been established in Ref. [44]. A detailed account on the topic can be found in Ref. [11]. Here we opted to address agent mobility using simple channel passing instead of a more complex representation of object mobility. Calculi of higher orders are used in scenarios in which incomplete processes are passed. This is the case of dynamic and functional programming languages. But here we aim at judging the possibility of loosely coupled systems to successfully interact. Therefore, not only channel passing is enough for our needs (as we model visibility extrusion, which can be completely represented using channel transmission), but a higher-order calculus would make it hard for us to prove bisimulation which equates to behavioral equivalence.

We define a component model and focus on applying the π-calculus as the means for contract specification to describe behavior and mobility, rather than trying to make a contribution to the field of the calculus itself.

We are basing our middleware on the Java Message Service (JMS) message topics, in which agents subscribe to topics and messages are directed to all connected nodes. Node subscription can contain a filter that may be used by the server to emulate one-to-one messages. The Calculus of Broadcasting Systems (CBS) [13], [39] provides a formalization for systems based on one-to-many messages, which are the most natural transmission mode in JMS. Here we decided not to address one-to-many messages the way CBS does since we are basing our assumptions on an OOP programming model in which objects exchange messages through method calling. However, we use broadcast messages exchange in message topics for tasks such as service declaration for resource brokers [3]. Such details regarding the underlying MOM are beyond the scope of this paper.

3. A model for Service Contracts

In this section we describe our model for service contracts. A contract is an abstraction for process behavior expressed by means of channel interactions. While contracts are expressed using the π-calculus, the implementations are written using a Java-based DSL. To make such relation more clear, let us consider a contract K, and an agent implementation c. Also, let $[\![c]\!]$ be the translation of c into the π-calculus. What we want to achieve is a set of syntactic structures that, when used to encode c, will enable us to tell if $[\![c]\!]$ has a behavior that is compatible with K. We will see later a more formal definition of compatibility, but for now we can state that if K has the form $P_1 \mid P_2$, then for c to be compatible with K, $[\![c]\!]$ has to successfully replace either P_1 or P_2. In other words, c should be a concretion of one of the roles (or abstractions) defined by K.

Because introducing the π-calculus is beyond the scope of this paper, we refer the reader to Ref. [41] for a detailed account. The π-calculus grammar we use is as follows.

$$
\begin{aligned}
&P \coloneqq (P \mid P) \mid P + P \mid !P \mid new\{x_1, \cdots, x_n\}P \mid Seq \\
&Seq \coloneqq A_1.\cdots.A_n(.0 \mid .S)? \\
&A \coloneqq M(X_1, \cdots, X_n) \mid \overline{M}\langle X_1, \cdots, X_n\rangle \mid \tau \\
&M \coloneqq m \ : \ (T_1 \times \cdots \times T_n) \to T \\
&X \coloneqq x \ : \ T \\
&S \coloneqq stateName \text{ `=' } P \\
&T \coloneqq \text{unit} \mid \text{Int} \mid \text{Boolean} \mid \cdots \mid Class \\
&Class \coloneqq M_1, \cdots, M_n
\end{aligned}
\tag{1}
$$

The first vertical bar represents processes in parallel, while the rest of the vertical bars mark EBNF options. The plus sign represents mutually exclusive options. An exclamation point is a replication ($!P$ is equivalent to $P|!P$), and *new* identifies bound (restricted) variables. P, Seq, A, M, and T stand for "process," "sequence," "action," "method," and "type" respectively. X is a method argument. The question mark is the EBNF mark for optional terms (zero or one). A zero marks the end of a sequence or is itself a sequence, and is implicit when not present at the end of a sequence. The Greek letter τ is the silent action. S is a named state, which is defined by a state name and a process expression. When a state is reached, necessarily at the end of a sequence, the sequence becomes whatever is expressed as the state. T represents a type, which can be a unit (equivalent to a void in Java or C), a primitive, or a class. Methods m receive zero or more parameters and have one result type. Classes are a collection of methods and have no fields, in contrast with Ref. [11]. We model interactions between objects, not the internal structure of methods or algorithms, so there are no variables available. The only possible reference are those to objects and channels. We can think of our definition of classes as OOP interfaces with extra requirements that govern the interactions between instances of those interfaces. Instead of fields that determine object state, objects have named states, as we will see in the following section.

The following equation illustrates the notation we use in this paper for clarity:

$$
o!m(x).P \mid o!\overline{m}\langle y\rangle.Q \to \{^{x}/_{y}\}P \mid Q \tag{2}
$$

We use the notation $o!m$ to mean a channel m which is part of an array of channels grouped under the name or class o, which is used on Ref. [11] and is roughly equivalent to a standard polyadic calculus convention: $m!x_1, \ldots, m!x_n$ instead of $\vec{x} = (x_1, \ldots, x_n)$.

The notation $\{^{x}/_{y}\}P$ is an alpha conversion in which all names y in P are replaced by x. We are now ready to define agents and contracts.

Definition 1 – Agent role. An agent role C is a π-calculus class with at least one unbound channel. All its unbound channels are represented using $C!m$, so that every passing of C in a reaction implicitly passes references to all its unbound channels.

Definition 2 – Contract. A contract K is a set of m restricted channels x_m, n agents roles C_n, and p client agent roles $C_p^{(client)}$ having the form:

$$K = \text{new}\{x_1, \ldots, x_m\}\,(C_1 \mid \ldots \mid C_n \mid C_1^{(client)} \mid \ldots \mid C_p^{(client)}) \tag{3}$$

We assume that contracts are free from deadlocks, live locks, lack of possible mutual termination, and other defects. It is not the contracts that we are trying to check, but the candidates to materialize the contracts.

Each agent role C_n or $C_p^{(client)}$ should be implemented by a single class in an OOP language. But the contract should not necessarily specify the internal behavior of methods. Contracts can be purposefully unspecific to allow for service implementations to define their own internal strategies to realize the contract. The expression of an agent role can consist of many sub-processes communicating through private channels. There is then a compromise between two extremes. On the one hand, a too vague contract will prevent us from detecting implementations that have run time interaction issues. On the other hand, specifying roles in too many details may leave no room for improvements in algorithms.

3.1 Middleware Model

We specify that the actual communication between agents has the following characteristics:

- all agents communicate through message topics, which are logical domains identified by a name and in which all connected agents can exchange messages freely;
- each agent subscription to a message topic is subject to its own reading and writing restrictions;
- references to topics can be passed along messages;
- new message topics may be created and destroyed dynamically according to commands issued by agents with enough rights.

Such model provides the vocabulary for the translation of classes written using the DSL. Programmers do not write their code based on such a model, but on a more abstract object model. For instance, when a programmer writes a code that sends the reference to an agent, the equivalent process (in terms of the π-calculus) is one in which a new channel reference is passed through an output channel, whereas the procedure that is actually executed is a message being sent through a message topic. Such message contains a reference to the topic in which the remote object can be reached.

Another reason for adopting such a model is a practical one. We created a prototype implementation of our proposal using the ActiveMQ [*4] implementation of the JMS specification, which has all the characteristics above except for the last one. We have developed the last capability by using tools that are not part of the JMS specification. Instead of going into implementation details, we outline that we can realistically assume the behavior described above for an MOM.

Actual message delivery, which implies a translation from a topic name to the actual physical location of agents, is done by the underlying message delivery method. This allows us to change the actual location of agents without changing the way agents communicate provided that agents are insulated from effects caused by changes in environment. We accomplish that by allowing only channel name exchange, not actual objects, as proposed by Ref. [44].

3.2 Compatibility Criteria

We take a source code c representing a single OOP class and obtain from it a π-calculus equivalent process $C = [\![c]\!]$. Our translation process differs from Ref. [22] in that we only consider elements of c that either make direct references to other agents in the contract or that change the execution flow in which those interactions occur. Also, c is written using a DSL built on top of Java, which allows for a more expressive power and extractions of C that fit into our middleware model. The translation is described in details in the next subsection. Here we define compatibility with a contract.

We allow for the DSL source code to make reflections over the availability of methods. For this to be possible, the client implementation is put in a context Θ that differs from the one that the client contract expresses. A formal definition of Θ is too long for the body of this paper, so we left it for Appendix A.1. For now, we only need to know that $\Theta([\![c]\!], C_x)$ is the environment in which a class c is checked for compatibility with an agent role C_x using (strong or weak) simulation or bisimulation. Θ provides reflection channels for each method in C_x. Originally proposed to prove algorithm equivalence [32], bisimulation allows us to verify behavior equivalence in distributed systems.

Definition 3 – Compatibility with a client role. We consider a class c to be compatible with the client agent role $C_x^{(client)}$ in a contract K, if $\Theta([\![c]\!], C_x)$ **simulates** C_x.

Definition 4 – Compatibility with a non-client role. We consider a class c to be compatible with the non-client agent role C_x in a contract K, if $\Theta([\![c]\!], C_x)$ **bisimulates** C_x.

In other words, we have a compatibility if there is a binary relation R such that $\Theta([\![c]\!], C_x)RC_x$ is a simulation. A more forgiving definition of compatibility accepts all $\Theta([\![c]\!], C_x)$ that weakly simulates C_x. For bisimulation in clients, we have the following definition:

Definition 5 – Canonical implementation of client role. We say that c is a canonical implementation of $C_x^{(client)}$ if $\Theta([\![c]\!], C_x^{(client)})$ **strongly bisimulates** $C_x^{(client)}$.

In the definitions above we have used three kinds of relations: (unilateral) simulation, bisimulation, and strong simulation. Each kind of simulation has its own purpose.

[*4] http://activemq.apache.org/

For the client role, we should not impose a strong simulation to allow for clients to provide partial implementations of contracts. If, for instance, the client contract enables the client to call fifty different methods, we should not expect each client implementation to call all those methods. Conversely, we should not force the client to accept whatever response the server wants to send. That does not imply that clients should not have any responsibility. The immediate question that arises is how to ensure that a client implementation does not break the service. The answer is by imposing that mutual termination (reaching an acceptance state) should be possible.

We call a canonical implementation of a client one that uses all service capabilities exactly as specified. Such property is equivalent to saying that the client in the Θ context is a strong bisimulation. In other words, not only everything that the client can do is possible under the contract, but also everything that the contract stipulates is possible under the client implementation without any of the parts having to wait for the other one to transition through intermediate states not present in the contract. For instance, if the client contract is $m_1.m_2$, then we cannot call canonical an implementation with the form $m_1.\tau.m_2$.

For the non-client role, bisimulation is required since, as we stated, we want any client implementation to be compatible with any non-client implementation. Therefore, each non-client role implementation should be capable of expressing the complete behavior specified by the contract. Nevertheless, we do not impose a strong bisimulation to non-client roles to allow for a certain flexibility to non-client agents in which the agent has intermediate states not present in the contract. For instance, if the non-client agent contract is $\overline{m_1}.\overline{m_2}$, then $\overline{m_1}.\tau.\overline{m_2}$ is a valid implementation. As we will see in Section 4.3, it is possible for clients to check if a certain output channel (such as m_2 in our example) is available.

4. A Domain Specific Language for Process Calculus-based Contracts

Now that we have established our compatibility criteria, in this section we introduce how we translate source code into π-calculus expressions. This section will present the DSL in a descriptive manner while a formal definition of the syntax is explained on Appendix A.2.

We start by discussing the proposed programming model around which the DSL was built. **Figure 1** is a diagram of this programming model. The figure represents one client accessing one remote server. Business objects in the client side indirectly access resources Res.1 and Res.2 that are located in a remote host. Examples of such resources include a remote database or some special kind of sensor. In our proposed architecture each node is capable of hosting agents on the client and service layers. These two layers host objects written using our DSL and that are checked against contracts.

Objects in the client host 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 proxies to 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.

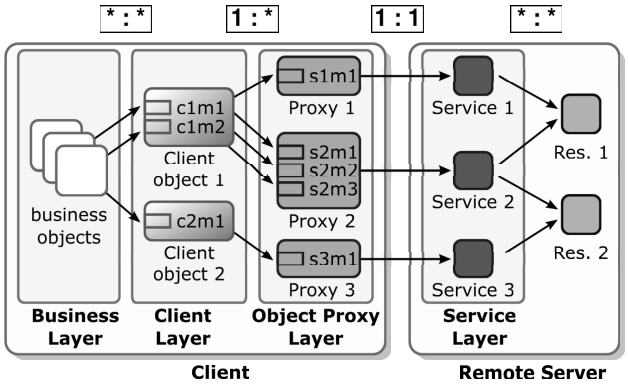

Fig. 1 Programming model for client and service.

Client objects are agents that implement a client agent role and are therefore checked against $C^{(client)}$ roles. Each client object provides one or more public methods (marked as "c1m1" etc in Fig. 1). These 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.

Proxy objects are an implementation of the remote proxy design pattern, which were also applied by Ref. [43]. 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 remote object.

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.

On the top of Fig. 1 there are database-like cardinality relations. When a client object is instantiated this object receives one of more service references through inversion of control (also known as "dependency injection"). 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 2 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 `to()` method sends the object to another state. During execution of the method, the object is in an undefined state until a `to()` is reached. This is because we equate states with π-calculus process definitions. For instance, if we have a process P that starts at a configuration $A = m_1.m_2.B$, in which B is a process definition, we call A and B named states that can be referred to in the source code and $m_2.B$ an unnamed intermediate state between A and B. If a method does not call `to()` as the last statement, or as

```
@InitialState(A)
clientclass Client1 {
    Service s;
    @State(A)
    public void x(String d) {
        s.m1(d); s.m2();
        System.out.println("Message to console");
        s.m3(); s.m2();
        to(B); // state transition to B
    }
    @State(B)
    @Scope(s)
    public void m4(String d) {
        // use data from service...
        to(C); // state transition to C
    }
}
```

Fig. 2 Example of client object class.

```
@InitialState(A)
clientclass Client2 {
    Service s1;
    Service s2;
    @State(A)
    @Scope(s1)
    public void m4(String d) {
        System.out.println("x");
        to(B);
    }
    @State(B)
    @Scope(s2)
    public void m6(String d) {
        s1.m3();
        to(C);
    }
}
```

Fig. 3 A client object class interacting with two services.

the statement right before a return statement, we assume that the method makes a transition to the state in which it started, having an unnamed state during its execution.

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.

From the perspective of the service `s`, the source code in Fig. 2 translates into:

$$\begin{aligned}&\text{new}\{r\}\ (s!\overline{m_1}\langle r\rangle.r).\text{new}\{r\}\ (s!\overline{m_2}\langle r\rangle.r).\\&\text{new}\{r\}\ (s!\overline{m_3}\langle r\rangle.r).\text{new}\{r\}\ (s!\overline{m_2}\langle r\rangle.r).s!m_4(r).\bar{r}\end{aligned} \tag{4}$$

Restricted channels r are callback channels that are called by the service to signal the end of a method call. Arguments passed to `m1` and `m4` are ignored since they represent data sent by value, not agents described in the contract. This concept will become more clear when we explain the translation process. The call to `System.out.println()` is ignored for a similar reason, 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.

Figure 3 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 $C_{C,i3} = \text{new}\{r\}\ (s_1!m_4(r).\bar{r}).\text{new}\{r\}\ (s_1!\overline{m_3}\langle r\rangle.r)$, while the interaction with `s2` is translated into $C_{C,i4} = \tau.\text{new}\{r\}\ (s_1!m_6(r).\bar{r})$.

$C_{C,i3}$ 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, $C_C = \text{new}\{r\}\ (s!m_6.\bar{r}.0)$, then $C_{C,i4}$ simulates C_C, but only weakly, since $C_{C,i4} \xrightarrow{s!m_6} 0$. In other words, $C_{C,i4} \stackrel{s!m_6}{\Longrightarrow} 0$.

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.

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, client requests would be not limited to client objects. Coming up with the equivalent π expression would be impossible, as we could predict which classes will end up having access to the service.

Opacity – Fields of client objects should not be directly accessible from other objects.

In the rest of this section, we present how control flow blocks and some DSL-specific features we have created are translated into π-calculus expressions. We will represent generic sections of code using processes `P`, `Q`, ... that can be translated to π-calculus.

4.1 Method Calls

Method calls are based on the standard encoding for objects in the π-calculus [44]. A method call consists of an output channel in which is passed a list of arguments $arg_1, arg_2, \ldots$ and a return channel r. We also allow for an extra channel e to receive exceptions or errors that occur during the method execution. The following equation shows the general expression of a non-blocking method call, where P and Q are processes whose execution is triggered by a method return and a method exception respectively. R is a process that starts running right after the call to the method.

$$\begin{aligned}&\text{new}\{r, e\}\ (obj!\overline{method}\langle arg_1, arg_2, \ldots, r, e\rangle.\\&(\llbracket R\rrbracket \mid r.\llbracket P\rrbracket \mid e.\llbracket Q\rrbracket))\end{aligned} \tag{5}$$

This strategy uses the Visitor design pattern [17], 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. A blocking version of method calls omits $\llbracket R\rrbracket$ from the equation above. Here we treat all method calls as blocking for simplicity. A non-blocking version requires the use of a specific API for this purpose, such as the Java Future inter-

face [*5].

4.2 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.

$$\text{new}\{\kappa_1, \kappa_2\}\ (\llbracket P \rrbracket.\overline{\kappa_1} \mid \kappa_1.\llbracket Q \rrbracket.\overline{\kappa_2} + \kappa_1.\llbracket R \rrbracket.\overline{\kappa_2} \mid \kappa_2.\llbracket S \rrbracket) \tag{6}$$

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. 4** 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 Eq. (6). We offer a better construct on the next subsection.

4.3 Contract-based If-blocks

Our DSL allows for contract entities to be used as if block conditions. For instance, consider the source code in **Fig. 5**.

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:

$$\begin{aligned}&\text{new}\{\kappa_2, \vartheta_T, \vartheta_F\}\ (\llbracket P \rrbracket.\overline{\vartheta_{m2}}\langle \vartheta_T, \vartheta_F \rangle \mid \\ &\vartheta_T.\llbracket Q \rrbracket.\overline{\kappa_2} + \vartheta_F.\llbracket R \rrbracket.\overline{\kappa_2} \mid \kappa_2.\llbracket S \rrbracket)\end{aligned} \tag{7}$$

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 A.1.

4.4 Generic Loop Blocks

Loops such as the one in **Fig. 6** are translated into more complex structures. Again, we need restrictions in order to create internal reactions.

```
public void x(boolean condition) {
    P
    if (condition) { Q } else { R }
    S
}
```

Fig. 4 Simple if-block.

```
P
if (s.m2 callable) { Q } else { R }
S
```

Fig. 5 Simple if-block.

[*5] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html

$$\begin{aligned}&\text{new}\{\kappa_{1S}, \kappa_{1E}, \kappa_{2S}, \kappa_{2E}\}\ (\llbracket P \rrbracket.\overline{\kappa_{1S}}.\kappa_{1E}.\llbracket R \rrbracket) \mid \\ &!(\kappa_{1S}.\overline{\kappa_{2S}} + \kappa_{1S}.\overline{\kappa_{1E}}) \mid !(\kappa_{2S}.\llbracket Q \rrbracket.\overline{\kappa_{1S}})\end{aligned} \tag{8}$$

Each of the three expressions in parallel has 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.

4.5 Java Threads

$$\text{new}\{\kappa_1\}\ (\llbracket P \rrbracket.\overline{\kappa_1}.\llbracket R \rrbracket) \mid (\kappa_1.\llbracket Q \rrbracket) \tag{9}$$

Modeling a Java thread as in **Fig. 7** using the π-calculus is trivial since parallel computation is central to the π-calculus.

A different approach is used in Ref. [11]. That model allows for methods to return before the termination of the processing, which breaks the synchronization that exists between caller and method. The synchronization break allows for the already parallel processes to become independent. Again, the Java's java.util.concurrent package provides the Future interface that can be used for methods to detach method return from method execution.

4.6 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. **Figure 8** 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).

The π-calculus expression that represents the source code in Fig. 8 is:

```
P
for (String a : collection) {
    Q
}
R
```

Fig. 6 Loop block.

```
P
new Thread() { public void run() {
    Q
}}.start();
R
```

Fig. 7 Java thread.

```
P
fork {
    Q
} {
    R
}
S
```

Fig. 8 A fork-join example.

```
class X {
    public synchronized(G1) void m1() { ... }
    public synchronized(G1) void m2() { ... }
    public synchronized(G2) void m3() { ... }
    public synchronized(G2) void m4() { ... }
    public void m5() { ... }
}
```

Fig. 9 Sets of synchronized methods.

$$\begin{aligned}&\text{new}\{\kappa_f, \kappa_j, \kappa_{1S}, \kappa_{1E}, \kappa_{2S}, \kappa_{2E}\}\ (\llbracket P \rrbracket.\overline{\kappa_f}.\kappa_j.\llbracket S \rrbracket)\ |\\ &(\kappa_f.\kappa_{1S}.\kappa_{2S} + \kappa_f.\kappa_{2S}.\kappa_{1S})\ |\\ &(\kappa_{1S}.\llbracket Q \rrbracket.\overline{\kappa_{1E}})\ |\ (\kappa_{2S}.\llbracket R \rrbracket.\overline{\kappa_{2E}})\ |\\ &(\kappa_{1E}.\kappa_{2E}.\overline{\kappa_j} + \kappa_{2E}.\kappa_{1E}.\overline{\kappa_j})\end{aligned} \tag{10}$$

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.

4.7 Monitor Object

The monitor object design pattern [25] aims 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. We translate a class c having synchronized methods $m_1, m_2, \ldots$ as:

$$\llbracket c \rrbracket = (\llbracket m_1 \rrbracket + \llbracket m_2 \rrbracket + \ldots).\llbracket c \rrbracket \tag{11}$$

Contracts 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 class would be:

$$\begin{aligned}\llbracket c \rrbracket &= C^{(1)} \mid C^{(2)} \mid C^{(3)}\\ C^{(1)} &= (\llbracket m_1 \rrbracket + \llbracket m_2 \rrbracket).C^{(1)}\\ C^{(2)} &= (\llbracket m_3 \rrbracket + \llbracket m_4 \rrbracket).C^{(2)}\\ C^{(3)} &= (\llbracket m_5 \rrbracket).C^{(3)}\end{aligned} \tag{12}$$

Figure 9 outlines a class that translates into such expression. `G1` and `G2` are names of two monitor objects that work independently. The method m_5 has no restriction, therefore it does not need to be marked as synchronized.

4.8 Agent Creation

We also need a way then to represent channel creation and channel passing along messages. Otherwise we would restrict our model to static channel structures. Passing a channel in the π-calculus means giving to the receiver the capability to interact with that channel. Controlling which agent has access to which

```
R newAgent = new R();
client1.receive(newAgent);
```

Fig. 10 Channel creation and passing.

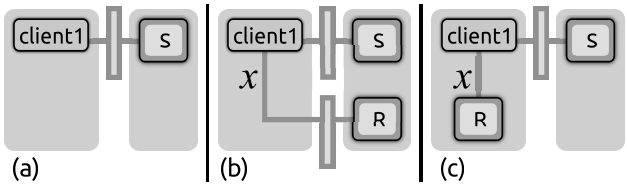

Fig. 11 Agent creation process. (a) Before the execution of the code in Fig. 10. (b) After the execution, the agent R was created with its own new private message topic. (c) A different result of execution in which R is mobile and migrates to the client.

channel is done by means of the contract. A contract may specify that a certain agent role A can receive a set of channels C. If the same contract does not describe any operation in which A sends C to another agent, then we can be sure that the access to C is limited to the boundaries of A.

In contrast with the simple channel passing in the π-calculus, we only allow object reference passing, which carries one or more channels. The reason is because a single channel passing can be easily modeled using an object with a single method, so restricting reference passing to objects is general enough. We decided to equate channel creation with object creation, which provides for a more consistent and concise programming model.

Figure 10 shows an example of agent creation and passing. Channel creation is not different from object creation except from the fact that such operation is present in the contract, so the instantiation is not ignored during translation. The main difference is that, behind the scenes, a new channel creates a new service and a new JMS message topic to reach such a service. **Figure 11** shows the actual agent creation and JMS topic creation that takes place. Vertical bars are JMS topics. Initially, `client1` has access to the service S. The execution of the code in Fig. 10 creates a new JMS topic x and a new agent R (Fig. 11 (b)). If the R class is marked with the `@Mobile` annotation, then R will reside in the client and no additional JMS topic is needed (Fig. 11 (c)).

New services need their own execution flow, independent from the process that created it. So an agent creation is translated into a new parallel process, similar to the creation of a new thread. Let S be a service that exposes a method m_1 unrelated to agent creation and a method m_2, as in Fig. 10, that creates a new agent. If m_1 and m_2 are synchronized, by applying a pattern similar to Eq. (11) we obtain the following translation of S:

$$\begin{aligned}&\llbracket S \rrbracket = \llbracket m_1 \rrbracket.\llbracket S \rrbracket +\\ &\text{new}\{x, \kappa_1\}\ (\overline{\kappa_1}.client_1!\overline{receive}\langle x, r\rangle.\bar{r}.\llbracket S \rrbracket \mid \kappa_1.\llbracket R \rrbracket\langle x\rangle)\end{aligned} \tag{13}$$

Where $\llbracket R \rrbracket\langle x\rangle$ is the translation of R in which x is used as the reference to reach R. We pass x to $client_1$ instead of passing $\llbracket R \rrbracket$ itself, which would be the HOπ method. We do not need the HOπ to represent mobility because we equate passing processes to passing object references, and object references contain all channels (all the free names of the process) necessary to interact with the agent. Also, the agent does not directly provide access to its fields, as we saw. So there is nothing in the object that can be of interest to another process and that would justify

using something more sophisticated than passing references.

So far, a new agent is a new object that shares a new channel with a client. An agent that exposes methods $m_1, m_2, \ldots$ will have the format Eq. (11). But we also need to give meaning to output channels (potential method calls that the agent makes) in the newly created agent. The answer is that the agent will be able to interact with (or informally, to see) the client that received x. In terms of the DSL, there will be dependency injection of the client in the agent. This mechanism is similar to a listener but without the burden to have to define a listener. We will see an example of such mechanism in the following section when a client asynchronously receives messages from a channel subscription agent.

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. Authentication and authorization are implemented by JMS. What is missing in the JMS Service Provider Interface (SPI) specification [*6] is user identity in messages, but the ActiveMQ API provides such a feature. Context can be modeled using contracts to specify what each agent can access.

5. Example of Compatibility Checking

In order to illustrate our proposal, let us analyze an example of an Internet Relay Chat (IRC) service [23], [35] implementation using our proposed method. In this protocol, each client connects to a single server. Each IRC network may have one or more servers connected in a spanning tree. Messages are sent from a client to an IRC chat room [*7] and may pass through a series of servers until it reaches its destination. At any given time, each client may be connected to zero or more chat rooms.

The actual IRC protocol has more features such as chat room operators, but here we will only implement joining chat rooms, sending messages, and receiving messages for simplicity. **Table 1** lists all functions provided by our partial implementation of the protocol. Agents $Server$, $Conn$, and Ch represent, respectively, the server that provides login and connections, a connection to the IRC system, and a chat room.

5.1 Service Contract

We define the initial server state using the following expression, with r and e having the usual semantics: return and error channels for the server to send messages back to the client.

Table 1 List of IRC agents and functions.

Agent	Function name	Short name	Data arguments
$Server$	login	login	user/password
$Conn$	join chat room	jc	chat room name
Ch	send message	sm	message
	receive message	rm	message
	disconnect	d	–

[*6] JMS is a facade specification. It simply consists of a set of interfaces that an Application Programming Interface (API) should implement, together with the specification on how those interface should behave. Each API that implements the JMS SPI may offer its own message delivery method as long as the facade behaves as expected.

[*7] The IRC specification calls each space in which clients exchange messages a "channel". We refrain from using this term in order to avoid ambiguities with π-calculus channels. We will always refer to the IRC abstraction as a "chat room".

$$
\begin{aligned}
&Server \coloneqq \text{new}\{x\}\ (Server'\langle x\rangle)\\
&Server' \coloneqq login(r, e).\\
&(\overline{r}\langle x\rangle.(Conn\langle x\rangle \mid Server'\langle x\rangle) + \overline{e}.Server'\langle x\rangle)
\end{aligned}
\tag{14}
$$

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 $Conn\langle x\rangle$ 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 executes $\overline{e}$, which informs the client about a failure. We define the $Conn$ process using:

$$
\begin{aligned}
Conn(ref) \coloneqq \text{new}\{x\}\ &ref!jc(r, e).(\overline{r}\langle x\rangle.(Ch\langle x\rangle \mid Conn\langle ref\rangle)\\
&+ \overline{e}.Conn\langle ref\rangle)
\end{aligned}
\tag{15}
$$

If the client is not allowed to connect to the chat room, the $Conn$ agent will reply by interacting with the output channel $\overline{e}$. If joining the chat room is successful, the $Conn$ agent will create a new process Ch that handles the subscription and will send back to the client a reference x to this new Ch. The contract of Ch is:

$$
\begin{aligned}
Ch(ref) \coloneqq\ &(ref!sm(r).\overline{r} + ref!\overline{rm}\langle r\rangle.r\\
&+ ref!d(r).\overline{r}).Ch\langle ref\rangle
\end{aligned}
\tag{16}
$$

Agents in this context are the expression of services from the point of view of clients. For instance, a subscription to a chat room is represented by a connection with a Ch agent. But several clients should be simultaneously connected to the same IRC chat room for them to communicate. Nevertheless, each client has its own instance of Ch with which it interacts according to the contract, in spite of all the complex processes that may be happening on the server.

5.2 Client Contract

The client contract is given by the following:

$$
\begin{aligned}
&C^{(s)} \coloneqq \text{new}\{r, e\}\ (\overline{login}\langle r, e\rangle.(r(c).C^{(c)}\langle c\rangle + e.0))\\
&C^{(c)}(c) \coloneqq c!\overline{jc}\langle r, e\rangle.(r(ch).(C^{(t)}\langle ch\rangle \mid C^{(c)}\langle c\rangle)\\
&\qquad + e.C^{(c)}\langle c\rangle) + 0\\
&C^{(t)}(ch) \coloneqq ch!\overline{sm}\langle r\rangle.r.C^{(t)}\langle ch\rangle + ch!rm(r).\overline{r}.C^{(t)}\langle ch\rangle\\
&\qquad + ch!\overline{d}\langle r\rangle.r.0
\end{aligned}
\tag{17}
$$

The superscripts (s), (c), and (t) stand for "start," "connected," and "talking," respectively. At the $C^{(s)}$ state, the client sends channels r and e to the server together with a user name and a password that are only present in the Java portion of the contract. After receiving such a login request, the server is responsible for checking the data provided by the client (user name and password) and call either r or e in case of a 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 $c!\overline{jc}$ or die, going to the zero state. Calls to the e channel cause the client to simply terminate, which here is represented by 0, a zero [*8].

[*8] Another way commonly found in the literature to represent termination is using the word "stop" instead of 0 (zero).

At the $C^{(c)}$ state, the client has a connection with the server and can use such a connection to request to join a chat room. If joining is successful, the server will respond by interacting with r and passing x, a reference to the newly created chat room agent. The client should then create a new process $C^{(t)}$ that will interact with the chat room.

If, on the other hand, joining is not allowed [*9], the connection will respond by calling the e channel instead. When this happens, the client remains at the connected state, in which it can try again to join a chat room.

We did not model termination of any sort for simplicity. In an IRC network the service is interactive and therefore dependent on the end user wish 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 [34] should take place.

It is also important to emphasize that, after a successful connection, a $C^{(c)}$ process is always available in the client. Therefore there is no limit on the number of simultaneous chat room 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.

5.3 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 12** shows the source code of an attempt to implement the client. This client keeps a hash map containing all subscribed chat rooms. When some business class calls the `sendMessage` method, the procedure checks if the required chat room is already in the hash map. If it is, then the method uses it, if it is not, then the method attempts to retrieve a new chat room object.

Figure 13 is the `ChatRoomWrapper` class, which has two main roles. The first is to provide a callback method `receive`, which is equivalent to an input channel in the client side, using π-calculus terms. The second role is to provide a concretion of $C^{(t)}$. The contract was designed such that $C^{(t)}$ becomes independent from its creator $C^{(c)}$. The independence between processes is represented by a parallel block.

Note that the client never calls the `disconnect` method, which is part of the contract. This would clearly violate a contract if we apply a strict criterion of bisimulation or of necessary mutual termination, but on this example we will instead apply a forgiving definition of compatibility. The π-calculus expression that is equivalent to the source code in Fig. 12 is as follows:

$$\begin{aligned} A &:= \overline{login}\langle r, e\rangle.(r(conn).B\langle conn\rangle + e.Z) \\ B(c) &:= c!\overline{jc}\langle r, e\rangle.(r(ch).(CW\langle ch\rangle \mid B\langle c\rangle) + e.B\langle c\rangle) \\ CW(ch) &:= (ch!\overline{sm}\langle r\rangle.r + ch!rm(r).\overline{r}).CW\langle ch\rangle \\ Z &:= 0 \end{aligned} \tag{18}$$

The following relation R provides the proof of simulation:

[*9] In the IRC protocol, a subscription request may be denied for private chat rooms. Although we do not implement private chat rooms, we have decided to include connection refusals here to illustrate situations in which clients need to deal with the server rejecting a request.

```
import java.util.HashMap;
@InitialState(A)
public class IRCClient {
  private IRCServer server;
  private IRCConnection conn;
  private HashMap<String, ChatRoomWrapper> rooms =
  new HashMap<String, ChatRoomWrapper>();
  @State(A)
  public void login(String username, String password) {
    try {
      conn = server.login(username, password);
      to(B);
    } catch(RemoteException e) {
      // Prevents any further calls to this object
      to(Z);
    }
  }
  @State(B)
  public void sendMessage(String roomName,
                          String message) {
    ChatRoomWrapper crw;
    if (rooms.containsKey(roomName)) {
      crw = new ChatRoomWrapper(rooms.get(roomName));
    } else {
      try {
        crw = new ChatRoomWrapper(conn.join(roomName));
        rooms.put(roomName, crw);
      } catch(RemoteException e) { return; }
    }
    crw.send(message);
}}
```

Fig. 12 IRC client.

```
public class ChatRoomWrapper
implements IRCMessageListener {
    private IRCChatRoom room;
    public ChatRoomWrapper(IRCChatRoom room) {
        this.room = room;
        room.setListener(this);
    }
    public void send(String message) {
        room.send(message);
    }
    @Scope(room)
    public void receive(IRCMessage message) {
        System.out.println(message);
}}
```

Fig. 13 The IRC chat room wrapper class.

$$\begin{aligned} &R = \{(A, C^{(s)}), \\ &(r(conn).B\langle conn\rangle + e.Z, r(c).C^{(c)}\langle c\rangle + e.0), \\ &(B\langle conn\rangle, C^{(c)}\langle c\rangle), (Z, 0), \\ &(r(ch).(CW\langle ch\rangle \mid B\langle c\rangle) + e.B\langle c\rangle, \\ &r(ch).(C^{(t)}\langle ch\rangle \mid C^{(c)}\langle c\rangle) + e.C^{(t)}\langle c\rangle), \\ &(CW\langle ch\rangle \mid B\langle c\rangle, C^{(t)}\langle ch\rangle \mid C^{(c)}\langle c\rangle), \\ &(r.CW\langle ch\rangle, r.C^{(t)}\langle ch\rangle), (\overline{r}.CW\langle ch\rangle, \overline{r}.C^{(t)}\langle ch\rangle)\} \end{aligned} \tag{19}$$

R does not present all reachable states since each call to $c!\overline{jc}$ adds an extra process $C^{(t)}\langle ch\rangle$ to the system. Therefore, formally the number of states can grow indefinitely. Let R^* be a pattern of relations obtained from R in which each $CW\langle ch\rangle$ is re-

placed by n copies in parallel ($CW\langle ch_1\rangle \mid \ldots \mid CW\langle ch_n\rangle$) and in which each $C^{(t)}\langle c\rangle$ is also replaced by n copies in parallel ($C^{(t)}\langle c_1\rangle \mid \ldots \mid C^{(t)}\langle c_n\rangle$). It can be shown that such R^* is a simulation, since it proves that the client implementation in Eq. (18) simulates any state that can be reached by the client contract.

Note that the inverse relation R^{-1} is not a simulation. Take the pair $(X, Y) = (CW\langle ch\rangle \mid B\langle c\rangle, C^{(t)}\langle ch\rangle \mid C^{(c)}\langle c\rangle)$. It is easy to see that the inverse, (Y, X), cannot be in a simulation relation since $Y \xrightarrow{ch!\overline{d}} r.C^{(t)}\langle ch\rangle$, but no transition is possible from X by means of $ch!\overline{d}$. Therefore no relation that contains (Y, X) is a simulation. We already expected that R^{-1} would not be a simulation since the client does not use the `disconnect` method.

5.4 First Service Implementation: Agents as Servers

We show two possible implementations of the service to illustrate the usage of our DSL. We will not discuss compatibility verification on these implementations since this topic was already covered on the previous subsection. Our first service implementation, depicted in **Fig. 14** (a), is one in which agent topology mimics the topology of the IRC network. Each rectangle (t1, t2, t3, t4, t5, and IRCBus) represents one JMS 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 IRCBus that represents the message domain in which services share information regarding client topology and propagates messages. For instance, a message from C2 to an IRC chat room to which C5 is connected would pass through t1, S1, IRCBus, S3, and t5 until reaching C5. In this solution, IRC chat rooms are logical entities that have no parallel with JMS topics.

While Fig. 14 depicts the logical way in which agents are interconnected, **Fig. 15** is an example of an actual JMS deployment, which can be used to implement both solutions (a) and (b). Nodes M1, M2, and M3 are JMS messaging processes. All other agents are directly connected to M1, M2, or M3 through sockets. The protocol used for agents and messaging processes to communicate depends on the JMS implementation in use, as the JMS specifies an SPI rather than a message exchange protocol.

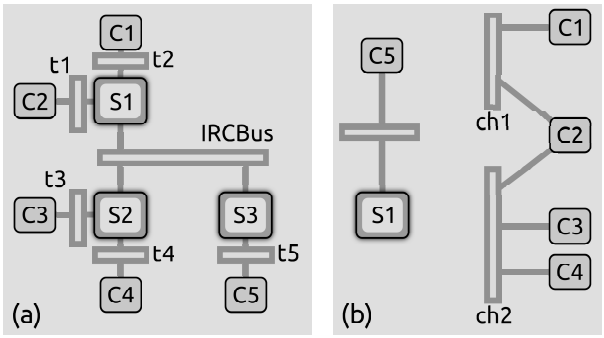

Fig. 14 The two implementations of a chat system. (a) A solution in which services reproduce IRC servers. (b) A solution in which communication channels reproduce IRC chat rooms.

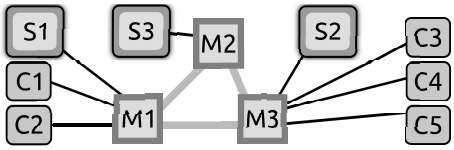

Fig. 15 Example of actual topology of process connections.

Figure 16 shows the outline of the source code for the *Server* agent, which is a trivial implementation. **Figure 17** is the connection class, which uses a local database of chat rooms called `roomDB`. The most important piece of this implementation is the `IRCChatRoom` class, depicted in **Fig. 18**. All objects from this class use the same JMS message topic IRCBus, as depicted in

```
public class IRCServer {
  public IRCConnection login(String username,
    String password) throws LoginException {
    if ( /* check access */) {
      return new IRCConnection(username);
    } else {
      throw LoginException();
}}}
```

Fig. 16 IRC login server.

```
public class IRCConnection {
  // (...) Logic related to roomDB, etc.
  public IRCChatRoom join(String name) {
    if (roomDB.contains(name)) {
      IRCChatRoom room  = roomDB.get(name);
      return room;
    } else {
      IRCChatRoom room = new IRCChatRoom(name);
      roomDB.put(name, room);
      return room;
}}}
```

Fig. 17 IRC connection class.

```
@InitialState(A)
@Service
class IRCChatRoom {
  private Listener listener;
  private String name;
  private MessageBus bus;
  public IRCChatRoom(String name) {
    this.name = name;
    this.bus = MessageBus.getInstance("IRCBus");
  }
  @State(A)
  public void registerListener(Listener listener) {
    this.listener = listener; to(B);
  }
  @State(B)
  public void sendMessage(String text) {
    bus.send(new IRCMessage(name, text));
  }
  @State(B)
  @Scope(bus)
  public void receive(IRCMessage message) {
    if (message.roomName().equals(name)) {
      listener.deliver(message.getText());
    }
  }
  @State(B)
  public void disconnect() {
    to(Z);
}}
```

Fig. 18 IRC chat room class.

```
@InitialState(A)
@Service
@Mobile
class ImprovedChatRoom extends IRCChatRoom {
  public ImprovedChatRoom(String name) {
    super(name);
    this.bus = MessageBus.getInstance(name);
  }
  @State(B)
  @Scope(bus)
  public void receive(IRCMessage message) {
      listener.deliver(ircMessage.getText());
  }
}
```

Fig. 19 An improved IRC chat room class.

Fig. 14 (a). The `MessageBus` class is part of the API available for the DSL and allows for nodes to get access to message topics. The `receive` method needs to check each message received to filter only those that are sent to its chat room.

5.5 Second Service Implementation: Message Domains as Chat Rooms

On this second service implementation, depicted in Fig. 14 (b), each IRC chat room 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. In Fig. 14 (b), ch1 and ch2 are chat rooms and we need service the process S1 only to provide connections that creates and delivers IRC chat room objects as mobile agents. After mobile agent creation, the agents are autonomous and can exchange messages without server intervention.

Figure 19 shows a subclass of `IRCChatRoom`. Note the `@Mobile` annotation, which allows objects of this class to be hosted on the client, as illustrated in Fig. 11 (c). The `IRCServer` and `IRCConnection` classes are the same of Fig. 16 and Fig. 17.

6. Discussion

Related research, as Refs. [28] and [29], 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 cannot expect, in the general case, to be able to extract a behavioral pattern from client class structure. For instance, consider a client contract having the form $C = (m_1.m_2).C$. Let us assume that the client layer in Fig. 1 does not exist and that business objects access the object proxy layer directly. In this new architecture, interaction with proxy objects can be done by more than one object. Let's say that there are two such business objects B_1 and B_2 that can call m_1 or m_2 according to data from sensors or any external factor. There is no way to determine if such a system complies with C since there is no guarantee over the order in which m_1 and m_2 are called. That is the reason why we need the client layer illustrated in Fig. 1, in which each proxy object is accessed by a single client object (therefore the one-to-many cardinality).

To the best of our knowledge, this research is the first to propose a DSL specifically to enable formal verification of contracts expressed using the π-calculus. This allows for our contracts to represent channel mobility, which we equated to object reference mobility. Also, basing the contract on π-calculus enables us, by means of a weak simulation, to define partial compatibility for clients while preserving behavior checking. We preserve the class as a first citizen principle of the Java language, but passing a single channel can be accomplished by passing a class with a single method.

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 the client layer make 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 validate them formally before actually implementing a distributed system.

Another contribution we made was on the π-calculus-Java model translation. The π-calculus 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 way for active processes to evaluate the possibility of interaction. The Θ context offers such a feature.

The π-ADL [36] allows for the declaration of components, connections between components, and composites. It is possible to define portions of the system that are subject of dynamic changes during execution. Such changes are represented as π-calculus processes that can be introduced in a certain environment. π-ADL has many tools to formally check described architectures, generate Java code from models, among other features. π-ADL defines a complex type system with primitives such as architec-

ture, port, connection, and protocol. Besides, it uses HOπ meaning expressions passed can be incomplete processes to be filled by the receiver. The aim is that π-ADL is used to describe boundary conditions in which elements of a connector type are responsible for data exchange.

Our proposal and the π-ADL are competing in that they both allow for π-calculus expressions to describe a distributed system architecture, but there are important differences. Our model of a system is much more abstract. We do not define connectors or boundaries but only π-calculus expressions that represent an outline for OOP object behaviors. All interactions are translated into method calls. Using our approach it is possible to invent classes that play the role of connectors or value objects [16] that can be transferred as messages from node to node, among other design patterns. But our language does not provide such primitives, or a vocabulary that delivers ready to use building blocks for a distributed application. What we propose is that we should take a contract in π-calculus and compare it with a concrete implementation written by a programmer in a variation of Java. The environment that we envisage is one in which the programmer should think in terms of an OOP, not in terms of a new paradigm in which he or she needs to explicitly deal with messages or channels.

We aim at creating a model to make it easier to check contract adherence so that we can have a large number of agents that can communicate. For instance, in a cloud computing environment, there could exist several client implementations $C_1, C_2, \ldots$ of the same client contract C and several server implementations $S_1, S_2, \ldots$ of the same service contract S. We want to ensure that any combination of client and service will be able to interact successfully.

Finally, perhaps the most important difference with ADLs is that we do not define a concrete architecture. As we focus on interactions (which is the emphasis given by the π-calculus to communication), agents may have any internal structure as long as the perceived behavior is the one expressed in the contract. For instance, Fig. 14 shows two implementations of server contracts behaving the same way from the perspective of clients even though their internal organizations are very different.

Our contracts aim at describing behavior rather than providing legal conditions of data to avoid defensive programming, as in the DbC paradigm. For example, our contracts are able to specify that a certain agent is able to spawn more agents and pass a reference to such newly-created agents over the network, as the IRC connection in our example does. After those references are made available to remote nodes, they allow for new interactions to take place.

Our proposal also differs from DbC in that our contracts define interactions between agents, which is a popular approach in distributed computing. 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" can also be deemed as a relation that ensures that the behavior of a process P' remains confined within 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.

Although our proposal is one in which verification is performed in a way that is similar to DbC, in our case we do not check constraints that are imposed by the programmer. On the other hand, DbC languages such as JML have the potential to be used with success to improve our model.

It was proposed that the π-calculus could be extended to represent objects [11], [44], but in our model we treat objects as a set of channels that move together and that are related to the same process. An object for us is then a process with input and output channels. There are no fields or algorithms in contracts since behavior is only expressed in the interactions between processes. We purposely wanted to restrict expressiveness of objects by allowing only methods to be declared. Not only this makes it possible for objects to be moved without the special treatment necessary that is given by HOπ and objects as defined in Refs. [8], [11], [41], but it also gives developers some freedom over contract implementation. Such flexibility is at the core of our proposal, as we already justified.

A point we still did not discuss was how to represent our contracts. Instead of presenting a possible actual representation, we rather discuss the points that make such representation difficult. The problem we face is that the π-calculus represents processing by means of matching channels. Whenever there is a pair of matching input and output channels, they have the chance to interact and such interaction is what provides change to the processes which is what processing means in the π-calculus. Actual process locations is a concept purposefully made abstract in the π-calculus definition, so it is the interaction, not the distribution that is primarily modeled.

OOP processing, on the other hand, is all based on methods. Those two ways to define computation do not perfectly match. The main difference is that for the π-calculus there are two kinds of channels: an output one, through which data is sent, and an input one that receives data. But Java has only one kind of method: one that is called, which we equate to an input channel since methods receive arguments.

Usually, such inversion of processing initiative is modeled in OOP using listeners. An object A becomes ready to have its methods called by a remote object B when A complies with a listener interface. But such an interface is in fact a new type that we want to avoid having. Our solution is to use the `@Scope` annotation.

A fair point can be made that the π-calculus presents itself as a contrived programming language for developers who are accustomed to writing imperative code. Our idea is that developers should not write π-calculus expressions but see the results of automated checking of their codes against the π-calculus expressions that a service designer wrote.

7. 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 source code that has no reflection over the structure of the contract could not be analyzed the same way. Our main contribution is an architecture to make these analyses possible, and a middleware model based on the JMS as a proof of concept. 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.

Acknowledgments This research was made possible through the financing provided by the Ministry of Education, Culture, Sports, Science and Technology (MEXT).

References

[1] AspectJ, available from ⟨http://www.eclipse.org/aspectj/⟩ (accessed 2014-02).
[2] Eiffel (2013), available from ⟨http://www.eiffel.com/⟩ (accessed 2014-02).
[3] Afgan, E.: Role of the resource broker in the grid. *ACM-SE 42: Proc. 42nd Annual Southeast Regional Conference*, pp.299–300, New York, NY, USA, ACM (2004).
[4] Benatallah, B., Casati, F. and Toumani, F.: Representing, analysing and managing Web service protocols, *Data & Knowledge Engineering*, Vol.58, No.3, pp.327–357 (Sep. 2006).
[5] Berardi, D., Calvanese, D., De Giacomo, G., Lenzerini, M. and Mecella, M.: Automatic composition of E-services that export their behavior, Orlowska, M.E., Weerawarana, S., Papazoglou, M.P. and Yang, J. (Eds), *Service-Oriented Computing–ICSOC 2003*, *Lecture Notes in Computer Science*, Vol.2910, pp.43–58, Springer Berlin / Heidelberg (2003).
[6] Bordeaux, L., Salaün, G., Berardi, D. and Mecella, M.: When are two web services compatible? Shan, M.-C., Dayal, U. and Hsu, M. (Eds), *Technologies for E-Services*, *Lecture Notes in Computer Science*, Vol.3324, pp.15–28, Springer Berlin / Heidelberg (2005).
[7] Buyya, R., Abramson, D. and Venugopal, S.: The grid economy (2005).
[8] Caromel, D., Henrio, L. and Cardelli, L.: *A theory of distributed object: Asynchrony - mobility - groups - components*, Springer (2005).
[9] Cherchago, A. and Heckel, R.: Specification matching of web services using conditional graph transformation rules, Ehrig, H., Engels, G., Parisi-Presicce, F. and Rozenberg, G. (Eds.), *Graph Transformations*, *Lecture Notes in Computer Science*, Vol.3256, pp.259–262, Springer Berlin / Heidelberg (2004).
[10] Cohen, D. and Fredman, M.: Products of finite state machines with full coverage, Lingas, A., Karlsson, R. and Carlsson, S. (Eds.), *Automata, Languages and Programming*, *Lecture Notes in Computer Science*, Vol.700, pp.469–477, Springer Berlin / Heidelberg (1993).
[11] Sangiorgi, D. and Walker, D.: *The π-calculus: A theory of mobile processes*, Cambridge University Press (2001).
[12] Dumas, M., Benatallah, B. and Motahari Nezhad, H.R.: Web service protocols: Compatibility and adaptation, *IEEE Data Eng. Bull*, pp.40–44 (2008).
[13] Ene, C. and Muntean, T.: A broadcast-based calculus for communicating systems, *Proc. 15th International Parallel & Distributed Processing Symposium, IPDPS '01*, pp.149–158, Washington, DC, USA, IEEE Computer Society (2001).
[14] Parr, T. et al.: Java Grammar in Antlr format, available from ⟨https://github.com/antlr/antlr4/blob/master/tool/test/org/antlr/v4/test/Java-LR.g4⟩ (accessed 2014-02).
[15] Foster, I., Zhao, Y., Raicu, I. and Lu, S.: Cloud computing and grid computing 360-degree compared, *Grid Computing Environments Workshop, GCE '08*, pp.1–10 (Nov. 2008).
[16] Fowler, M.: *Patterns of Enterprise Application Architecture*, A Martin Fowler signature book. Addison-Wesley (2003).
[17] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design patterns: Elements of reusable object-oriented software, Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc. (1995).
[18] Gong, X., Liu, J., Zhang, M. and Hu, J.: Formal analysis of services compatibility, Vol.2, pp.243–248 (July 2009).
[19] Gosling, J., Joy, B., Steele, G., Bracha, G. and Buckley, A.: Java 7 Language Specification, available from ⟨http://docs.oracle.com/javase/specs/jls/se7/html/index.html⟩ (accessed 2014-02).
[20] Hatcliff, J., Leavens, G.T., Rustan, K., Leino, M., Müller, P. and Parkinson, M.: Behavioral interface specification languages, *ACM Comput. Surv.*, Vol.44, No.3, pp.16:1–16:58 (June 2012).
[21] Holzmann, G.J.: *The SPIN Model Checker: Primer and Reference Manual*, Addison-Wesley Professional, 1 edition (Sep. 2003).
[22] Jacobs, B. and Piessens, F.: A *pi*-calculus semantics of java: The full definition, CW Reports CW355, Department of Computer Science, K.U. Leuven, Leuven, Belgium (Jan. 2003).
[23] Kalt, C.: RFC 2810 - IRC architecture description (Apr. 2000), available from ⟨http://tools.ietf.org/html/rfc2810.html⟩ (accessed 2014-02).
[24] Kramer, R.: iContract - The Java Design by Contract Tool, *Proc. Technology of Object-Oriented Languages and Systems, TOOLS '98*, pp.295–307, Washington, DC, USA, IEEE Computer Society (1998).
[25] Lavender, R.G. and Schmidt, D.C.: Active object – An object behavioral pattern for concurrent programming, Pattern Languages of Program Design 2, pp.483–499, Addisou-Wesley (1996).
[26] Leavens, G.T.: Tutorial on JML, the java modeling language, *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering, ASE '07*, p.573, New York, NY, USA, ACM (2007).
[27] Leavens, G.T. and Cheon, Y.: Design by contract with JML (2006), available from ⟨http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf⟩.
[28] Maia, M.E.F., Maia, P.H.M., Mendonca, N.C. and Andrade, R.M.C.: An aspect-oriented programming model for bag-of-tasks grid applications, *7th IEEE International Symposium on Cluster Computing and the Grid* (*CCGrid'07*) (2007).
[29] Maia, P.H.M., Mendonca, N.C., Furtado, V., Cirne, W. and Saikoski, K.: A process for separation of crosscutting grid concerns, *Proc. 2006 ACM Symposium on Applied Computing SAC '06*, pp.1569–1574, ACM, New York, NY (2006).
[30] Matsui, A.A.M. and Aida, H.: Advanced client-service compatibility assessment via analysis of references to service-side FSMs, *2011 IEEE 3rd International Conference on Communication Software and Networks* (*ICCSN*), pp.69–77 (2011).
[31] Meyer, B.: Applying 'design by contract', *Computer*, Vol.25, No.10, pp.40–51 (1992).
[32] Milner, R.: An algebraic definition of simulation between programs, Technical report, Stanford, CA, USA (1971).
[33] Nepal, S., Zic, J. and Chau, T.: Compatibility of service contracts in service-oriented applications. *IEEE International Conference on Services Computing*, pp.28–35 (2006).
[34] OASIS: Web Services Business Activity Version 1.1 (July 2007), available from ⟨http://docs.oasis-open.org/ws-tx/wstx-wsba-1.1-spec/wstx-wsba-1.1-spec.html⟩ (accessed 2014-02).
[35] Oikarinen, J. and Reed, D.: RFC 1459 - IRC protocol specification (May 1993), available from ⟨http://tools.ietf.org/html/rfc1459.html⟩ (accessed 2014-02)
[36] Oquendo, F.: π-ADL: An Architecture Description Language Based on the Higher-order Typed π-calculus for Specifying Dynamic and Mobile Software Architectures, *SIGSOFT Softw. Eng. Notes*, Vol.29, No.3, pp.1–14 (May 2004).
[37] Parr, T.: *The Definitive Antlr Reference: Building Domain-Specific Languages*, Pragmatic Bookshelf (2007).
[38] Parr, T.: *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*, Pragmatic Bookshelf, 1st edition (2009).
[39] Prasad, K.V.S.: A calculus of broadcasting systems, *Science of Computer Programming*, Vol.25, No.2-3, pp.285–327 (1995), *Selected Papers of ESOP'94, the 5th European Symposium on Programming* (1995).
[40] Rho, T., Kniesel, G. and Appeltauer, M.: Fine-grained generic aspects, *Workshop on Foundations of Aspect-Oriented Languages* (*FOAL'06*), *AOSD 2006* (2006).
[41] Milner, R.: *Communicating and mobile systems: The π-calculus*, Cambridge University Press (1999).
[42] Sewell, P.: Modules, abstract types, and distributed versioning, *Proc. 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '01*, pp.236–247 (2001).
[43] van Heiningen, W., Brecht, T. and MacDonald, S.: Exploiting dynamic proxies in middleware for distributed, parallel, and mobile java applications, *Proc. 20th International Conference on Parallel and Distributed Processing, IPDPS'06*, pp.231–231, Washington, DC, USA, IEEE Computer Society (2006).
[44] Walker, D.: Objects in the π-calculus, *Information and Computation*, Vol.116, No.2, pp.253–271 (1995).
[45] Wu, Q. and Li, Y.: Scudadl: An architecture description language for adaptive middleware in ubiquitous computing environments, *ISECS International Colloquium on Computing, Communication, Control, and Management, CCCM 2009*, Vol.4, pp.611–614 (2009).

Appendix

A.1 A Formalization of the Equivalent Expression to Verify Simulations: Θ

In this appendix we formally define Θ. We had to define such environment to extend the semantics of π-calculus and add reflection over channel availability, which can be used in algorithms. Without Θ, our definitions of compatibility would have been much more verbose. Also, Θ specifically models a characteristic of our proposed middleware, which allows for querying agent states.

This appendix also aims at supporting the claim that Θ is an application of π-calculus, rather than a proper extension of it. To keep track of available methods, we need counters, which we will implement using lists. The π-calculus defines very primitive building blocks, so describing an algorithm using it requires a long expression, which we will define by parts.

First we define the linking operator that connects two processes P and Q, which is based on the one defined in Ref. [41], Section 4.4. Let $\mathrm{fn}(P)$ be the set of free names in P. Given two processes P and Q in which $left, right \in \mathrm{fn}(P), \mathrm{fn}(Q)$, we define the linking operator $\frown$ as:

$$P \frown Q \coloneqq \mathrm{new}\{x\}\,(\{^{x}/_{right}\}P \mid \{^{x}/_{left}\}Q) \tag{A.1}$$

Channels *right* and $left$ in P and Q respectively are "connected" by making them point to the same concrete channel, and made private to these two processes. 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 Ref. [41], Section 7.5.

$$\begin{aligned}
&Empty(c,i,d) \coloneqq i.(Cell(c,i,d) \frown Empty)\\
&+c(\vartheta_T,\vartheta_F).\overline{\vartheta_F}.Empty(c,i,d)\\
&Empty \coloneqq left(c,i,d).Empty(c,i,d)\\
&Cell(c,i,d) \coloneqq i.(Cell(c,i,d) \frown Cell)\\
&+d.\overline{right}\langle c,i,d\rangle.0 + c(\vartheta_T,\vartheta_F).\overline{\vartheta_T}.Cell(c,i,d)\\
&Cell \coloneqq left(c,i,d).Cell(c,i,d)
\end{aligned} \tag{A.2}$$

$Empty(c,i,d)$ 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:

$$\begin{aligned}
\varphi_x^{(0)} &= Empty_x(\vartheta_x,\sigma_{x,INC},\sigma_{x,DEC})\\
\varphi_x^{(n)} &= \underbrace{Cell(\vartheta_x,\sigma_{x,INC},\sigma_{x,DEC}) \frown Cell_x \frown \ldots}_{n\ times} \frown Empty_x
\end{aligned} \tag{A.3}$$

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:

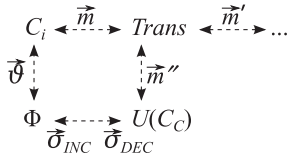

Fig. A·1 Communications between each part of Θ.

$$\begin{aligned}
&\Theta(C_i,C_C) \coloneqq \mathrm{new}\{\vec{m},\vec{m''}\}\ C_i \mid U(C_C) \mid Trans(C_i) \mid \Phi(C_C)\\
&Trans(C_i) \coloneqq \overline{m1}.m1'.\overline{m1''} \mid \ldots \mid \overline{mn}.mn'.\overline{mn''}\\
&\Phi(C_C) \coloneqq \varphi_{m1}^{(0)} \mid \ldots \mid \varphi_{mn}^{(0)}\\
&\vec{m} \coloneqq m1,\ldots,mn\\
&\vec{m'} \coloneqq m1',\ldots,mn'\\
&\vec{m''} \coloneqq m1'',\ldots,mn''
\end{aligned} \tag{A.4}$$

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:

$$\begin{aligned}
&(\overline{\sigma_{x1,INC}}.\ldots.\overline{\sigma_{xn,INC}}).\\
&(x''_{1,1}.\overline{\sigma_{X,DEC}}.\overline{\sigma_{x12,INC}}.x''_{1,2}.\overline{\sigma_{x12,DEC}}\ldots \mid\\
&\qquad\qquad\qquad\qquad\qquad\vdots\\
&\mid x''_{n,1}.\overline{\sigma_{X,DEC}}.\overline{\sigma_{xn2,INC}}.x''_{n,2}.\overline{\sigma_{xn2,DEC}}\ldots)
\end{aligned} \tag{A.5}$$

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 $\vec{m}$.

Figure A·1 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 $\vec{m'}$.

A.2 Concise Grammar of the DSL

Another topic not to be in the body of this text is relative to the formal specification of the DSL. In the appendix we will briefly introduce the grammar of the DSL we propose, which is based on the standard Java 7 grammar. We refer the reader to Ref. [19] for the Java specification and to Ref. [14] for the grammar rules that we reference in this appendix. Both the grammar in this appendix and the one in Ref. [14] are based on the Antlr tool. For more details on Antlr, we refer the reader to Refs. [37], [38]. The basic EBNF syntax is: pipe characters mean an option, question marks mean zero or one occurrences, and asterisks mean zero or more occurrences. We also used such syntax in Eq. (1). For simplicity,

```
statement
 : block | 'fork' block block* | assertion ';'
 | 'if' conditionTestExpression statement
   ('else' statement)?
 | // contract-based if-block
   'if' '(' methodId 'callable' ')' statement
   ('else' statement)?
 | forStatement
 | 'while' conditionTestExpression statement
 | 'do' statement 'while' conditionTestExpression ';'
 | trystatement | switchStatement
 | 'synchronized' conditionTestExpression block
 | 'return' (expression )? ';' | 'throw' expression ';'
 | 'break' (IDENTIFIER)? ';' | 'continue'(IDENTIFIER)? ';'
 | action ';' | IDENTIFIER ':' statement | ';'
 ;

// method id for contract-based if-blocks
methodId : identifier '.' identifier;
```

Fig. A·2 Grammar rule for statements.

```
action
 : assignment | incrementDecrement | call
 | // state transition
   'to' '(' identifier ')'
 ;
```

Fig. A·3 Grammar rule for actions.

we omit most of the Abstract Syntax Tree (AST) tree rewriting rules from the grammar fragments we present in this appendix, but we will need to discuss some of them.

In Antlr, AST trees are represented using the form `^(X a b)` where `X` is the tree root node and `a` and `b` are the children nodes. An example of tree rewriting rule for a parser rule `a` could be `a : 'x' Identifier -> ^('x' Identifier)`. This example means a syntax in which a string starts with 'x' (usually followed by a number of space characters thrown away) followed by a lexer rule called `Identifier` becomes a tree whose root is `x` containing one child whose value is the identifier. Lexer rules are identified by having its first character in upper case, as in `Identifier`.

The four annotations that we introduce in this paper (`@InitialState`, `@State`, `@Scope`, and `@Mobile`) are translated into standard Java annotations but are also treated as class and method modifiers (equivalent to reserved words) in that they figure in the grammar rules and that they do not need to be explicitly imported but are a language feature. We also decided to include them as grammar reserved words in order to have those annotations as roots of enclosing AST trees, as the rewriting rules show.

The grammar in Ref. [14] is a work in process given the complexity of Java. By the time we downloaded such grammar, it was too lenient with modifiers, allowing modifiers that are exclusive to methods to be applied to classes and vice-versa. In order to solve this issue we use `methodModifier` and `typeModifier` grammar rules instead of simply the original `modifier` parser rule. This separation also allowed us to specifically pinpoint where each annotation is to be placed. In the generated Java code, annotations are

```
methodModifier
    : annotation
    | 'public' | 'protected' | 'private' | 'static'
    | 'abstract' | 'final' | 'native'
    // Additional annotations
    | '@State' '(' identifier ')' -> ^(STATE_ identifier)
    | '@Scope' '(' identifier ')' -> ^(SCOPE_ identifier)
    | 'synchronized' ( '(' identifier ')' )?
    | 'strictfp'
    ;
```

Fig. A·4 Grammar rule for method modifiers.

```
typeModifier
    : annotation | 'public' | 'protected' | 'private'
    | 'static' | 'abstract' | 'final' | 'strictfp'
    // Additional annotations
    | '@State' '(' identifier ')' -> ^(STATE_ identifier)
    | '@Scope' '(' identifier ')' -> ^(SCOPE_ identifier)
    | 'synchronized' ( '(' identifier ')' )?
    ;
```

Fig. A·5 Grammar rule for type modifiers.

themselves annotated with `@Target(ElementType.TYPE)` or `@Target(ElementType.METHOD)` in order to specify that they should be applied to types or methods respectively. Keeping annotations in the generated code allows for the middleware to perform reflection at run time.

Figure A·2 shows the grammar rule for statements. The change from the standard Java grammar is identified by the comment. This new rule adds the contract-based if blocks we saw on Section 4.3. We need the `methodId` rule for method references.

Figure A·3 is the grammar for imperative invocations. Besides imperative calls to methods or assignments, we also add the state change action.

Figures A·4 and **A·5** show the new method and type modifiers respectively. Finally, we added the `clientclass` reserved word, which is simply another way to express a class, but with a more specific semantic. We omitted the grammar rule for `clientclass` because it can be easily inferred.

Aurélio Akira Mello Matsui was born in 1978. He received his B.E.E. degree from The University of São Paulo in 2002, and his M.E. degree on Computer Engineering in 2006 from the same university. He was a research student at The University of Tokyo during 2007, and is currently a Ph.D. candidate at the same university. His research interests include distributed systems, programming paradigms, and process calculi.

Hitoshi Aida was born in 1957. He graduated from The University of Tokyo in 1980, and received Doctor of Engineering in 1985. After joining The University of Tokyo, he stayed two years at SRI International in California as an international fellow from 1988 to 1990. He has been a professor of The University of Tokyo since 1999. His current research interests include computer networks, distributed processing, wireless network applications, and disaster resilient networks. He is a senior member of IEEE and a member of ACM, IPSJ, IEICE, JSSST, JSAI and IEIEJ.