Transforming Service Compositions into Cloud-Friendly Actor Networks ⋆

Dragan Ivanović[1] and Manuel Carro[1,2]

[1] IMDEA Software Institute, Spain
[2] School of Computer Science, T. University of Madrid (UPM), Spain
{dragan.ivanovic,manuel.carro}@imdea.org

Abstract. While conversion of atomic and back-end services from centralized servers to cloud platforms has been largely successful, the composition layer, which gives the service-oriented architecture its flexibility and versatility, often remains a bottleneck. The latter can be re-engineered for horizontal and vertical scalability by moving away from coarser concurrency model that uses transactional databases for keeping and maintaining composition internal state, towards a finer-grained model of concurrency and distribution based on actors, state messaging, and non-blocking write-only state persistence. In this paper we present a scheme for automatically transforming the traditional (orchestration-style) service compositions into Cloud-friendly actor networks, which can benefit from high performance, location transparency, clustering, load balancing, and integration capabilities of modern actor systems, such as Akka. We show how such actor networks can be monitored and automatically made persistent while avoiding transactional state update bottlenecks, and that the same networks can be used for both executing compositions and their testing and simulation.

Keywords: Service Composition, Actor Systems, Cloud Service Provision.

1 Introduction

In recent years, the use of private and public clouds for providing services to users has proliferated as organizations of all sizes embraced the Cloud as an increasingly technically mature and economically viable way to reach markets and meet quality requirements on the global scale. This is especially true for simple (atomic and back-end) services that perform individually small units of work. Such services can be distributed on different cloud nodes, and the requests are routed to different instances based on node availability and load balancing. The key enablers here are distributed databases, which offer high availability and distribution at the price of limited, eventual consistency [7].

Service compositions typically need to store their internal state (point of execution and state variables) along with the domain-specific user data on which they operate. That is needed because service compositions may be long-running and may involve

⋆ The research leading to these results has received funding from the EU FP 7 2007-2013 programme under agreement 610686 POLCA, from the Madrid Regional Government under CM project S2013/ICE-2731 (N-Greens), and from the Spanish Ministry of Economy and Competitiveness under projects TIN-2008-05624 DOVES and TIN2011-39391-C04-03 StrongSoft.

X. Franch et al. (Eds.): ICSOC 2014, LNCS 8831, pp. 291–305, 2014.
© Springer-Verlag Berlin Heidelberg 2014

many internal steps, so that it would be inefficient to let them occupy the scarce server resources (such as threads and database connections) for the whole duration of their execution, most of which is typically spent waiting for responses from other services. Besides, saving the composition state in a persistent store allows resumption after server restarts or network failures. This leads to an essentially event-driven implementation of most composition engines, where incoming events (messages or timeouts) either create new composition instances or wake up dormant ones, which perform a short burst of processing and then either terminate or go to sleep until the next wake-up event.

However, even when eventual consistency on user data is permitted, any inconsistency in the saved internal state of an executing composition may lead to wrong or unpredictable behavior, and must be avoided. That is why most service composition engines, such as Apache ODE [5], Yawl [1], and Orchestra [17], rely on a transactional database to ensure state consistency of long-running processes. This presents a problem for scaling the SOA's composition layer in the Cloud, as concurrent processing of events within the same composition instance implicitly requires access synchronization, transactional isolation, and locking or conflict detection on a central database.

In this paper, we argue that SOA's service composition layer can more successfully exploit the advantages offered by the Cloud if it is based on state messaging rather than mutable shared state kept in a database. This means basing the design of composition engines on well-defined, fine-grained, and Cloud-friendly parallelism and distribution formalisms, rather than "hacking" the existing centralized implementations.

In Section 2, we motivate our approach and outline it in Section 3. Section 4 presents the details of the approach, and Section 5 gives some implementation notes and presents an experimental validation of the approach. We close with conclusions in Section 6.

2 Motivation

According to the *Reactive Manifesto* [4], the ability to react to events, load fluctuations, failures, and user requirements is the distinguishing mark of reactive software components, defined as being readily responsive to stimuli. In this paper, we try to facilitate some of those capabilities in service compositions, starting with service orchestrations with centralized control flow.

```
r := 0;
while ¬empty(in) do begin
  join begin
    send head(in) to P;
    receive x from P
  end and begin
    send head(in) to Q;
    receive y from Q
  end;
  r := r + max(x,y);
  in := tail(in)
end;
send r to caller
```

Fig. 1. Sample composition

Take, for instance, an example currency exchange composition whose pseudo-code is shown in Figure 1. (The syntax and semantics of a sample composition language is given in Section 4.1.) This composition takes a list of amounts in different currencies (in), and tries to find the maximal amount of Euros to which they can be converted, using two external currency conversion services, P and Q. Each amount/currency pair $(\mathsf{head}(in))$ is sent to P and Q in parallel, and the responses (x and y) add to the result (r) before continuing with the rest of the input list $(\mathsf{tail}(in))$. Finally, the result is sent to the caller.

To allow the sample composition to scale both up and out, we need to surpass the limits posed by the shared state store architecture. One way to achieve that is to turn

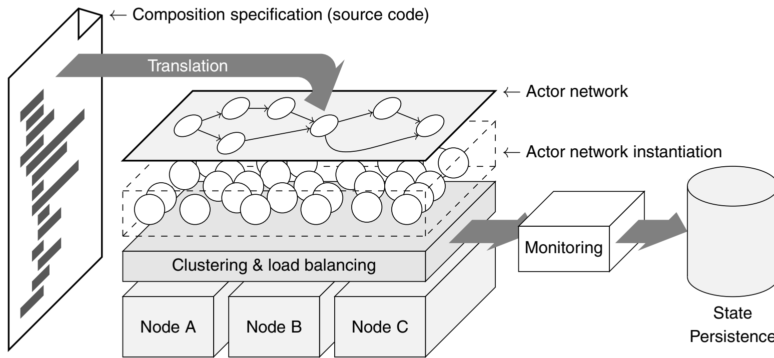

Fig. 2. Outline of the approach

the logical flow of control within the composition into a message flow, by transforming the composition into a network of interconnected stateless, reactive components, each performing a small unit of work, and forwarding results down the logical control flow. Ideally, slower components would be automatically pooled and load-balanced in order to enhance throughput, and/or spread between different nodes in a cluster, depending on available cloud resources. Instead of being kept in a shared data store, the composition state would be reconstructed from observed messages and pushed to a persistent store in a write-only, non-blocking manner.

A major challenge – and the main contribution of this paper – is to find a method for automatically and transparently transforming compositions into such networks of readily scalable reactive components. The transformation needs to hide the underlying implementation details and preserve semantics of state variables, complex control constructs (loops and parallel flows), operations on a rich data model, and message interchange with external services.

We therefore address a similar problem as the concept of Liquid Service Architecture [8], but targeting specific issues in the composition layer, based on formal models of composition semantics and semantically correct transformations.

3 Outline of the Approach

Figure 2 shows the outline of the proposed approach. The starting point is a specification of a composition, expressed in some composition language. This source code is translated into an actor network, which expresses the behavior of the composition as a (statically inferred) collection of stateless, reactive components that perform individually small units of processing. The translation ensures that the behavior of the actor network is consistent with the original semantics of the composition.

We use actor systems [12,13,2,3] as the underlying model of concurrent and distributed computing. Along with π-calculus [16], join-calculus [10], and ambient-calculus [9], actor systems are one of well known approaches to modeling and reasoning about concurrent and distributed computations. However, their component and open asynchronous messaging model makes actor systems closer to the conventional

$$
\begin{array}{rll}
S & ::= \textsf{skip} \mid \textsf{begin}\ S\ \textsf{end} \mid x := E & (\textit{no-op, grouping and assignment}) \\
& \mid \textsf{if}\ C\ \textsf{then}\ S\ \textsf{else}\ S \mid \textsf{while}\ C\ \textsf{do}\ S & (\textit{conditionals and loops}) \\
& \mid S; S \mid \textsf{join}\ S\ \textsf{and}\ S & (\textit{sequential and parallel flows}) \\
& \mid \textsf{send}\ E\ \textsf{to}\ P \mid \textsf{receive}\ x\ \textsf{from}\ P & (\textit{message exchange}) \\
P & ::= \langle \textit{partner service name} \rangle & \\
x & ::= \langle \textit{identifier} \rangle & \\
C, E & ::= \textsf{true} \mid \textsf{false} \mid \textsf{null} \mid \langle \textit{numeral} \rangle \mid \langle \textit{string} \rangle \mid x & \\
& \mid f(E, \ldots, E) \mid E \circ E & (f, \circ \in \text{Builtins}) \\
& \mid \{\} \mid \{x : E[, x : E]^*\} \mid E\{x : E\} \mid E.x & (\textit{records and fields})
\end{array}
$$

Fig. 3. Abstract syntax of a sample composition language

(e.g., object-oriented and functional) programming languages and facilitates efficient implementation (cf. Section 5).

At run-time, the actor network is used as a blueprint to instantiate sets of actors that implement the behavior specified by the network. The instantiated network is deployed into an actor system, where it can benefit from clustering, load balancing, integration and other capabilities of the state-of-the-art actor systems. Being stateless and reactive, the instantiated actors can be scaled both vertically (by organizing them in pools), and horizontally (by distributing them among different interconnected nodes).

The internal state of the executing compositions is not stored in a database, but is kept in messages sent and received by the communicating actors. By monitoring these messages, it is possible to keep an up-to-date snapshot of the state of each executing composition instance, and to record it to a persistent store.

4 Translating Compositions into Actor Networks

4.1 Sample Composition Language

Figure 3 shows the abstract syntax of a composition language fragment. Our intention here is not to "invent" a new composition language, but to present a fragment containing some of the most common control and data handling constructs (found in actual languages like BPEL) whose semantics – control flow, data operations, and messaging – can be formally specified. Such a formal specification of semantics is crucial for reasoning about the correctness of our approach.

The composition language fragment includes state updates (assignments), sequential constructs (such as conditionals and loops), messaging primitives (**send** and **receive**), and **join**-**and** parallel flows which wait for both branches to complete. The language is based on a rich data model that features Boolean, numeric and string literals, the special **null** value, as well as records. Expressions include literals, composition state variables, record constructors, record field accesses, and a set of arithmetic, logical and string built-ins (always terminating).

Records can be used to represent many other data structures. For instance, a list $[A|B]$ with the first element A and the remainder B can be modeled with $\{\textsf{cons} : \textsf{true}, \textsf{head} :$

$$\frac{\{C\wedge\phi \mid \pi\}\, S_1\, \{\phi' \mid \pi'\} \quad \{\neg C\wedge\phi \mid \pi\}\, S_2\, \{\phi' \mid \pi'\}}{\{\phi \mid \pi\}\ \textsf{if}\ C\ \textsf{then}\ S_1\ \textsf{else}\ S_2\ \{\phi' \mid \pi'\}}\ \text{COND} \qquad \frac{}{\{\phi \mid \pi\}\ \textsf{skip}\ \{\phi \mid \pi\}}\ \text{SKIP}$$

$$\frac{\{C\wedge\phi \mid \pi\}\, S\, \{\phi \mid \pi'\}}{\{\phi \mid \pi'\}\ \textsf{while}\ C\ \textsf{do}\ S\ \{\neg C\wedge\phi \mid \pi'\}}\ \text{LOOP} \qquad \frac{\{\phi \mid \pi\}\, S_1\, \{\phi' \mid \pi'\} \quad \{\phi' \mid \pi'\}\, S_2\, \{\phi'' \mid \pi''\}}{\{\phi \mid \pi\}\, S_1\, ;S_2\, \{\phi'' \mid \pi''\}}\ \text{SEQ}$$

$$\frac{}{\{\phi[x\backslash E] \mid \pi\}\ x := E\ \{\phi \mid \pi\}}\ \text{STATE} \qquad \frac{\{\phi \mid \pi\}\, S_1\, \{\phi' \mid \pi'\} \quad \{\phi \mid \pi\}\, S_2\, \{\phi' \mid \pi'\}}{\{\phi \mid \pi\}\ \textsf{join}\ S_1\ \textsf{and}\ S_2\ \{\phi' \mid \pi'\}}\ \text{JOIN}$$

$$\frac{\pi'\ \text{contains no}\ (_ \leftarrow P) \quad \pi''\ \text{contains no}\ (_ \not\leftarrow P)}{\{\phi[x\backslash u] \mid \pi'(u\leftarrow P)\pi''\}\ \textsf{receive}\ x\ \textsf{from}\ P\ \{\phi \mid \pi'(u \not\leftarrow P)\pi''\}}\ \text{RECV}$$

$$\frac{\phi \vdash E = u \quad \pi''\ \text{contains no}\ (_ \rightarrow P)\ \text{or}\ (_ \not\leftarrow P)}{\{\phi \mid \pi'\pi''\}\ \textsf{send}\ E\ \textsf{to}\ P\ \{\phi \mid \pi'(u\rightarrow P)\pi''\}}\ \text{SEND}$$

Fig. 4. Abstract semantics of a fragment of the composition language

A, tail : $B\}$, and the empty list $[]$ with $\{$cons : **false**$\}$. In turn, records and lists can represent JSON and XML documents. In examples, we use sans serif font to distinguish field, built-in and other global names from local names in cursive.

We use a form of axiomatic semantics to specify the meaning of control constructs, data operations, and message exchanges for the language fragment, with the inference rules (axiom schemes) shown in Figure 4. The pre- and post-conditions are expressed in the form $\{\phi \mid \pi\}$, where logic formula ϕ characterizes the composition state as in the classic Hoare Logic [14,6], and π is a chronological sequence of outgoing messages $(u \rightarrow P)$, incoming unread messages $(u \leftarrow P)$, and incoming read messages $(u \not\leftarrow P)$, where u is a datum. The consequence rule, which states that pre-conditions can always be strengthened as well as post-conditions weakened, is implicit. Condition $\{\phi' \mid \pi'\}$ is stronger than $\{\phi \mid \pi\}$ iff ϕ' logically implies ϕ (in the data domain theory), and π is a (possibly non-contiguous) sub-sequence of π'.

Rules COND, SKIP, LOOP, SEQ, and STATE are direct analogues of the classical Hoare Logic rules for sequential programs. The abstract semantics of parallel and-join flow is given in rule JOIN. The parallel branches are started together, and race conditions on state variables and partner services are forbidden: variables modified by one branch cannot be read of modified by the other, and the branches cannot send or receive messages to or from a same partner service.

In rule RECV, the conditions on π' and π'' ensure that messages are read in the order in which they are received, and the condition on π'' in rule SEND ensures the chronological ordering of outgoing messages. The underscores here denote arbitrary data. The message exchange is asynchronous, and thus the relative ordering of messages to/from a partner matters more than the absolute ordering of all messages.

4.2 Actor Language

The abstract syntax of a functional actor language is given in Fig. 5, along the lines of Aga et al. [3] and Varela [18], with some syntactic modifications. Its domain of values (V) is the same as in the sample composition language, with addition of actor references (A) used for addressing messages. The expressions (E) extend expressions in the composition language with functional and actor-specific constructs.

$$
\begin{array}{rll}
E ::= & L \mid x \mid \lambda x \to E \mid E(E) \mid \textbf{rec}(E) & (\textit{standard } \lambda\textit{-calculus constructs}) \\
& \mid f(E,...,E) \mid E \circ E & (f, \circ \in \text{Builtins}) \\
& \mid R_E \mid E\{x : E\} & (\textit{record of expressions, update}) \\
& \mid \textbf{match } E \textbf{ with } T \to E[;T \to E]^* \textbf{ end} & (\textit{pattern matching}) \\
& \mid \textbf{new}(E) \mid \textbf{stop} & (\textit{actor creation \& termination}) \\
& \mid \textbf{ready}(E) \mid \textbf{send}(E,E) & (\textit{message reception \& dispatch}) \\
L ::= & \textbf{true} \mid \textbf{false} \mid \textbf{null} \mid \langle \textit{numeral} \rangle \mid \langle \textit{string} \rangle & (\textit{primitive value}) \\
V ::= & L \mid A \mid R_V & (\textit{value}) \\
R_\Phi ::= & \{\} \mid \{x : \Phi[, x : \Phi]^*\} & (\textit{record structure}) \\
T ::= & x \mid _ \mid L \mid R_T & (\textit{pattern}) \\
x ::= & \langle \textit{identifier} \rangle \qquad A ::= \langle \textit{actor reference} \rangle &
\end{array}
$$

Fig. 5. The basic actor language

Function abstractions and applications from λ-calculus are included together with the special recursion operator **rec**. The **match** construct searches for the first clause $T \to E$ where pattern T matches a given value, and then executes E to the right of "$\to$". At least one match must be found. Variables in patterns capture matched values, and each underscore stands for a fresh anonymous variable. The order of fields in record patterns is not significant, and matched records may contain other, unlisted fields. Several common derived syntactic forms are shown in Table 1.

Construct **new** creates a new actor with the given behavior, and returns its reference. An actor behavior is a function that is applied to an incoming message. Construct **ready** makes the same actor wait for a new message with the given behavior. Construct **send** sends the message given by its second argument to the agent reference to which the first argument evaluates. Finally, **stop** terminates the actor.

Fig. 6 shows two simple examples of actor behaviors. The sink behavior simply accepts a message (m) without doing anything about it, and repeats itself. The cell behavior models a mutable cell with content x. On a 'get' message, the current cell value x is sent to the designated recipient a, and the same behavior is repeated. On a 'set' message, the cell forgets the current value x and repeats the same behavior with the new value y. Note how in both cases the construct **rec** allows the behavior to refer to itself via b.

$$
\begin{array}{l}
\text{sink} \equiv \textbf{rec}(\lambda b \to \lambda m \to \textbf{ready}(b)) \\
\\
\text{cell} \equiv \textbf{rec}(\lambda b \to \lambda x \to \lambda m \to \\
\quad \textbf{match } m \textbf{ with} \\
\qquad \{\text{get: } a\} \to \textbf{do send}(a, x) \textbf{ then ready}(b(x)); \\
\qquad \{\text{set: } y\} \to \textbf{ready}(b(y)) \\
\quad \textbf{end})
\end{array}
$$

Fig. 6. Two simple actor behaviors

Table 1. The derived actor language constructs and abbreviations

Abbreviation	Basic construct
let $x = E_1$ **in** E_2	**match** E_1 **with** $x \to E_2$ **end**
do E_1 **then** E_2	**match** E_1 **with** $_ \to E_2$ **end**
if E_1 **then** E_2 **else** E_3	**match** E_1 **with true** $\to E_2$; $_ \to E_3$ **end**
$E.x$	**match** E **with** $\{x : y\} \to y$; $_ \to$ **null end**

The operational semantics of actor systems is expressed in terms of transitions between actor configurations. Each actor configuration $\langle\!\langle\, \alpha \parallel \mu \,\rangle\!\rangle$ consists of a set of actors α and a bag (multiset) μ of messages in transit. Elements of α are written as $[E]_a$, denoting an actor with the unique address (actor reference) $a \in A$ and behavior E. Elements of μ are written as $(a \Leftarrow v)$, denoting a value v sent to an actor whose reference is a. In rules, both α and μ are written as unordered sequences without repetition of elements.

We first frame the actor expressions in terms of redexes and reduction contexts, shown in Figure 7. It can be shown that any actor expression E can be uniquely framed in the form $E_\Box \rhd e \lhd$, where $E_\Box$ is a reduction context which contains exactly one hole ($\Box$) which is filled by redex e. A redex is the next sub-expression to be evaluated (and replaced with the evaluation result, if any) in the left-to-right call-by-value evaluation strategy. The exceptions are stand-alone values (V) and function abstractions ($\lambda x \to E$), which are syntactically valid, but do not denote any meaningful actor behavior.

The transitions defining the operational semantics of the actor language are given in Figure 8. The purely functional redexes follow the relation "$\longrightarrow_\lambda$" and are reduced locally within an actor under rule FUN. For instance, APP is β-reduction from λ-calculus, REC defines the behavior of the recursion operator, UPD defines record updates, and BI the application of built-ins. Rule MATCH_1 fires if there exists a substitution θ of variables from pattern T which makes $T\theta$ identical to the value v that is matched; in that case, the **match** expression reduces to the expression $E\theta$, i.e., E to the right of "$\to$" with these variable substitutions applied. Rule MATCH_2 throws away the first pattern if a matching substitution cannot be found, and continues with the rest.

The actor redexes are regulated by rules other than FUN. In STOP, any actor that encounters **stop** is immediately terminated. Rule NEW creates a new actor which becomes **ready** to execute behavior w given by **new**, and returns its address a' to the creating actor. Rule READY says that whenever an actor executes construct **ready**, it blocks if necessary until there is a message v sent to it, and then starts from the scratch by applying the behavior w given by **ready** to the message. Finally, rule SEND creates a new message for the receiver, and returns **null** on the sender side.

An important characteristic of the actor system semantics is fairness, in the sense that all enabled transitions eventually fire. In particular, this means that every message sent to an actor is eventually received, unless the actor is terminated, halted by an error, or caught in an infinite loop while processing an earlier message.

$$
\begin{aligned}
W &::= V \mid \lambda x \to E \\
e &::= W(W) \mid x \mid f(W,\dots,W) \mid W \circ W \mid \textbf{rec}(W) \mid R_W \mid W\{x:W\} \\
&\quad \mid \textbf{match}\ W\ \textbf{with}\ T \to E[;T \to E]^*\ \textbf{end} \mid \textbf{new}(W) \mid \textbf{stop} \mid \textbf{ready}(W) \mid \textbf{send}(W,W) \\
E_\Box &::= \Box \mid W(E_\Box) \mid E_\Box(E) \mid f(W,\dots,W,E_\Box,E,\dots,E) \mid W \circ E_\Box \mid E_\Box \circ E \\
&\quad \mid \textbf{rec}(E_\Box) \mid \{x:W,\dots,x:W,x:E_\Box,x:E,\dots,x:E\} \mid W\{x:E_\Box\} \mid E_\Box\{x:E\} \\
&\quad \mid \textbf{match}\ E_\Box\ \textbf{with}\ T \to E[;T \to E]^*\ \textbf{end} \mid \textbf{new}(E_\Box) \mid \textbf{ready}(E_\Box) \mid \textbf{send}(W,E_\Box) \mid \textbf{send}(E_\Box,E)
\end{aligned}
$$

Fig. 7. Reduction contexts and redexes

$$\frac{w \in W}{(\lambda x \to E)(w) \longrightarrow_\lambda E[x \backslash w]}\ \text{APP} \qquad \frac{e \equiv \mathbf{rec}(\lambda x \to E)}{e \longrightarrow_\lambda E[x \backslash e]}\ \text{REC} \qquad \frac{r \equiv \{x : _, \varphi\} \quad v \in V}{r\{x : v\} \longrightarrow_\lambda \{x : v, \varphi\}}\ \text{UPD}$$

$$\frac{f^n \in \text{Builtins} \quad n \geq 0 \quad [\![f^n]\!] : V^n \to V}{f^n(v_1, \ldots, v_n) \longrightarrow_\lambda [\![f^n]\!](v_1, \ldots, v_n)}\ \text{BI} \qquad \frac{v \in V \quad \exists \theta \cdot v \equiv T\theta}{(\mathbf{match}\ v\ \mathbf{with}\ T \to E[; \tau]\ \mathbf{end}) \longrightarrow_\lambda E\theta}\ \text{MATCH}_1$$

$$\frac{v \in V \quad \nexists \theta \cdot v \equiv T\theta}{(\mathbf{match}\ v\ \mathbf{with}\ T \to E; \tau\ \mathbf{end}) \longrightarrow_\lambda (\mathbf{match}\ v\ \mathbf{with}\ \tau\ \mathbf{end})}\ \text{MATCH}_2$$

$$\frac{e \longrightarrow_\lambda e'}{\langle\!\langle\, \alpha, [E_\Box \rhd e \lhd]_a \parallel \mu \,\rangle\!\rangle \longrightarrow \langle\!\langle\, \alpha, [E_\Box \rhd e' \lhd]_a \parallel \mu \,\rangle\!\rangle}\ \text{FUN} \qquad \frac{}{\langle\!\langle\, \alpha, [E_\Box \rhd \mathbf{stop} \lhd]_a \parallel \mu \,\rangle\!\rangle \longrightarrow \langle\!\langle\, \alpha \parallel \mu \,\rangle\!\rangle}\ \text{STOP}$$

$$\frac{w \in W \quad a' \in A \text{ fresh}}{\langle\!\langle\, \alpha, [E_\Box \rhd \mathbf{new}(w) \lhd]_a \parallel \mu \,\rangle\!\rangle \longrightarrow \langle\!\langle\, \alpha, [E_\Box \rhd a' \lhd]_a, [\mathbf{ready}(w)]_{a'} \parallel \mu \,\rangle\!\rangle}\ \text{NEW}$$

$$\frac{w \in W}{\langle\!\langle\, \alpha, [E_\Box \rhd \mathbf{ready}(w) \lhd]_a \parallel \mu, (a \Leftarrow v) \,\rangle\!\rangle \longrightarrow \langle\!\langle\, \alpha, [w(v)]_a \parallel \mu \,\rangle\!\rangle}\ \text{READY}$$

$$\frac{a' \in A \quad v \in V}{\langle\!\langle\, \alpha, [E_\Box \rhd \mathbf{send}(a', v) \lhd]_a \parallel \mu \,\rangle\!\rangle \longrightarrow \langle\!\langle\, \alpha, [E_\Box \rhd \mathbf{null} \lhd]_a \parallel \mu, (a' \Leftarrow v) \,\rangle\!\rangle}\ \text{SEND}$$

Fig. 8. Operational semantics of the actor language

4.3 Translating Compositions into Actor Networks

After explaining the syntax and semantics of the sample composition language and the actor language, we now proceed with the crucial step in our approach: the transformation of a service composition into an actor network.

An actor network is a statically generated set of actor message handling expressions that correspond to different sub-constructs in a composition. At run-time, actor networks are instantiated into a set of reactive, stateless actors, which accept, process and route information to other actors in the network, so that the operational behavior of the instantiated network is correct with respect to the abstract semantics of the composition language. The stateless behavior of the actors in an instantiated network enables their replacement, pooling, distribution, and load-balancing.

For a composition S, by $\mathscr{A}[\![S]\!]$ we denote its translation into an actor network, as a set whose elements have the form $\ell_i : E_i$ or $\ell_i \mapsto \ell_j$. Here, ℓ_i and ℓ_j are (distinct) code location labels, which are either 0 (denoting composition start), 1 (denoting composition finish), or are hierarchically structured as $\ell.d$, where d is a single decimal digit (denoting a child of ℓ). Element $\ell_i : E$ means that the behavior of the construct at ℓ_i is realized with actor behavior E over input message m. Element $\ell_i \mapsto \ell_j$ means that ℓ_i is an alias for ℓ_j. Alias $\ell_i \mapsto \ell_j$ is sound iff $\mathscr{A}[\![S]\!]$ contains either $\ell_j : E_j$ or $\ell_i \mapsto \ell_k$ such that $\ell_k \mapsto \ell_j$ is sound. Unsound or circular aliases are not permitted.

$\mathscr{A}[\![S]\!]$ is derived from the structure of S, by decomposing it into simpler constructs. Figure 9 shows the translations $\mathscr{A}[\![S']\!]_\ell^{\ell'}$ for each construct S' located at ℓ, and immediately followed by a construct at ℓ'. For the whole composition, $\mathscr{A}[\![S]\!] = \mathscr{A}[\![S]\!]_0^1$. Items P, ℓ, ℓ', $\ell.1$, $\ell.2$, etc. are treated as string literals in actor expressions.

The translation of **skip** simply maps the behavior of location ℓ to that of ℓ', without introducing new actors. For other constructs, the structure of the incoming message m is relevant: $m.\mathsf{inst}$ holds the unique ID of the composition instance; $m.\mathsf{loc}$ maps location labels to actor addresses (discussed below); $m.\mathsf{env}$ is a record whose fields are the

$$
\begin{aligned}
\mathscr{A}\llbracket \textbf{skip} \rrbracket_{\ell}^{\ell'} &= \{\ell \mapsto \ell'\} \\
\mathscr{A}\llbracket x := E \rrbracket_{\ell}^{\ell'} &= \{\ell : (\textbf{send}(\mathsf{fget}(m.\mathsf{loc}, \ell'), m\{\mathsf{env}.x : \bar{E}\}\{\mathsf{from} : \ell\}))\} \\
\mathscr{A}\llbracket \textbf{if } C \textbf{ then } S_1 \textbf{ else } S_2 \rrbracket_{\ell}^{\ell'} &= \{\ell : (\textbf{send}(\mathsf{fget}(m.\mathsf{loc}, \textbf{if } \bar{C} \textbf{ then } \ell.1 \textbf{ else } \ell.2), m\{\mathsf{from} : \ell\}))\} \\
&\quad \cup \mathscr{A}\llbracket S_1 \rrbracket_{\ell.1}^{\ell'} \cup \mathscr{A}\llbracket S_2 \rrbracket_{\ell.2}^{\ell'} \\
\mathscr{A}\llbracket \textbf{while } C \textbf{ then } S \rrbracket_{\ell}^{\ell'} &= \{\ell : (\textbf{send}(\mathsf{fget}(m.\mathsf{loc}, \textbf{if } \bar{C} \textbf{ then } \ell.1 \textbf{ else } \ell'), m\{\mathsf{from} : \ell\}))\} \cup \mathscr{A}\llbracket S \rrbracket_{\ell.1}^{\ell} \\
\mathscr{A}\llbracket S_1 ; S_2 \rrbracket_{\ell}^{\ell'} &= \{\ell \mapsto \ell.1\} \cup \mathscr{A}\llbracket S_1 \rrbracket_{\ell.1}^{\ell.2} \cup \mathscr{A}\llbracket S_2 \rrbracket_{\ell.2}^{\ell'} \\
\mathscr{A}\llbracket \textbf{send } E \textbf{ to } P \rrbracket_{\ell}^{\ell'} &= \{\ell : (\textbf{do send}(\mathsf{fget}(m.\mathsf{link}, P), m\{\mathsf{out} : \bar{E}\}) \textbf{ then} \\
&\qquad \textbf{send}(\mathsf{fget}(m.\mathsf{loc}, \ell'), m\{\mathsf{from} : \ell\}))\} \\
\mathscr{A}\llbracket \textbf{receive } x \textbf{ from } P \rrbracket_{\ell}^{\ell'} &= \{\ell : (\textbf{send}(\mathsf{fget}(m.\mathsf{link}, P), m\{\mathsf{in} : \text{"}x\text{"}\}\{\mathsf{from} : \ell\}\{\mathsf{to} : \ell'\}))\} \\
\mathscr{A}\llbracket \textbf{join } S_1 \textbf{ and } S_2 \rrbracket_{\ell}^{\ell'} &= \{\ell : (\textbf{let } m_2 = m\{\mathsf{from} : \ell\}\{\mathsf{loc} : \mathsf{fset}(m.\mathsf{loc}, \ell.2, \textbf{new}(\mathscr{J}\llbracket S_1, S_2 \rrbracket_{\ell}^{\ell'}(m)))\} \\
&\qquad \textbf{in do send}(\mathsf{fget}(m.\mathsf{loc}, \ell.1.1), m_2) \textbf{ then send}(\mathsf{fget}(m.\mathsf{loc}, \ell.1.2), m_2))\} \\
&\quad \cup \mathscr{A}\llbracket S_1 \rrbracket_{\ell.1.1}^{\ell.2} \cup \mathscr{A}\llbracket S_2 \rrbracket_{\ell.1.2}^{\ell.2} \\
\mathscr{J}\llbracket S_1, S_2 \rrbracket_{\ell}^{\ell'} &\equiv \lambda m \to \lambda m_1 \to \textbf{ready}(\lambda m_2 \to \textbf{do send}(\mathsf{fget}(m.\mathsf{loc}, \ell'), \\
(S_2 \textit{ writes } \bar{z}) &\qquad (\textbf{if } m_1.\mathsf{from} \geq \ell.1.1 \textbf{ then } m\{\mathsf{env} : m_1.\mathsf{env}\overline{\{z : m_2.\mathsf{env}.z\}}\} \\
(S_1 \textit{ writes } \bar{y}) &\qquad \textbf{else } m\{\mathsf{env} : m_2.\mathsf{env}\overline{\{y : m_1.\mathsf{env}.y\}}\})\{\mathsf{from} : \ell.2\}) \textbf{ then stop}
\end{aligned}
$$

Fig. 9. Translation of composition constructs as actor networks

composition state variables with their current values; and m.link is a map from available partner service names to references of the actors which serve as their mailbox interfaces. The initial content of m is set up upon the reception of the initiating message with which the composition is started. For simplicity, we assume that m.env.in holds the input message, and that the initiating party is by convention called caller.

The translation of an assignment uses the built-in fget to fetch the value of m.loc associated with ℓ' (as a string literal). That value is the reference of the next actor in the flow, to which a message is sent with the modified value of the assigned variable x. With $\bar{E}$ we denote the result of replacing each state variable name y encountered in E with m.env.y. Here, as in other translations, we additionally modify the from field in m to hold the location from which the message is sent.

The translation of the conditional creates two sub-locations, $\ell.1$ and $\ell.2$ to which it translates the then- and the else-part, respectively. Then, at run-time the incoming message is routed down one branch or another, depending on the value of the condition $\bar{C}$ (which is rewritten from C in the same way as $\bar{E}$ from E in assignment). The translation of the **while** loop is analogous to that of the conditional. When a sequence is translated, two sub-locations $\ell.1$ and $\ell.2$ are created and chained in a sequence.

The translations of the messaging primitives rely on partner links in m.link. For **send**, the outgoing message is asynchronously sent to the partner link, wrapped in m.out, and then the incoming message is forwarded to the next location in the flow. For **receive**, the partner mailbox is asked to forward m to ℓ' when the incoming message becomes available, by placing it in m.env under the name of the receiving variable.

The most complex behavior is for the **join** construct, which needs to create a transient join node (at $\ell.2$) which collects and aggregates the results of both parallel branches before forwarding it to ℓ'. The branches are translated at $\ell.1.1$ and $\ell.1.2$. The branches receive message m_2 whose m_2.loc is modified (using the built-in fset) to point to the transient join node under $\ell.2$. Its behavior of is defined by $\mathscr{J}\llbracket S1, S2 \rrbracket_{\ell}^{\ell'}$: m is the

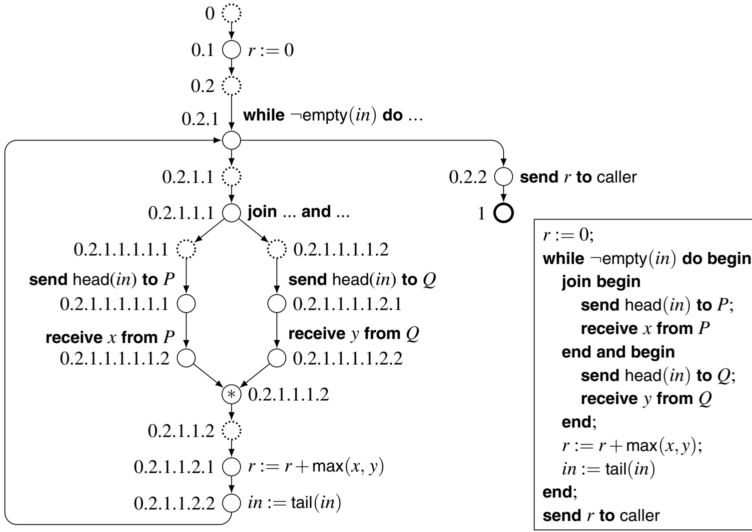

Fig. 10. Deployment of the example composition between $\ell = 0$ and $\ell' = 1$

original incoming message, and m_1 and m_2 are messages received from the branches. The outgoing message is based on m, and inherits env from the first branch to terminate, with the added modifications from the other one: the value of each state variable z written by S_2 (or state variable y written by S_1) is copied into the resulting environment.

Figure 10 shows the topology of the actor network resulting from the translation of the our example composition, annotated with location labels and the corresponding composition constructs, with the message flow indicated with arrows. The transient node is marked with an asterisk, and the dotted nodes correspond to the sequences and are aliased to the next node in the flow.

4.4 Actor Network Instantiation and Semantic Correctness

An instantiation of an actor network $\mathscr{A}[\![S]\!]$ is a pair $\langle \Lambda, \alpha \rangle$, where Λ is a (partial) mapping from locations to actor references, and α a minimal set of actors such that (a) for each $\ell : E \in \mathscr{A}[\![S]\!]$, there is $[\textbf{ready}(\textbf{rec}(\lambda b \to m \to \textbf{do } E \textbf{ then ready}(b)))]_{\Lambda(\ell)} \in \alpha$; and (b) for each $\ell_1 \mapsto \ell_2 \in \mathscr{A}[\![S]\!]$, $\Lambda(\ell_1) = \Lambda(\ell_2)$. When a new composition instance is created, its m.loc is set to Λ.

The following theorem is central to validating the correctness of the approach:

Theorem 1. *The operational behavior of any instantiation $\langle \Lambda, \alpha \rangle$ of $\mathscr{A}[\![S]\!]$ is correct with respect to the abstract semantics of S.*

The correctness criteria applies to any valid triplet $\{\phi \mid \pi\}\ S\ \{\phi' \mid \pi\pi'\}$ that can be inferred from the rules in Figure 4 (with the implicit consequence rule), and any instantiation $\langle \Lambda, \alpha \rangle$ of $\mathscr{A}[\![S]\!]$. It requires that whenever S terminates and ϕ holds on the input message received at location 0: (i) exactly one output message (of the same

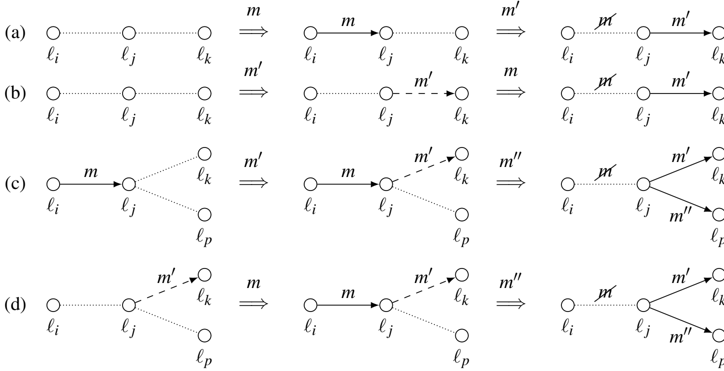

Fig. 11. Example updates of an instance snapshot

instance) is sent to location 1; (ii) ϕ' holds on the output message; and (iii) the messages sent to and received from the external service mailboxes are compatible with π'.

The proof of this theorem is based on structural induction of correctness on the building blocks of S. For each building block, the operational semantics of the actors in α (augmented with partner mailbox actors) is validated against the pre- and post-conditions defined in the abstract semantics of the composition language, applied to the content and the circulation of messages that belong to a same composition instance.

Note that the behavior $\textbf{rec}(\lambda b \to m \to \textbf{do}\ E\ \textbf{then}\ \textbf{ready}(b))$ with which new actors are created is fully stateless and repetitive, and thus a single actor can be seamlessly replaced with a load-balanced and possibly dynamically resizable pool of its replicas attached to the same location, without affecting the semantics of the instantiation.

4.5 Composition State Persistence

By observing all messages sent between actors in the instantiated actor network (with the addition of partner service mailbox actors), a monitor can keep the current snapshot of the execution state for each executing instance, distinguished by $m.\mathsf{inst}$. The snapshot can be represented as a tuple $\langle \sigma, \varsigma \rangle$, where σ is the stable, and ς the unstable set of observed messages. The two sets are needed because messages can arrive out of order.

For example, part (a) of Figure 11 treats the case of location ℓ_j which needs one incoming, and produces one outgoing message. When messages come in order, the incoming message m from ℓ_i is placed in σ and is subsequently replaced with the outgoing message m'. It may, however, happen, as in Figure 11(b), that the outgoing message m' is observed first. In that case, it is placed in ς (indicated by the dashed line). When the incoming message m is observed, it is discarded, and m' is moved from ς to σ. Two analogous cases for a location corresponding to a parallel split node, which sends two outgoing messages, is shown in Figure 11(c)-(d). In these examples, we tacitly merge the aliased locations together, and include the partner service mailboxes.

After each observation, the stable set of observed messages σ can be written to a persistent data store, and used for reviving the execution of the instance in case of a system stop or crash, simply by replaying the messages from σ. This may cause

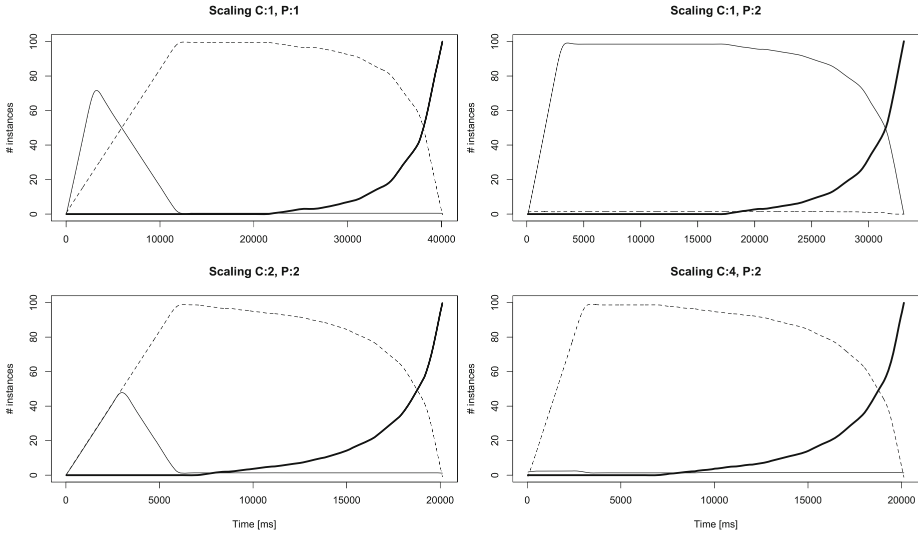

Fig. 12. Dynamic behavior of the sample composition deployed as an actor network

repetition of some steps (including the external service invocations), whose completion has not been observed when the last stable set was committed to the persistent store, but the messages in σ always represent a complete and consistent instance snapshot.

4.6 Use for Testing and Simulation

The presented actor network translation and instantiation scheme, which aims at executing compositions in a production environment, can also serve as a basis for service testing and simulation. Service composition can be tested by observing the messages between the locations in the actor network to verify pre- and post-conditions at various points in the composition code. For that purpose, the syntax of composition statements (S) in Figure 3 can be extended with assertions that express conditions on state variables and external messages. A testing monitor can then check the conditions at run-time and compute code and path coverage of the tests.

In a simulation mode, external service mailboxes can be replaced with mock-ups, and the translation scheme in Figure 9 can be slightly extended to include a new message field $m.\mathsf{time}$ that represents the simulated time. Such a simulation could be used to study the behavior of the system under different load scenarios, and would have the advantage of correctly modeling its state, logic, and control flow.

5 Implementation Notes and Experimental Validation

We base our implementation of the proposed approach on Akka [11], a toolkit and runtime for building concurrent, distributed, and fault tolerant event-driven applications on the JVM platform. Among other capabilities, Akka enables transparent remote actor

creation, supervision and communication between different network nodes, as well as easily configurable actor pooling and load balancing. Akka also easily integrates with Apache Camel [15], a versatile integration framework which allows actor systems to interface with external services and systems using a large number of standard protocols.

To evaluate the potential benefits of the proposed approach with respect to the performance of service compositions, we have implemented the sample currency conversion composition from Figures 1 and 10 as an actor network, which was then instantiated with different scaling factors, where the composition scaling factor n means that each logical actor from the actor network is instantiated as a pool of n load-balanced actors. The external service invoked within the loop has also been implemented in a scalable manner, with a scaling factor of its own, and with a round-trip invocation time (of a solitary request in a quiescent system) between 12 ms and 18 ms. In each experimental run, the composition was fed with 100 input requests (at intervals of 10 ms) with input lists of size 10, and the messages within the actor network were monitored to reveal the number of instances awaiting or undergoing processing at different locations in the network.[1]

The top-left graph in Figure 12 shows the results for the base configuration, where the scaling factor for both the composition (C) and the external service (P) is 1, which means that each actor in C, as well as P, could process one request at a time. The thick solid rising line gives the number of finished instances over time (i.e., the number of messages reaching location $\ell' = 1$ in Figure 10). It takes approx. 40 s for the entire train of 100 input requests to be served. The dashed line shows the backlog of invocations to P, and the thin solid line shows the backlog of all internal operations in the composition, such as the control constructs and assignments. As the requests arrive, the backlog of internal operations quickly builds up, and then recedes as more and more instances block waiting on P.

Since P represents an obvious bottleneck, a common-sense approach would be to scale it up. The top-right graph in Figure 12 shows the behavior of a configuration where the scaling factor for C is kept at 1, and that of P increased to 2. However, it turns out that in spite of modest performance improvements (cutting the overall execution time by 17.5% from 40 s to approx. 33 s), C cannot significantly exploit the benefits of scaling up P without scaling up itself. In fact, the graph shows that the backlog of P now almost disappears, while the backlog of the internal operations in C now dominates the dynamics of the system. In this case, the reason for such highly non-linear aggregate behavior is the effective halving (on the average) of P's request-response time, which now becomes shorter than the interval between incoming requests.

The bottom-left graph in Figure 12 shows the case when both C and P have scaling factor 2. In this case, the composition performance of the system is practically doubled, with the overall execution time cut from 40 s to approx. 20 s. The bottom-right graph in Figure 12 suggests that scaling C more than P does not yield significant performance improvements: an appropriate scaling strategy seems to be using the same scaling factor for both C and P. Note that in our approach the scaling factor for C can be configured at will at run-time, without affecting the transformation.

[1] The experiment was performed on a Mac Airbook computer with 1.7 GHz Intel Core i5 and 4 GB of RAM, running Mac OS X 10.9.2, Oracle Java 1.7_55, and Akka 2.3.2.

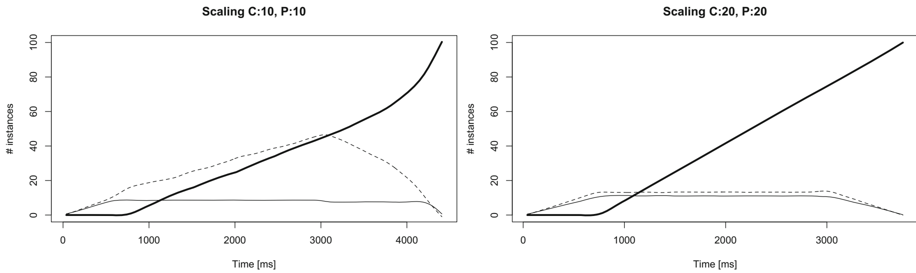

Fig. 13. Limits of performance improvements when increasing the scaling factor

Figure 13 shows how the effects on performance degrade as the common scaling factor increases. While scaling factors $n = 2$, $n = 4$ and $n = 10$ reduce the overall execution time almost proportionally by factors 1.993, 3.937, and 9.097, respectively, for $15 \leq n \leq 20$ the reduction factor remains close to 10.7.

6 Conclusions and Future Work

In this paper we presented an approach for ensuring scalability of service compositions (focusing on orchestrations with centralized control flow) – with rich control structure (involving branches, loops and parallel flows), state and data operations – by translating them in a network of actor behaviors which behaves correctly with respect to the semantics of the composition specification. Such a network can be instantiated and automatically scaled up/out by the underlying actor platform (in this case Akka on JVM) whose remoting and clustering capabilities facilitate deployment in the Cloud.

The experimental results indicate that using this approach the composition can be easily scaled (in this case vertically) to match the elasticity of the external services and to yield significant performance improvements. We have also shown how the state of an executing composition instance can be monitored and pushed to a persistent store in a non-blocking manner to allow for restoring and continuing a stopped instance. The same monitoring mechanism can be used for testing the composition on a fine-grained level against pre- and post-conditions on the composition as a whole and individual constructs from which it is built, and for computing code and path coverage of a test suite. Additionally, the scheme can be easily adapted for simulation of service behavior against different load scenarios.

There are several directions for expanding on the work presented in this paper. The authors are currently working on expanding the prototype implementation into a system which allows drop-in of composition definitions and their compilation into actor specifications, which is parametric with respect to the syntax of the composition language. Our plan is to support not only orchestration languages based on the procedural, but also on logical and functional programming paradigms. This can be followed with an elaboration of a testing framework that integrates automatic generation of test cases and performance analysis. Another direction would be creating, on the common basis, of an offline simulation platform that can be complemented with online data to provide forecasts of system performance under different load scenarios. Additionally, the underlying

actor formalism into which compositions are translated can be used for reasoning about safety and liveness properties of choreographies involving several orchestrations.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: Yet Another Workflow Language. Information Systems 30(4), 245–275 (2005)
2. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
3. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming 7(1), 1–72 (1997)
4. et al., M.D.D.: The reactive manifesto. Web (September 2013), http://www.reactivemanifesto.org/
5. Apache Software Foundation: Apache ODE Documentation (2013), https://ode.apache.org/
6. Apt, K.R., De Boer, F.S., Olderog, E.R.: Verification of sequential and concurrent programs. Springer (2010)
7. Bailis, P., Ghodsi, A.: Eventual consistency today: Limitations, extensions, and beyond. Commun. ACM 56(5), 55–63 (2013), http://doi.acm.org/10.1145/2447976.2447992
8. Bonetta, D., Pautasso, C.: An architectural style for liquid web services. In: 2011 9th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 232–241 (June 2011)
9. Cardelli, L., Gordon, A.D.: Mobile ambients. Theoretical Computer Science 240(1), 177–213 (2000)
10. Fournet, C., Gonthier, G.: The join calculus: A language for distributed mobile programming. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 268–332. Springer, Heidelberg (2002)
11. Gupta, M.: Akka Essentials. Packt Publishing Ltd. (2012)
12. Hewitt, C.: A universal, modular actor formalism for artificial intelligence. In: IJCAI 1973. IJCAI (1973)
13. Hewitt, C.: Viewing control structures as patterns of passing messages. Artificial Intelligence 8(3), 323–364 (1977)
14. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10) (1969)
15. Ibsen, C., Anstey, J.: Camel in Action, 1st edn. Manning Publications Co., Greenwich (2010)
16. Milner, R.: Communicating and mobile systems: The pi calculus. Cambridge University Press (1999)
17. Team, O.: Orchestra User Guide. Bull-SAS OW2 Consortium (October 2011), http://orchestra.ow2.org/
18. Varela, C.A.: Programming Distributed Computing Systems: A Foundational Approach. MIT Press (2013)