Ensuring Faultless Communication Behaviour in an E-Commerce Cloud Application

Rustem A. Kamun and Ross Horne

Kazakh-British Technical University, Faculty of Information Technology,
Almaty, Kazakhstan
r.kamun@gmail.com

Abstract. An increasing scope and complexity of Web Services raises a new challenge of controlling their interaction. The goal of this work is to ensure that processes in a production Cloud are correctly interacting according to a specification of their communication behaviour. To accomplish this goal, we employ session types to analyse the global and local communication patterns. Session types represents "formal blueprints" of how communicating participants should behave and offers a concise view of the message flows.
This work confirms the feasibility of application of session types on "non-linear" business protocols used by an e-commerce Cloud provider and developed in Session-Java, an extension of Java implementing Session-Based programming. Furthermore, we highlight the importance of this approach for services replicated across multiple Cloud providers each of which must correctly cooperate.

1 Introduction

The need for distributed highly available services presents challenges for application development. It is necessary for applications to be integrated both within an enterprise, and between businesses. Service-Oriented Architectures (SOA) are widely accepted as a paradigm for integrating software applications within and across organizational boundaries. In this paradigm, independently deployed applications are exposed as Web Services which are then interconnected using a stack of standards (depicted in Figure 1).

There remain open challenges when it comes to managing service interactions that go beyond simple sequences of requests and responses or involve large numbers of participants (multi-party communication). A need arises for new transaction implementations, more suitable for the Web. One technique for describing collaboration between a collection of services is a choreography model. Choreographies capture the interactions in which the participating services engage and interconnections between these interactions, including control-/data-flow dependencies. However, a choreography does not describe internal effects within a participating service. Furthermore, a choreography does not specify how a global description can be executed.

Much literature exist on the specification of systems that describe services from the local viewpoint [11,4]. The concept of a participant in a communication is essential in complex interactions. Applications include business transactions with short life span, operating in closely coupled context (e.g. the online stock exchange (ForEX), and e-commerce services based on Buyer-Seller-Shipper (BSH) protocols). Although, in

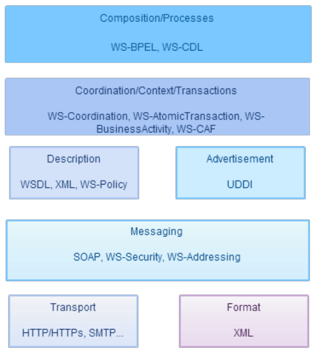
Composition/Processes
WS-BPEL, WS-CDL
Coordination/Context/Transactions
WS-Coordination, WS-AtomicTransaction, WS-BusinessActivity, WS-CAF
Description
WSDL, XML, WS-Policy
Advertisement
UDDI
Messaging
SOAP, WS-Security, WS-Addressing
Transport
HTTP/HTTPs, SMTP...
Format
XML

Fig. 1. Stack of WS Standards

closely coupled settings, the SOA standards may incur a significant performance overhead. Highly available services are likely to have a long life span that may result in deadlock.

Such challenges motivated the design of Web Services Choreography Description Language (WS-CDL) [7]. The WS-CDL working group identified critical issues [2] including:

1. the need for tools to validate conformance to choreography specifications to ensure correct cooperation between Web Services;
2. design time validation and verification of choreographies to guarantee correctness of such properties like deadlock, livelock e.g. behaviour of participants conforms to the choreography interface.

The aforementioned challenges can be tackled by adopting a solid foundational model. Successful approaches based on session types [7,6] include: the Chor and Jolie programming languages of Carbone and Montesi [14,8] based on sessions and trace sets [16]; Session-Java [13], Scribble [12] and Session C [15] due to Honda and Yoshida; Sing# [3] that extend Spec# with choreogrphies; and UBF(B) [1] for Erlang.

In this paper, we demonstrate a method of controlling process interactions represented by sessions. The formal theory based on session types ensures communication safety by verifying that session implementations of each engaged participant conform to the defined protocol specification. In order to evaluate the feasibility of this theory we use Session-Java, an extension to Java language. Session Java works by specifying the intended process transaction protocol using session types and implementing the interaction using session operations.

In Section 2 we provide an overview of SJ. In Section 3, we provide detailed explanation of business protocols used by an e-commerce Cloud provider. Finally, in Section 4, we highlight how session types can improve the design of Intercloud [5] communication protocols.

2 Basics of Session-Java

Session programming begins from the protocols specification for interaction (using session types), which can then be concretely implemented using structured communication operations available on session sockets. Session programming is applicable for applications where the parties or components cooperate according to specified protocols: session types are formal specifications of such protocols. Session types describe structured sequences of interaction including basic message passing, branching and recursion. A session is an instance of a session type, i.e. the unit of interaction encapsulating one run of a protocol. From the perspective of abstraction, each session, is conducted on a separate channel.

Session programming in SJ consists of the following ordered actions:

1. design specification (protocol) of target communication;
2. mapping protocols into the programs for each participant. For instance, in BSH protocols, we can distinguish three main participants whose actions (processes) are mapped to corresponding programs (software component);
3. By utilizing session programming constructs, implementing the protocol, where each operation is performed as method call;
4. verification of sessions fulfilment by compiler;
5. execution and system testing.

2.1 Protocol Specification

Session programming begins by declaring the protocol for the intended cooperation as follows:

$$\text{protocol name } \{\ldots\}$$

where name identifies the protocol, following the standard Java naming rules. The body of the protocol defines a *session type*, according to the grammar in Figure 2.

$T ::= T\,.\,T$	Sequencing
$\mid$ `begin`	Session initiation
$\mid\ !\langle M\rangle$	Message send
$\mid\ ?(M)$	Message receive
$\mid \oplus\{L_1 : T_1, \ldots, L_n : T_n\}$	Session branching
$\mid \oplus[T]*$	Session iteration
$\mid$ `rec` $L\,[T]$	Session recursion scope
$\mid \#L$	Recursive jump
$\mid @p$	Protocol reference

Fig. 2. SJ protocol specification

The session type shows how a session should be designed in terms of actions that the participant should perform. The key point is that the implementation of a session is governed by associated protocol: the SJ compiler (Polyglot[1]). It can be clearly seen from the Table 2, that SJ has enough constructs to describe a diverse range of complex interactions: message passing, conditional and repeated expressions. Its worth noting, that each session type element has its dual element, because there is a requirement that two parties implement compatible protocol such as the specification of one party has dual relation to another party.

2.2 Higher Order Communication

In order to describe richer behaviour, SJ has a feature of subtyping. It means that message types can themselves be session types. It also enhances the agility of the type system by allowing the participants in a session to follow different protocols which are compatible [18]. Such communication can be expressed by the following dual constructs:

$$!\langle ?(\text{int})\rangle \qquad\qquad ?(?(\text{int}))$$

In short, it says that we are expected to send and receive a session of type ?(int). Higher order communication, as we will convince further, is often referred to a session delegation. Figure 3 shows a basic delegation scenario.

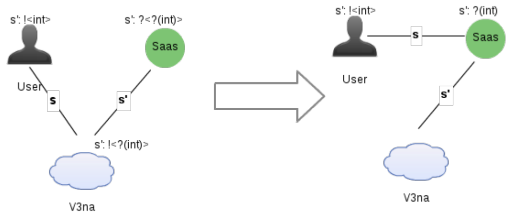

Fig. 3. Session delegation

The left part illustrates the session configuration before the delegation is performed: User engaged in a session s of type !⟨int⟩ with V3na (Cloud), while Cloud is also involved in a session s' with SaaS of type !⟨?(int)⟩. So, instead of accepting the integer from User himself, Cloud delegates his role in s to SaaS, so that he will receive this message. This delegation action corresponds to higher-order send type for the session s' between Cloud and SaaS. The right part of figure illustrates the change in session configuration after the delegation has been performed: User now directly interacting with SaaS for the session s.

2.3 Protocol Implementation using Session

Another meaning for sockets Session sockets are implementing the actual session code according to the specified session type (protocol). They are represent the endpoints (par-

[1] http://www.cs.cornell.edu/Projects/polyglot/. Extensible compile framework.

ticipants) of a session connection: each of the parties owns one endpoint and performs the specified interactions via the SJ session operations on that endpoint. In SJ session sockets are objects that extend the abstract *SJSocket* class. *SJRSocket::SJSocket* and *SJFSocket::SJSocket*, both, employ TCP as underlying transport. SJ is distinguishing session client and server sockets, where the former are used to request sessions from the latter.

Session operations After creating a protocol (session type) and encapsulating the session into SJ socket, it can be implemented within a session-try scope using the session operations depicted in Figure 4.

$s.request()$	`begin`
$s.send(m)$	$!\langle M\rangle$
$s.receive()$	$?(M)$
$s.outbranch(L)\ \{P\}$	$!\{L{:}T\}$
`s.inbranch()` $\{caseL1{:}P1 \ldots caseLn{:}Pn\}$	$?\{L_1: T_1, \ldots, L_n: T_n\}$
$s.outwhile(cond)\ \{P\}$	$[T]*$
$s.inwhile()\ \{P\}$	$?[T]*$
$s.recursion(L)\ \{P\}$	`rec` $L\,[T]$
$s.recurse(L)$	$\#L$

Fig. 4. SJ protocol specification

The session operations are invoked via session in a method call-like manner. To delegate a session, the session socket variable must be passed to a send operation on the target session.

```
s1.send(s2) // !<T>, where T is the remaining session type of '
    s2'
```

Only active session sockets can be delegated. The receive operation receives delegated sessions:

```
SJSocket s2 = s1.receive()
```

3 Business case studies

V3na.com[2] is an e-commerce Web portal that sells SaaS applications for business needs. V3na has developed on Django framework — a high-level Web framework for python. The persistence layer is based on MongoDB and Memcached. One of the challenging task was to automate the process of SaaS integration. By integration we understand the following processes with a particular SaaS application[3]:

[2] http://v3na.com. Cloud platform for optimizing your business performance.

[3] Source code is available at https://github.com/Rustem/Master-thesis.

- connection: SaaS user can connect SaaS for trial period by simply clicking on the button;
- subscription extension and freezing;
- payment confirmation;
- one entry point to all user's applications.

3.1 Simple Scenario

Protocol declaration As a starting point, let's specify simple business protocol of one of the processes just mentioned, SaaS connection. Informally, it may be interpreted as follows:

1. User begins a request session (s) with cloud service (V3na) and sends the request "Connect SaaS" as JSON-encoded message.
2. V3na sends either:
3. FAIL, if user has no active session (not signed in on V3na) and further interaction terminates
4. OK, if user has logged in and request data has passed validation steps. Then Cloud initiates a new session (s') with SaaS and requests it for new user connection with HttpRequestJSONMessage.
5. If OK label take place, Cloud initiates a new session (s') with SaaS and requests it for new user connection with HttpRequestJSONMessage. finally SaaS responds to Cloud with connection status (OK, FAIL) and V3na sends this status to User. Both sessions have to be terminated.

Protocols. The decision in the protocol will be incorporated through the use of out-branch. So the whole scenario is presented on Table 1.

Table 1: Protocols of scenario # 1

User	Cloud (V3na)	SaaS
`protocol p_uv { begin. !<JSONMessage>. ?{ OK: ?(JSONMessage).?(int), FAIL: } }`	`p_vu { begin.?(JSONMessage).!{ OK: !<JSONMessage>.!<int>, FAIL: } protocol http_req_rep { !<JSONMessage>. ? (JSONMessage) } protocol p_vs { begin. @http_req_rep }`	`protocol p_sv { begin. ?(JSONMessage). !<JSONMessage> }`

Interactions. The general syntax for global description has been interpreted into a sequence UML diagram, as depicted in Figure 5. The whole syntax is on the down-left side of the figure. In case of choice, terminated branches are out of scope of the main picture, but still a subpart of the whole diagram. Next step is implementation of this diagram in Session-Java.

The main feature of SJ in this scenario is that choice can be expressed by using *outbranch* construct.

3.2 Complicated Scenario

New scenario is a bit harder in complexity. We are going to utilize looping construct as well as demonstrating *session delegation*. The description of the scenario of this subsection presented below as follows:

1. User begins a request session (s) with cloud service (V3na)
2. V3na asks User to login, so next User provides V3na with login and password Strings.
3. V3na receives User credentials and verifies them: If User is authenticated with minimal amount of tries or amount of tries is out of limit, he is allowed to continue further interactions with V3na, otherwise — not. Go back to step 2.
4. If User is not allowed to access V3na, the interaction between User and V3na continues on DENY-branch, otherwise — on ACCESS-branch.

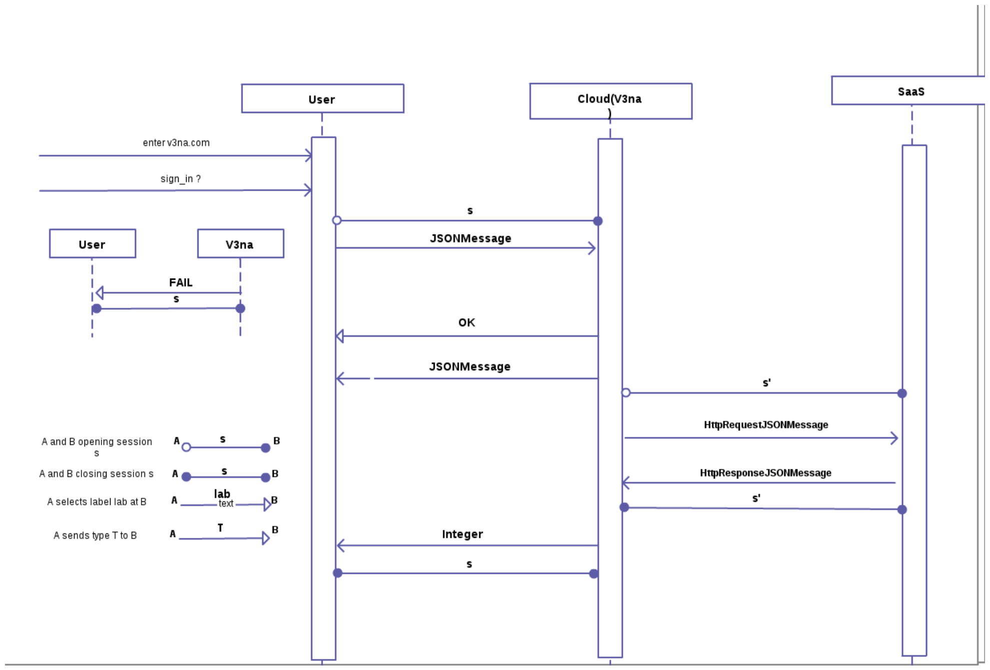

Fig. 5. Overview of interactions for Scenario # 1

5. If next branch is ACCESS, User sends his connection request with details to V3na. V3na creates new session with SaaS (s') and delegates the remaining session s with User on the latter and sends last user request details. Session s' is terminated.
6. SaaS continues interaction with user by session s. By steps of validation-verification, SaaS either responds User to proceed interaction by branch OK or FAIL. In both cases User receives from SaaS directly the reason and status of his request. Session s is terminated.

Protocols. First of all, the protocol provided with iterations using $![\ldots]*\ ?[\ldots]*$. Then protocol introduces higher order operations of type $! < T > \ \ ? < T >$. Full description is provided in tables 2 and 3.

Table 2: User-Cloud protocols

User	Cloud
`protocol p_uv {`	`private protocol p_vu {`
`  begin.?[!<String>.!<String`	`  begin.`
`      > ]*.`	`   ![ ?(String).?(String) //`
`  ?{`	`        login password`
`   ACCESS: !<JSONMessage>.`	`     ]*.`
`    ?{`	`  !{`
`      OK: ?(JSONMessage),`	`   ACCESS: ?(JSONMessage).`
`          FAIL: ?(JSONMessage)`	`    !{`
`    },`	`      OK: !<JSONMessage>,`
`  DENY: ?(String)`	`          FAIL: !<JSONMessage`
`  }`	`          >`
`}`	`     },`
	`      DENY: !<String>`
	`   }`
	`}`

Table 3: Cloud-SaaS protocols

Cloud	Saas
`protocol p_vs {`	`protocol p_msg {`
`  begin.`	`  !{`
`  !< !{`	`    OK: !<JSONMessage>,`
`     OK: !<JSONMessage>,`	`    FAIL: !<JSONMessage>`
`     FAIL: !<JSONMessage>`	`  }`
`  } >.!<JSONMessage>`	`}`
`}`	
	`protocol p_sv {`
	`  begin.?(@p_msg).?(`
	`       JSONMessage)`
	`}`

Unlike the previous protocol, the Cloud-Saas protocol significantly altered, also authentication process is added to the protocol in interaction between User — Cloud. It is important to note that

```
!<!{
        OK: !<JSONMessage>,
        FAIL: !<JSONMessage>
}>
```

corresponds to a higher-order message. The !< ··· > means that it is the Cloud that is passing the high order message and everything inside it is the protocol of the session that Saas should perform with the User. In Saas — Cloud, the protocol defined in more subtle way containing higher order messages by first defining them and then including them in the protocol. For syntactic convenience, one protocol can be referenced from another using @ operator. The $@p$ is syntactically substituted for the protocol of that name.

Interactions. Figure 3.2 depicts the protocols provided above using an UML sequence diagram. The language of the artifacts has already presented in the first scenario.

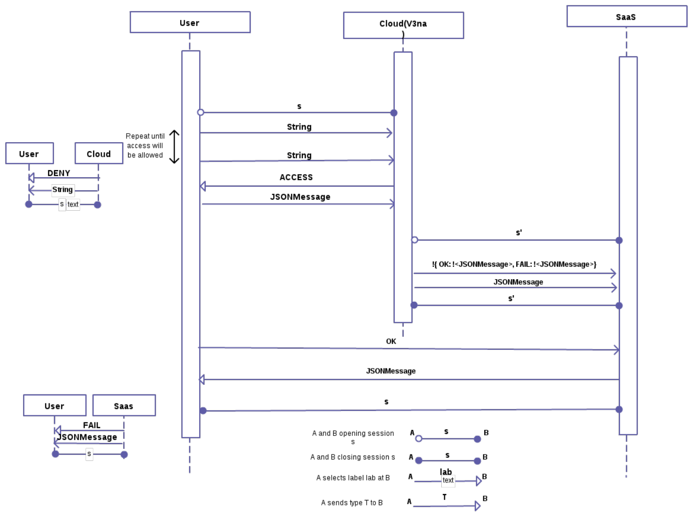

Fig. 6. Sequence diagram of interactions for Scenario # 2

Implementation. Despite the fact that session delegation takes place, the program still remains very simple. Actually delegating the protocol is straightforward and only consists of passing the socket to service:

```
s_vs.send(user_vu); // pass the remaining protocol
```

I have decided to include the whole segment of code to illustrate the following point.

```
user_vu.outbranch(ACCESS) {
  JSONMessage req_info = user_vu.receive();
  SJServerAddress addr_vs = SJServerAddress.create(p_vs,
      saas_hname, saas_port);
  SJSocket s_vs = SJRSocket.create(addr_vs); try(s_vs) {
    s_vs.request();
    s_vs.send(user_vu); // pass the remaining protocol
    s_vs.send(req_info);
  } catch(UnknownHostException uhe) {
    uhe.printStackTrace();
  }
}
```

To receive a high order message type casting must take place in the case of a protocol, the type of protocol must be explicitly defined:

```
v3na_user_socket = (@p_msg) v3na_sv.receive();
```

Where p_msg is defined in the protocols section. This is a reason why it is good practice to first exclusively write the protocol to be delegated and then include it in the final protocol.

4 Intercloud Communication

Nowadays, Cloud computing is moving to the provisioning their data centres as a network of virtual services (hardware, storage, UI, software), so that consumers may access and deploy applications from anywhere in the world in a "pay-as-you-g" manner at competitive costs depending on provided QoS characteristics.

Existing systems that utilized by cloud providers do not support mechanisms and policies for dynamically balancing load among different Cloud-based data centres in order to determine optimal location for hosting application services to achieve rational QoS levels. Cloud providers, Amazon, Google, etc. has the following limitations in their approach:

- *no seamless dynamic scaling* It is difficult for Cloud customers to determine in advance the best location for hosting their services.
- *no general standard for Cloud vendors* SaaS providers may unable to meet QoS characteristics of consumers originating from multiple geographical location.

Besides accomplishment of SLAs (Service-Level Agreements), it is necessary to consider QoS characteristics of service behaviour provided by Cloud vendors, such as request-response rate, service time distributions as well as additional metadata like price, etc. For example, consider a situation that in which Cloud consumer (SaaS owner) needs to use a storage service. Some of them offer this service but with different price and response time. It means that the service provided better quality in terms of mentioned characteristics will be chosen to be included in composition.

We need a middleware platform for composing services provided by Cloud Computing platforms. In paper [17] is called Cloud Exchange, while in [10] is called Cloud Integrator. This middleware must act as a market maker for bringing together service producers and consumers. In short, it aggregates the users technical requirements, and composes the most suitable package of services from different providers. In order to select the most suitable services, paper [Cloud Integrator: Building Value added services] suggests to distinguish types of services provided by different Cloud platforms such as resources as services, applications services, etc., and define each of these services according to its execution flow in an ordered set of tuples (execution plan) like (task, object), e.g. ((store, data), (send, letters), etc.). Such description of services helps to automate selection process, making inferences by Cloud Integrator. It is also important to note that, after finding required execution plans, the selection algorithm must account the metadata to choose the best one.

The author of aforementioned algorithm, suggests an architecture based on SOA standards. His layering model comprised of:

– *Service Layer*, which is responsible for interacting with Cloud platforms in order to create and execute workflows, select and compose services.
– *Integration Layer*,which is responsible to expose a unified API, by building binders per each cloud platform.

First of all, applying SOA standards in such situation is limitation. For instance, Service Layer includes a lot of subcomponents that must interoperate with each other by using middlewares such as CORBA, RPC. In addition, Service Layer interoperate with Cloud platforms through the WSDL. It means that the first part of architecture will lose in performance and flexibility. Also, interactions with Cloud vendors are constant and may result in deadlock.

Secondly, Integration Layer is not standardized. There will be a lot of ground coding work to build specific binder per Cloud provider.

For such a complex system like Cloud Integrator, it is essential to be ensured in faultless behaviour of Web services. Our approach in refining this architecture is to incorporate session types in designing intercloud protocol based on XMPP extension [4] as a transport [9].

References

1. Joe Armstrong. Getting erlang to talk to the outside world. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 64–72. ACM, 2002.
2. Alistair Barros, Marlon Dumas, and Phillipa Oaks. A critical overview of the web services choreography description language. *BPTrends Newsletter*, 3:1–24, 2005.
3. Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. *ACM SIGPLAN Notices*, 47(1):191–202, 2012.
4. Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theoretical computer science*, 37:77–121, 1985.
5. David Bernstein, Erik Ludvigson, Krishna Sankar, Steve Diamond, and Monique Morrow. Blueprint for the intercloud-protocols and formats for cloud computing interoperability. In *ICIW'09.*, pages 328–336. IEEE, 2009.
6. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for Web services. In *Programming Languages and Systems*, pages 2–17. Springer, 2007.
7. Marco Carbone, Kohei Honda, Nobuko Yoshida, and Robin Milner. A theoretical basic of communication-centred concurrent programming. Available at: http://www.eecs.qmul.ac.uk/~carbonem/cdlpaper/workingnote.pdf, 2006.

[4] Extensible Messaging and Presence Protocol, originally created by Jabber

8. Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 263–274. ACM, 2013.
9. Vij D. Bernstein. Using xmpp as a transport in intercloud protocols. In *In Proceedings of CloudComp 2010, the 2nd International Conference on Cloud Computing*, pages 973–982. CiteSeer.
10. T. Batista N. Cacho F. Delicato E. Cavalcante, F. Lopes and P. Pires. Cloud integrator: Building value-added services on the cloud. In *Network Cloud Computing and Applications (NCCA), 2011 First International Symposium on*, pages 135 – 142, 2011.
11. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
12. Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In *Distributed Computing and Internet Technology*, pages 55–75. Springer, 2011.
13. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP 2008*, pages 516–541. Springer, 2008.
14. Fabrizio Montesi and Marco Carbone. Chor — choreography programming language. http://www.chor-lang.org/.
15. Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty session c: Safe parallel programming with message optimisation. In *Objects, Models, Components, Patterns*, pages 202–218. Springer, 2012.
16. Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *Proceedings of the 16th international conference on World Wide Web*, pages 973–982. ACM, 2007.
17. Rajiv Ranjan Rajkumar Buyya and Rodrigo N. Calheiros. Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. *Proceedings of the 10th International Conference on Algorithms and Architectures for Parallel Processing*, 2010.
18. Malcolm Hole Simon Gay. Subtyping for session types in the pi calculus. *Journal Acta Informatica*, 42(2-3):191–225, 2005.