Engineering Semantic Self-composition of Services Through Tuple-Based Coordination

Ashley Caselli[1](✉), Giovanni Ciatto[2](✉), Giovanna Di Marzo Serugendo[1](✉), and Andrea Omicini[2](✉)

[1] Centre Universitaire d'Informatique (CUI), University of Geneva, Geneva, Switzerland
{ashley.caselli,giovanna.dimarzo}@unige.ch

[2] Department of Computer Science and Engineering (DISI), Alma Mater Studiorum—Università di Bologna, Cesena, Italy
{giovanni.ciatto,andrea.omicini}@unibo.it

Abstract. Service self-composition is a well-understood research area focusing on service-based applications providing new services by automatically combining pre-existing ones. In this paper we focus on tuple-based coordination, and propose a solution leveraging logic tuples and tuple spaces to support semantic self-composition for services. A full-stack description of the solution is provided, ranging from a theoretical formalisation to a technologically valuable design and implementation.

Keywords: Service self-composition · Semantic reasoning · Tuple-based coordination

1 Introduction

Nowadays an ever increasing number of IT scenarios leverages a services-based architecture. These sorts of systems are modelled as a collection of heterogeneous and loosely-coupled fine-grained processes, namely *services*, that communicate among them. Arguably, the pervasive adoption of services-based architectures will lead to an explosion in the number of services populating the Internet. In other words, *scalability* issues are going to arise soon.

On the other hand, novel business opportunities are likely to become available as the amount of services increases. In fact, the public availability of disparate services is commonly a key enabler for the creation of secondary services built on top of the pre-existing ones. To this end, effective techniques – such as *service* composition – are required at the technical level, in order to reuse the available functionalities. However, service composition sets many challenges from a system administration perspective. The experience of developers, as well as their careful work, is a necessary prerequisite for composition of services to be effective. Unfortunately, the effectiveness of human experts in tackling an increasing number of services does not scale up linearly with the total amount of services.

To deal with these issues, a viable solution may be represented by automatically handling the composition. To this end, approaches focused on the composition of the existing services have been proposed. The mechanism that combines two or more basic services into a more complex one is known as *service composition* [17]. It aims at creating higher-level functionalities within the system by leveraging the available resources.

The static nature of traditional approaches has been challenged by dynamic service composition approaches [7], which range over syntax-based composition to semantic-based composition and AI planning techniques. The adoption of such approaches paved the way to the design of systems with innate autonomous computational properties, such as *self-adaptation* and *self-composition*.

Many research works focus on coping with *"challenging problem of composing services dynamically"* [2]. Nevertheless, most of them solve it only partially: to the best of our knowledge, most of the existing solutions to the dynamic service composition challenge present limitations—e.g., syntax-based composition. Other approaches, although well-designed and sound at a conceptual level, are either discontinued or based on obsolete technologies [5].

This paper aims at providing a comprehensive tuple-based technology for semantic self-composition of services. A self-composition model that promotes and supports spontaneous service composition based on Linda [15] is proposed. The solution supports semantic reasoning leveraging on logic tuples and Linda tuple spaces. Moreover, a Java-based implementation of such model is also proposed, relying on the recent TuSoW [6] technology for tuple-based coordination.

The remainder of this paper is organised as follows. Section 2 provides an overview of the current approaches for service composition. Section 3 shows a formal definition of the designed system in terms of its syntax and operational semantics. The Java-based software architecture that implements the proposed technology is shown in Sect. 4. Section 5 presents a case study in a formal way. Finally, Sect. 6 concludes the paper by summarising the proposed solution.

2 State of the Art

2.1 Service Composition

Service composition is broadly known as the mechanism that combines two or more basic services into a more complex one that provides higher-level functionalities [17]. It deals with the needs of users to search for appropriate compositions of services that meet the required processes [27].

Service composition approaches may be categorised in terms of many orthogonal properties. A possible grouping considers the composition policy: *(i)* syntax-based: the matching among services is computed as mere equality operation on the input/output parameters of the services; *(ii)* semantic-based: it requires a taxonomy of concepts on which the composition process relies on to compute the matches; and *(iii)* through AI-planning solutions: it concerns the task of finding a course of action to reach a goal. From a different point of view, a composition process may be defined as the outcome of two minor phases – i.e., *selection* and *binding* –, hence a different grouping may be provided. A service composition approach may then be defined as *(i)* static, when the binding occurs at design-time; or *(ii)* dynamic, when the binding occurs either at deployment- or run-time. Using a static approach, the compositions are built during the design of the system (design-time), by the system designer that creates them once for all. This approach leads to correct compositions but lacks of scalability and adaptability. On the other hand, a dynamic approach ensures scalability and adaptability by adding computational overhead to the system. Dynamic approaches differ in the stage the binding phase occurs, which may be at *(i)* deployment-time, where the service binding phase occurs each time a service shows up in the system; or at *(ii)* run-time, where the binding occurs when a request is published.

Among these categories, we can mention the following works. From the semantic web domain, Talib et al. [25] provide a semi-automatic method to generate static web service composition in BPEL4WS language. Talantikite et al. [24] present a model for automatic Web services discovery and composition that exploits semantically annotated web services through an upper ontology (i.e. OWL-S [20]). In the field of ambient intelligence, Vallee et al. [26] propose an approach that combines multi-agent techniques with semantic web services to enable dynamic, context-aware service composition. In the field of multi-agent systems (MAS) approaches to self-composition usually involve planification, where agents reason on their respective services and the user's needs [14]. In this area, works on self-composition of method fragments bring a more dynamic solution based on cooperative agents, each representing a fragment and participating to the design of the fragments composition [3]. Using similar cooperative principles, Degas [9] proposes a syntax-based composition approach with collaborative agents for dynamic composition of aerial plane trajectories. Other approaches specifically involving chemical reactions for self-composition, possibly include the followings. Frei et al. [13] propose the use of chemical reactions, in the field of industrial robotics, to build self-organising assembly systems that participate in their own design by spontaneously organising themselves. Di Napoli et al. [11] show how a specified workflow can be instantiated using chemical reactions. In the context of tuple spaces, Viroli [27] proposes a syntax-based approach inspired by chemical reactions combined with the notion of competition among services. De Angelis [7] proposes a chemical-inspired model that promotes syntax-based self-composition of services at run-time. To alleviate the lack of semantics in the composition in [7], Ben Mahfoudh et al. [1] extend the original tuple space model with learning-based capabilities, thus providing pertinent and reliable services to the user.

2.2 Linda and TuSoW

Linda [15] is the archetypal tuple-based coordination model [22], inspiring and influencing a huge number of coordination models and technologies throughout the years [5]. The main elements of Linda are *tuples*, *templates*, *tuple spaces*, and *communication primitives*. A tuple is a piece of information represented according to a well-defined *tuple language*, specifying the structure of admissible tuples. A template is a concise way of representing a set of tuples: it consists of a pattern, represented according to a particular *template language*, which may be matched by several tuples. A tuple space is a repository where tuples may be inserted, observed, or withdrawn by an arbitrary number of agents willing to synchronise while being uncoupled in reference, space, and time. On purpose, a communication primitive is an operation provided to interacting agents to *synchronise* themselves upon tuples' insertion, observation, and consumption.

Linda is characterised by a few peculiar features: *(i) generative communication*, that is, tuples existing independently of the agents who produced them; *(ii) associative access*, namely, agents can access (i.e., observe or withdraw) the tuples stored in a tuple space by simply specifying a template, without the need of knowing the tuple "address" neither its "name"; and *(iii) suspensive semantics*, that is, agents' attempts of accessing a tuple matching a particular template are suspended until a tuple of such a sort actually exists.

Linda provides three communication primitives: `out` to insert a tuple in a tuple space, `in` to withdraw one, `rd` to read one. Despite their simplicity, such primitives are expressive enough to cope with several common interaction patterns [15]. Suspensive semantics, in particular, is the cornerstone of the coordination mechanism proposed by Linda, since it deals with synchronisation: whereas the `out` primitive always puts a tuple in the tuple space, `in` and `rd` *attempt* to get one based on a provided tuple template. If a tuple *matching* the template is found, it is returned to the caller agent that can continue execution; otherwise, the caller agent is *suspended* until a matching tuple becomes available.

Several variants of Linda have been proposed throughout the years, either extending the set of communication primitives, adding features such as mobility or access control [8,21], enabling distribution of multiple tuple spaces on a network of interconnected computers [12,19], and much more [23]. Nevertheless, only a few have been developed as a *technology* [5]—and, among these, some have already been exploited for service composition, as already discussed in the related works section above.

TuSoW [6] is tuple-based technology for coordination for distributed agents via Linda tuple spaces. It aims at providing a lightweight, modular, flexible, and highly interoperable implementation of Linda. It is designed as a multi-platform technology, making it suitable to be used by a wide community of developers in a wide range of application domains. In particular, TuSoW coordination facilities are provided to agents *as-a-Service*, via the HTTP protocol. For this reason we chose it as reference technology in the remainder of this paper.

3 Formal Model

The proposed model formalises a system composed by a number of active entities – namely, *agents* – acting as either service requesters (a.k.a. clients, or users), or service providers (a.k.a. servers). Users and servers do not interact directly but rather they interact by means of a Linda-like shared memory – that is, the *blackboard* –, acting as a coordination medium.

The interaction among users and servers is based on a simple protocol. On the one side, servers advertise their service descriptors by publishing them on the blackboard, upon startup. After that, they keep listening for incoming requests issued by users. As soon as a request is issued by some user, if a server exists which is capable of serving that request, then it is triggered. The invoked server must then execute its service, producing a result which is eventually output on the blackboard as well. On the other side, users are simple agents which may, from time to time, issue requests towards a particular service descriptor. When this happens, the user must then wait for a result to eventually appear on the blackboard, and finally consumes it before terminating.

Automatic semantic composition of services is provided by the blackboard using a dynamic deployment time approach [18]. In other words, whenever a novel service descriptor is published on the blackboard, the blackboard reacts by generating and automatically inserting a (possibly null) amount of *composite* service descriptors on it-self. In particular, the set of service descriptors to be generated is computed by combining the just-inserted one with *all* the service descriptors it may combine with, among the many already present on the blackboard.

Of course neither users nor clients are aware of the service composition performed by the blackboard. In other words, the service composition is transparent to both users and servers. To make this possible, the blackboard is in charge of *splitting* users' requests directed towards composite service descriptors into elementary request, which may then be served by servers. For the same reason, the blackboard is also in charge of handling the intermediary results possibly produced by servers when a composite service request is being served.

In the next sections we formalise such insights by means of process algebra. In particular, we first structurally define the most relevant notions of our model by means of an EBNF grammar, and then provide its semantics by means of a Labeled Transition System [16].

3.1 Syntax

Here we provide a syntax for the main concepts composing our model. To do so, we exploit EBNF grammars.

System. We define a system (Sys) as a parallel composition of one or more *agents* and a *blackboard* (B). In turn, each agent may be either a *user agent* (U) or *service agent* (S), according to their role in the system. Formally:

$$\begin{array}{lr} Sys ::= S_S \parallel U_S \parallel B & \text{main system} \\ S_S ::= S \mid (S \parallel S_S) & \text{list of services} \\ U_S ::= U \mid (U \parallel U_S) & \text{list of users} \end{array}$$

where $\parallel$ is the parallel composition operator—commutative and associative.

Blackboard. A blackboard is modelled as the space where the interaction among agents takes place. It is exploited as coordination medium by the agents, which may perform basic read/write operations on it. We define a blackboard (B) as a multiset that may either be empty or contain four sorts of data: *(i)* service descriptors, *(ii)* user requests, *(iii)* internal messages, or *(iv)* results. Formally:

$$B ::= \emptyset \mid SD \mid Req \mid \texttt{serve}(SD, C) \mid \texttt{serve\_comp}(SD, C) \mid Res \mid B \cup B$$

where $\cup$ is the union operator for multisets – associative and commutative –, whereas $\emptyset$ denotes the empty multiset.

Service. A service represents a *service agent*. It is capable of two operations embodied by *publish* and *accept*, which are grammar syntactic sugar. Intuitively, *publish* denotes the operation used by a service to advertise itself on the blackboard; *accept* says that the service is listening for incoming requests. Formally:

$$S ::= \texttt{publish}(SD) \mid \texttt{accept}(\texttt{service}(Q)) \mid S \cdot S$$

where $\cdot$ is the sequence operator—associative and not commutative.

User. A user represents a *user agent*. Similarly to a service agent, it is capable of two operations, represented through the *Req* and *Res* terms. They embody a request and a response message, respectively. At last, the `halt` term is used to represent the eventual termination event. Formally:

$$U ::= Req \cdot Res \cdot U \mid \texttt{halt}$$

where $\cdot$ operator is equivalent to the one defined above. By construction, well-formed users must wait for a response event after each request event.

Service Descriptor. A service descriptor (SD) provides the representation of a service. Thus, a service descriptor may either represent: *(i) an atomic service* – through its formal arguments: the (possibly empty) set of the *named input types* (I) it is able to accept and the *output type* (O) it produces as result –, or *(ii) a composed service*, as the concatenation of two services in such a way that the output of the first one is provided as input to the second one. Formally:

$$\begin{aligned}
SD &::= \texttt{service}(Q) \mid SD \overset{N}{\texttt{argof}} SD && \text{service descriptor}\\
Q &::= I,\ O && \text{query}\\
I &::= \epsilon \mid N : T \mid I, I && \text{input}\\
O &::= \epsilon \mid T && \text{output}\\
N &::= n_1 \mid n_2 \mid n_3 \mid \ldots && \text{name}\\
T &::= t_1 \mid t_2 \mid t_3 \mid \ldots && \text{type}
\end{aligned}$$

Request/Response. Agents may append *request* (*Req*) and *response* (*Res*) messages to the blackboard. A request message is defined as either *(i) query* (Q), or *(ii) call* (C). A query expresses an exploratory request, aimed at checking whether the system is capable of serving a particular signature or not, given the currently published services and their compositions. Conversely, a call represents an actual invocation of some service, which may involve the execution of one or more agents to serve the request. Requests are represented through their actual *input arguments* (A) – which are named as well – and the expected *output type* (O) they ask for. On the other side, response messages may instead contain a *(i) Const* term, which is a boolean value, or a *(ii) value* (V), that allows any kind of terminal value to be represented. Formally:

$$
\begin{aligned}
\mathit{Req} &::= \texttt{query}(Q) \mid \texttt{call}(C) && \text{request}\\
C &::= A,\ O && \text{call}\\
A &::= \epsilon \mid N : T(V) \mid A, A && \text{arguments}\\
V &::= v_1, v_2, \ldots, v_n && \text{terminal values}\\
\mathit{Res} &::= \texttt{res}(\mathit{Const}) \mid \texttt{res}(V) && \text{response}\\
\mathit{Const} &::= \top \mid \bot && \text{boolean value}
\end{aligned}
$$

3.2 Operational Semantics

A Labelled Transition System (LTS) is exploited to provide the operational semantics of our model. The transition relations model the effect of executing an action on the blackboard.

Labels. Labels are used in the LTS to formally capture events of interest for the operational semantics of our model. In order to ease their comprehension, all label names are suffixed by the name of the transition rules they are involved into. Only one exception is made for τ, denoting the silent transition.

$$
\begin{aligned}
E ::= {} & \mathit{publish\_sd} \mid \mathit{publish\_query} \mid \mathit{publish\_call} \mid \mathit{consume\_call} \mid\\
& \mathit{consume\_comp\_call} \mid \mathit{serve\_call} \mid \mathit{comp\_call} \mid \mathit{serve\_comp\_call} \mid\\
& \mathit{last\_comp\_call} \mid \mathit{prove} \mid \mathit{compose} \mid \tau
\end{aligned}
$$

Operators. A definition of functions and operators exploited within the transition rules is following. For the sake of brevity we only provide an intuition of each. An exhaustive formal definition of their semantics can be found in [4]. Notice that, in what follows, we often leverage the notation $\mathcal{L}(X)$, where X is some non-terminal symbol among the many defined in the EBNF production rules above. There, we write $\mathcal{L}(X)$ meaning "the set of all possible strings produced by all possible production rules for X".

- The function *typeof* : $\mathcal{L}(C) \rightarrow \mathcal{L}(SD)$ retrieves the data type of a call request and encodes it under the form of a service descriptor.
- The *match* operator $\sim \subseteq \mathcal{L}(SD) \times \mathcal{L}(SD)$ evaluates the matching degree among two service descriptors through semantic reasoning.

  The function *execute* : $\mathcal{L}(S) \times \mathcal{L}(Req) \rightarrow \mathcal{L}(V)$ triggers the service execution in order to fulfill the provided request and it subsequently provides the result.
- The function *prove* : $\mathcal{L}(Req) \times \mathcal{L}(SD) \rightarrow \mathcal{L}(Const)$ performs the evaluation of a query request.
- The function *fringe* : $\mathcal{L}(SD) \rightarrow \mathcal{L}(I)$ is in charge of retrieving a set containing the inputs of a compound service descriptor, namely its fringe.
- The function *compose* : $\mathcal{L}(SD) \times \mathcal{L}(SD) \rightarrow \mathcal{L}(SD)$ designs the binding among services, creating one or more new service descriptors which represent the composed service.
- Finally, the function *compositions* : $\mathcal{L}(B) \times \mathcal{L}(SD) \rightarrow \mathcal{L}(SD)$ aims to identify all the compositions in which a given service descriptor is involved.

Transition Rules. Transition rules define the admissible actions for a system compliant with our model. In a nutshell, admissible actions include: *(i)* publishing a service descriptor on the blackboard, *(ii)* composing two or more services, *(iii)* publishing a request message (call or query) on the blackboard, *(iv)* proving a query request, *(v)* serving a call request, and *(vi)* the decay of a service descriptor. The formal definition of the corresponding transition rules follows.

Service Descriptor Publication. The service descriptor publication is governed by the [PUBLISH-SD] transition rule. The rule may occur anytime during the system life-cycle. Its execution changes the blackboard state, enriching it with the published service descriptor. Formally:

$$\texttt{publish}(SD) \cdot S \parallel S_S \parallel U_S \parallel B \xrightarrow{publish\_sd} S \parallel S_S \parallel U_S \parallel B \cup SD \quad \texttt{[PUBLISH-SD]}$$

Composition. The composition is governed by the [COMPOSE] transition rule. It triggers each time a service is published and evaluates if there exists a service that matches with the published one. If it is the case, a composed service descriptor is generated and published on the blackboard.

$$\frac{SD = \texttt{service}(I, O) \wedge \exists\ (N : O) \in fringe(SD') \wedge SD'' = compose(SD,\ SD')}{S_S \parallel U_S \parallel B \cup SD \cup SD' \xrightarrow{compose} S_S \parallel U_S \parallel B \cup SD \cup SD' \cup SD''} \quad \texttt{[COMPOSE]}$$

Request Publication. The request publication is governed by the [PUBLISH-QUERY] and [PUBLISH-CALL] transition rules. Their execution publishes a query or call message, respectively, on the blackboard. They both may occur anytime during the system life-cycle.

$$\texttt{query}(Q) \cdot U \parallel S_S \parallel U_S \parallel B \xrightarrow{publish\_query} U \parallel S_S \parallel U_S \parallel B \cup \texttt{query}(Q) \qquad \texttt{[PUBLISH-QUERY]}$$

$$\texttt{call}(C) \cdot U \parallel S_S \parallel U_S \parallel B \xrightarrow{publish\_call} U \parallel S_S \parallel U_S \parallel B \cup \texttt{call}(C) \qquad \texttt{[PUBLISH-CALL]}$$

Proving. The result of a query request is generated by either the [POS-PROVE] or the [NEG-PROVE] transition rules. The former (resp. latter) is triggered when *(i)* there exists at least one service descriptor (either single or composed) on the blackboard that is able (resp. unable) to fulfill the current query, *(ii)* there exists a user waiting to consume the positive (resp. negative) result. Once triggered, each transition allows the waiting user to go on with its computation.

$$\frac{\texttt{service}(Q) \sim SD \wedge Const = prove(Q, SD)}{S_S \parallel \texttt{res}(\top) \cdot U \parallel U_S \parallel B \cup SD \cup \texttt{query}(Q) \xrightarrow{prove} S_S \parallel U \parallel U_S \parallel B \cup SD} \qquad \texttt{[POS-PROVE]}$$

$$\frac{\nexists\ SD \in B : \texttt{service}(Q) \sim SD}{S_S \parallel \texttt{res}(\bot) \cdot U \parallel U_S \parallel B \cup \texttt{query}(Q) \xrightarrow{prove} S_S \parallel U \parallel U_S \parallel B} \qquad \texttt{[NEG-PROVE]}$$

Serving. The management of a call request is governed by [CONSUME-CALL], [SERVE-CALL], [COMP-CALL], [CONSUME-COMP-CALL], [SERVE-COMP-CALL], and [LAST-COMP-CALL] transition rules.

The [CONSUME-CALL] rule is atomic: it is triggered each time a call request can be fulfilled by some simple service. The rule is triggered only if a simple service *SD* is listening for incoming requests. Once triggered, the rule consumes the call request and adds an internal call message `serve` to the blackboard.

$$\frac{SD = \texttt{service}(I, O) \wedge typeof(\texttt{call}(C)) \sim SD}{\texttt{accept}(SD) \cdot S \parallel S_S \parallel U_S \parallel B \cup SD \cup \texttt{call}(C) \xrightarrow{consume\_call} \texttt{accept}(SD) \cdot S \parallel S_S \parallel U_S \parallel B \cup SD \cup \texttt{serve}(SD, \texttt{call}(C))} \qquad \texttt{[CONSUME-CALL]}$$

The [SERVE-CALL] transition governs the serving of a call request. The rule is triggered only if *(i)* a simple service *SD* is listening for incoming requests, *(ii)* a user is waiting for a result, and *(iii)* an internal message `serve` generated from a call published by the same user is present on the blackboard. The transition allows both the waiting user and the service to go on with their computations, while the pending internal message `serve` is removed from the blackboard.

$$\frac{SD = \texttt{service}(I, O) \wedge typeof(\texttt{call}(C)) \sim SD \wedge V = execute(\texttt{accept}(SD), \texttt{call}(C))}{\texttt{accept}(SD) \cdot S \parallel S_S \parallel \texttt{res}(V) \cdot U \parallel U_S \parallel B \cup SD \cup \texttt{serve}(SD, \texttt{call}(C)) \xrightarrow{serve\_call} S \parallel S_S \parallel U \parallel U_S \parallel B \cup SD} \qquad \texttt{[SERVE-CALL]}$$

The [COMP-CALL] rule governs the serving of a call request by a composed service. The rule is triggered only if a composed service *SD* able to fulfil the published call request is present on the blackboard. During its execution, the blackboard state is modified and enriched with an internal call message `serve_comp` that contains the service descriptor *SD* of the composed service that is capable of serving the request, in addition to the original call request $\texttt{call}(C)$.

$$\frac{SD = SD' \overset{N}{\texttt{argof}} SD'' \wedge \mathit{typeof}(\texttt{call}(C)) \sim SD}{S_S \parallel U_S \parallel B \cup SD \cup \texttt{call}(C) \xrightarrow{\mathit{comp\_call}} S_S \parallel U_S \parallel B \cup SD \cup \texttt{serve\_comp}(SD, \texttt{call}(C))} \quad \texttt{[COMP-CALL]}$$

The `[CONSUME-COMP-CALL]` rule is in charge of initiating the chain of services executions that leads to the fulfilment of a call request with a composed service. The rule is triggered whenever a message `serve_comp` is published on the blackboard. Once triggered, this transition modifies the blackboard state, adding an internal message `serve` containing *(i)* the first service descriptor *SD* of the composition, and *(ii)* the portion of the call request that is fulfillable by the service described via the service descriptor *SD*.

$$\frac{SD = SD' \overset{N}{\texttt{argof}} SD'' \wedge \mathit{typeof}(\texttt{call}(C')) \sim SD'}{S_S \parallel U_S \parallel B \cup SD \cup \texttt{serve\_comp}(SD, \texttt{call}(C)) \xrightarrow{\mathit{consume\_comp\_call}} S_S \parallel U_S \parallel B \cup SD \cup \texttt{serve}(SD', \texttt{call}(C'))} \quad \texttt{[CONSUME-COMP-CALL]}$$

The `[SERVE-COMP-CALL]` rule is in charge of carrying on the execution of fulfilment of a call request using a composed service. It requires an internal message `serve` to be present. Once triggered, it generates a new internal message `serve` that contains *(i)* the service descriptor of the following service to be executed in the composition, and *(ii)* a new call with the result of the previous execution added as input parameter.

$$\frac{SD = SD' \overset{N}{\texttt{argof}} SD'' \wedge \mathit{typeof}(\texttt{call}(C')) \sim SD' \wedge V = \mathit{execute}(\texttt{accept}(SD'), \texttt{call}(C'))}{\texttt{accept}(SD') \cdot S \parallel S_S \parallel U_S \parallel B \cup SD' \cup \texttt{serve}(SD', \texttt{call}(C')) \xrightarrow{\mathit{serve\_comp\_call}} S \parallel S_S \parallel U_S \parallel B \cup SD' \cup \texttt{serve}(SD'', \texttt{call}(N : T(V), C'))} \quad \texttt{[SERVE-COMP-CALL]}$$

The `[LAST-COMP-CALL]` rule concludes the computational chain. It handles the last service execution providing the final result. Therefore, the user that published the call may consume the result and go on with its computation.

$$\frac{SD = SD' \overset{N}{\texttt{argof}} SD'' \wedge \mathit{typeof}(\texttt{call}(C'')) \sim SD'' \wedge V = \mathit{execute}(\texttt{accept}(SD''), \texttt{call}(C''))}{\texttt{accept}(SD'') \cdot S \parallel S_S \parallel \texttt{res}(V) \cdot U \parallel U_S \parallel B \cup SD'' \cup \texttt{serve}(SD'', \texttt{call}(C'')) \xrightarrow{\mathit{last\_comp\_call}} S \parallel U \parallel S_S \parallel U_S \parallel B \cup SD''} \quad \texttt{[LAST-COMP-CALL]}$$

Decay. The `[DECAY]` rule is defined with the purpose of keeping the *blackboard* (B) clean over the time.

$$\frac{B' = B - \mathit{compositions}(B, SD)}{S_S \parallel U_S \parallel B \cup SD \xrightarrow{\tau} S_S \parallel U_S \parallel B'} \quad \texttt{[DECAY]}$$

This rule grants the system the capability of cleaning out the blackboard from obsolete services. The operation also requires to clean out the composed services in which the service targeted to be removed is involved. Label τ is used here to denote a time-related recurrent operation. No specific frequency or rate is defined by our formal specification. Yet, we assume `[DECAY]` executes frequently enough to clean up stale service descriptors, but not so much frequently to hinder the activity of services.

4 Architecture

This section discusses how a rigorously engineered solution for semantic self-composition of services based on our model can be attained. In particular, because of space limitations, our discussion is articulated in two parts, describing the design and implementation phases of our solution, respectively. More precisely, in the first part we show how a software architecture for our model can be constructed by leveraging the Linda coordination model; whereas in the second part we show how such a software architecture can be reified into some actual JVM technology via the TuSoW framework.

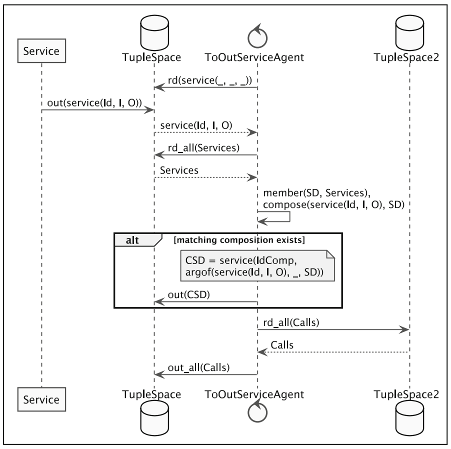
4.1 Linda-Based Architecture

A Linda system is composed by a number of agents interacting via tuple spaces. Our formal model as well can be briefly described in terms of agents interacting via blackboard, enacting a particular protocol. Thus, drawing a software architecture based on Linda for our framework essentially requires *(i)* the blackboard behaviour to be mimicked via some tuple space, and *(ii)* users and service agents to be designed as agents performing Linda operations on that tuple space.

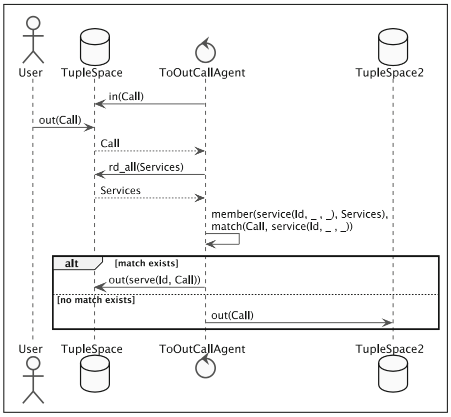
We stick to a logic-based interpretation of Linda, where both tuples and templates are first-order logic terms, and tuples are matched against templates via logic unification. Furthermore, we assume a wide spectrum of Linda primitives are available for agents, including *(i)* Linda's classic primitives – namely, `out`, `in`, `rd` –, with their ordinary generative and suspensive semantics; *(ii)* bulk primitives – such as `out_all`, `in_all`, `rd_all` –, letting agents insert, consume, or read multiple tuples at once; and *(iii)* predicative primitives – such as `inp`, `rdp` –, which differ from their classic counterparts because they are not suspensive.

Of course, given that the blackboard abstraction in our model is not a simple container of information – as it is in charge of automatically composing services as soon as they are deployed –, it cannot be simply reduced to a tuple space. To tackle this issue, at the architectural level, we introduce the notion of *helper agent*. An helper agent is a reactive entity which is in charge of implementing some transition rule from the model semantics described in Sect. 3.2. In other words, we translate each transition rule from Sect. 3.2 into an helper agent implementing it on the blackboard via Linda operations. Thus, there exists a fixed number of helper agents, whose names and functions are described below. For the sake of readability, helper agents are named using the pattern

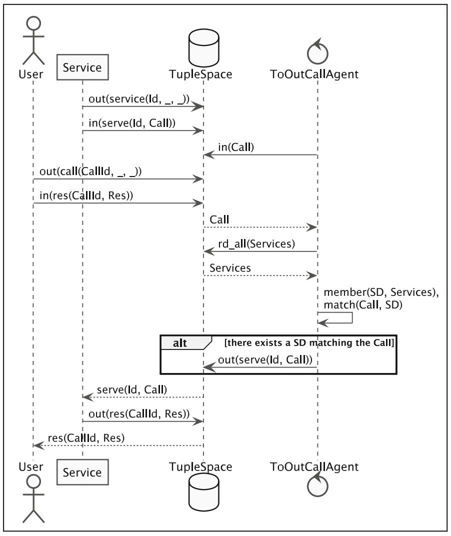
$$To\{EventName\}\{MessageName\}Agent$$



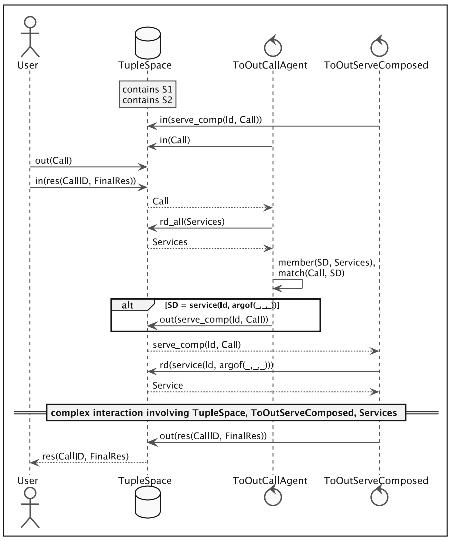
(a) Reaction to service publication in the tuple space



(b) Handling a call request that cannot be served



(c) Serving a call request with a single service


User
TupleSpace
ToOutCallAgent
ToOutServeComposed
contains S1 contains S2
in(serve_comp(Id, Call))
in(Call)
out(Call)
in(res(CallID, FinalRes))
Call
rd_all(Services)
Services
member(SD, Services), match(Call, SD)
alt [SD = service(Id, argof(_,_,_))]
out(serve_comp(Id, Call))
serve_comp(Id, Call)
rd(service(Id, argof(_,_,_)))
Service
complex interaction involving TupleSpace, ToOutServeComposed, Services
out(res(CallID, FinalRes))
res(CallID, FinalRes)

(d) Serving a call request with a composed service

Fig. 1. An overview of the most salient interactions among the system components during the publication, composition, and request serving phases

where {*EventName*} denotes the invocation of some LINDA operation on the blackboard tuple spaces – commonly, an `out` operation –, whereas {*MessageName*} is the tuple or template characterising that LINDA operation.

Accordingly, in the following we present a semi-formal definition of the LINDA-based architecture of our model via UML sequence diagrams. User agents publish the requests on the tuple space by means of the `out` primitive. Subsequently, they perform an `in` operation, waiting for a tuple to consume. Service agents, likewise, follow the same pattern of interactions. They publish their service descriptor and they consequently wait for tuples to be consumed.

Service Descriptor Publication and Service Composition. The transition rule `[COMPOSE]` has been implemented within the *ToOutServiceAgent* component. It reacts to the service publication action (`[PUBLISH-SD]`), evaluating all its viable compositions. If any, the composed service is generated and published on the *TupleSpace*. Figure 1a shows the full chain of interactions starting from the single service descriptor publication action to the subsequent composition evaluation and potential publication. Note that after a service descriptor is published, a list of unhandled call requests stored in a secondary tuple space is published on the primary tuple space. A more detailed description is provided in the following paragraphs.

Prove a Query Request. Operations `[POS-PROVE]` and `[NEG-PROVE]` are implemented by the *ToOutQueryAgent* component. It reacts to the publication action (`[PUBLISH-QUERY]`) of a query message, evaluating if there is an existing service configuration able to fulfil it: a positive result is returned *iff* any exists.

Serve a Call Request. Operations `[CONSUME-CALL]` and `[SERVE-CALL]` are implemented by the *ToOutCallAgent*. It reacts to the publication of a call request and evaluates if the current system configuration is capable of serving it—i.e. if there exists some service descriptor for the request at hand. Figure 1b shows the actions performed when a published call request cannot be fulfilled by any available service. Briefly, the matching among the call request and the available service is computed. If there exists no service that successfully matches the call, it is moved to another (secondary) tuple space, which is explicitly aimed at storing pending call requests which cannot be currently served. These calls are eventually moved back to the (primary) *TupleSpace* as soon as a service publication occurs—as the new service may make it possible to serve some of them. The involvement of two tuple spaces is an optimisation aimed at avoiding the waste of computational resources due to the processing of (currently) unsatisfiable calls.

Conversely, when the current system configuration allows the fulfillment of the call request, the request message is taken and processed. Figure 1c shows the serving of a call request in case it exists a single service that may wholly fulfil it. The opposite case is presented in Fig. 1d. In this case the rule `[COMP-CALL]`, implemented by *ToOutCallAgent*, occurs; while operations

`[CONSUME-COMP-CALL]`, `[SERVE-COMP-CALL]` and `[LAST-COMP-CALL]` are performed within the *ToOutServeComposedAgent* control flow.

4.2 Implementation Details

The aforementioned Linda-based architecture is implemented upon TuSoW. Briefly, the elements composing the system are *(i)* the Linda-like tuple space, i.e. blackboard, *(ii)* a number of agents, and *(iii)* a fixed number of helper agents.

TuSoW defines the Linda-like tuple space as the so-called `LogicSpace` architectural entity, representing an abstract version of an actual tuple space that can be provided in several versions—e.g.. local, remote, inspectable. TuSoW agents are implemented as simple control flows—i.e. threads. We implement the user and service agent entities as threads that communicate among them through the shared `LogicSpace`. Helper agents, in turn, are implemented as threads augmented with a `tu`Prolog engine [10]. In particular, they hold reasoning capabilities exploited within the system to evaluate *(i)* the viable service compositions, and *(ii)* the match degree between a request message and a service descriptor.

Adopting TuSoW makes handling the non-determinism of Linda read and consume operations challenging. In order to cope with it, the inspectable version of the `LogicSpace` comes to our aid, since it presents an inspectable interface, allowing tuple space state to be observed. To clarify how the feature is exploited within our implementation, an example is provided. An helper agent constantly consumes tuples matching a tuple template. For instance, the *ToOutServiceAgent* consumes tuples unifying with a tuple template that resembles a service descriptor, in order to react to a service descriptor publication. However, when many service descriptors coexist in the tuple space, such operation consumes one of them in a non-deterministic manner. Therefore it might return any service that is currently published. To cope with it, the inspectable feature of the tuple space is exploited by filtering out the tuples that do not belong to the tuple space internal writing event. In other words, a routine is bound to the internal writing event of the tuple space, filtering out the tuples resulting from the writing event that do not comply with the provided tuple template.

5 Case Study

A real-world scenario is here provided. Due to space reasons, we only show its formal representation. The corresponding implementation leveraging a TuSoW-based system architecture is publicly available[1].

Let us assume that there exists a system holding a knowledge base composed of the taxonomy of concepts depicted in Fig. 2. Let us now consider the system as including two services willing to advertise themselves by publishing their service descriptors, respectively SD and SD', on the blackboard (B). We assume the formal parameters (input and output) of those services are defined using concepts

[1] https://gitlab.com/ashleycaselli/tusow-semantic-composition.

that belong to the knowledge base of the system. In particular, we define SD as the service that given a city name is able to provide its GPS coordinates. In turn, we define SD' as the service that provides the current temperature (in Kelvin degrees) at the location described by some GPS coordinates.

Formally, service descriptors are described as follows:

$$SD = \texttt{service}(name : City,\ GPS)$$
$$SD' = \texttt{service}(loc : GPS,\ Kelvin)$$

(for the sake of simplicity, we define GPS coordinates as a single value uniquely identifying a city), whereas the service initial configurations are as follows:

$$S_0 = \texttt{publish}(SD) \cdot \texttt{accept}(\texttt{service}(Q)) \cdot S_0$$
$$S_0' = \texttt{publish}(SD') \cdot \texttt{accept}(\texttt{service}(Q')) \cdot S_0'$$

We also assume the blackboard is initially empty ($B_0 = \emptyset$), and that the system includes a user willing to perform a service invocation:

$$U_0 = Req \cdot \texttt{res}(v) \cdot \texttt{halt}$$

where $Req = \texttt{call}(name : City(Geneva),\ Temperature)$ denotes an invocation to a service computing the current temperature for a city (namely, Geneva), and returning a temperature through any possible measurement unit. Under these hypotheses, the initial state of the system is $Sys_0 = S_0 \parallel S_0' \parallel U_0 \parallel B_0$.

The publication of the service descriptors (operation [PUBLISH-SD]) changes the state of the system as follows:

$$Sys_1 = \underbrace{\texttt{accept}(\texttt{service}(Q)) \cdot S_0}_{S_1} \parallel \overbrace{\texttt{accept}(\texttt{service}(Q')) \cdot S_0'}^{S_1'} \parallel U_0 \parallel \overbrace{SD \cup SD'}^{B_1}$$

Eventually, their publication triggers the component that computes the semantic matching among the two service descriptors, computing all the possible compositions (operation [COMPOSE]). In particular, in this case the compose operation detects that the services represented by SD and SD' are composable w.r.t the parameter named loc. We call $\widehat{SD} = SD \overset{loc}{\texttt{argof}} SD'$ the composed

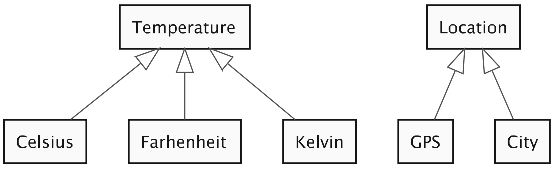


Fig. 2. An illustration of a taxonomy of concepts used in the presented case study

service attained by composing SD and SD'. The composed service $\widehat{SD}$ is then published on the blackboard, which can now be described as follows:

$$B_2 = SD \cup SD' \cup \widehat{SD}$$

The presence of $\widehat{SD}$ on the blackboard is what makes the user's invocation satisfiable. Suppose now that the user publishes (operation [PUBLISH-CALL]) its call request (Req). This would lead to a system state like the following:

$$Sys_3 = S_1 \parallel S'_1 \parallel U_0 \parallel \underbrace{SD \cup SD' \cup \widehat{SD} \cup Req}_{B_3}$$

According to the current system configuration (Sys_3) there is no simple service capable of serving the request. However, the request may be fulfilled using the composed service $\widehat{SD}$. In more details, $\widehat{SD}$ and Req are compatible because *(i)* the input ($I_{\widehat{SD}}$) of the composed service $\widehat{SD}$ and the input of the request (I_{Req}) hold the *exact* match degree, and *(ii)* the output ($O_{\widehat{SD}}$) of the composed service $\widehat{SD}$ and the output of the request (O_{Req}) hold the *subsume* match degree according to the provided taxonomy. Formally:

$$I_{\widehat{SD}} \equiv I_{Req} \wedge O_{\widehat{SD}} \sqsubseteq O_{Req}$$

The call request publication triggers the helper agent that is in charge of handling the request message. Such component, leveraging a Prolog engine for reasoning purposes, computes the semantic matching among the request and the available services. In this case, the reasoning process leads to the solution proposed above, inferring that the request may only be served by the composed service $\widehat{SD}$. In order to manage the execution of all the services involved in the composition, another helper agent is triggered (operation [COMP-CALL] is executed). Formally:

$$B_4 = SD \cup SD' \cup \widehat{SD} \cup \texttt{serve\_comp}(\widehat{SD}, \texttt{call}(...))$$

The helping agent is also in charge of collecting the intermediary responses that each service provides, and of providing the final response. Each time a service is triggered to serve the call, it computes the result and publishes it as Res message on the blackboard (operation [SERVE-COMP-CALL]). For the sake of brevity we only show one round of the "service execution-response publication" loop:

$$B_5 = SD \cup SD' \cup \widehat{SD} \cup \texttt{serve}(SD_x, \texttt{call}_x(...))$$

where SD_x and $\texttt{call}_x$ represent respectively the service descriptor of the x-th service of the composition and the call request that is served by such service.

Finally, operation [LAST-COMP-CALL] is executed and the user agent gets the result.

$$B_6 = SD \cup SD' \cup \widehat{SD}$$
$$U_6 = \texttt{halt}$$

6 Conclusion

This paper proposes a solution for the semantic self-composition of services, exploiting tuple-based coordination. We provide an end-to-end description of the engineering challenges hidden in the production of such sorts of systems, and sketch the formalisation of a middleware supporting *(i)* the self-composition of services, at deploy time, and *(ii)* the transparent invocation of the composed services from the client-side. In particular, we rely on a central blackboard used by service providers to advertise their own service descriptors, and in charge of orchestrating the execution of composed services. In this way, clients may invoke both composed and simple service through a uniform API.

Accordingly, the design of our solution is deliberately minimal as our focus is on the engineering of an actual implementation. In particular, the actual design of our middleware leverages *(i)* Linda-like tuple spaces exploiting logic terms as both clauses and templates, and *(ii)* logic programming to provide the system components with semantic reasoning. Finally, a prototype implementation is described exploiting the TuSoW coordination technology, and the tuProlog logic reasoner.

We consider this work as a starting point for a number of research directions. In fact, in the future, we plan to assess different strategies for implementing our model, from both the theoretical and technological perspectives. For instance, we are planning the exploitation of different matching mechanisms – possibly modelling semantic matching as a similarity function rather than a binary relation –, as well as different interaction protocols for the helper agents used in our prototype—possibly focusing on the scalability of service composition.

Acknowledgements. The authors would like to thanks the anonymous reviewers for their valuable remarks.

This work has been partially supported by the H2020 Project "AI4EU" (G.A. 825619).

References

1. Ben Mahfoudh, H., Di Marzo Serugendo, G., Naja, N., Abdennadher, N.: Learning-based coordination model for spontaneous self-composition of reliable services in a distributed system. Int. J. Softw. Tools Technol. Transfer **22**(4), 417–436 (2020). https://doi.org/10.1007/s10009-020-00557-0
2. Benatallah, B., Dumas, M., Fauvet, M.C., Rabhi, F.A.: Towards patterns of web services composition. In: Rabhi, F.A., Gorlatch, S. (eds.) Patterns and Skeletons for Parallel and Distributed Computing, pp. 265–296. Springer, London (2003). https://doi.org/10.1007/978-1-4471-0097-3_10
3. Bonjean, N., Gleizes, M.P., Maurel, C., Migeon, F.: SCoRe: a self-organizing multi-agent system for decision making in dynamic software development processes. In: International Conference on Agents and Artificial Intelligence (ICAART) (2013). (short paper)
4. Caselli, A.: Logic-based coordination: a semantic approach to self-composition of services. Master's thesis, Alma Mater Studiorum-Università di Bologna, School of Engineering (2019). http://amslaurea.unibo.it/17984

5. Ciatto, G., Di Marzo Serugendo, G., Louvel, M., Mariani, S., Omicini, A., Zambonelli, F.: Twenty years of coordination technologies: COORDINATION contribution to the state of art. J. Log. Algebraic Methods Program. **113**, 1–25 (2020). https://doi.org/10.1016/j.jlamp.2020.100531
6. Ciatto, G., Rizzato, L., Omicini, A., Mariani, S.: TuSoW: tuple spaces for edge computing. In: The 28th International Conference on Computer Communications and Networks (ICCCN 2019), Valencia, Spain, 29 July–1 August 2019. IEEE (2019). https://doi.org/10.1109/ICCCN.2019.8846916
7. De Angelis, F.L.: A logic-based coordination middleware for self-organising systems: distributed reasoning based on many-valued logics. Ph.D. thesis, University of Geneva, School of Social Sciences - Information Systems (2017)
8. De Nicola, R., Ferrari, G.L., Pugliese, R.: KLAIM: a kernel language for agents interaction and mobility. IEEE Trans. Softw. Eng. **24**(5), 315–330 (1998). https://doi.org/10.1109/32.685256
9. Degas, A.: Auto-structuration de trafic temps-réel multi-objectif et multi-critère dans un monde virtuel. Ph.D. thesis, Université de Toulouse III - Paul Sabatier, IRIT - UMR 5505, Toulouse, France (2020)
10. Denti, E., Omicini, A., Ricci, A.: tuProlog: a light-weight prolog for internet applications and infrastructures. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 184–198. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45241-9_13
11. Di Napoli, C., Giordano, M., Németh, Z., Tonellotto, N.: Using chemical reactions to model service composition. In: 2nd International Workshop on Self-organizing Architectures (SOAR 2010), pp. 43–50. ACM, New York (2010). https://doi.org/10.1145/1809036.1809047
12. Freeman, E., Arnold, K., Hupfer, S.: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley Longman Ltd., Essex (1999)
13. Frei, R., Şerbănuţă, T.F., Di Marzo Serugendo, G.: Self-organising assembly systems formally specified in Maude. J. Ambient Intell. Humaniz. Comput. **5**(4), 491–510 (2012). https://doi.org/10.1007/s12652-012-0159-2
14. Gabillon, Y., Calvary, G., Fiorino, H.: Composing interactive systems by planning. In: 4th French-Speaking Conference on Mobility and Ubiquity Computing (UbiMob 2008), pp. 37–40. ACM, New York (2007). https://doi.org/10.1145/1376971.1376979
15. Gelernter, D.: Generative communication in Linda. ACM Trans. Program. Lang. Syst. **7**(1), 80–112 (1985). https://doi.org/10.1145/2363.2433
16. Gorrieri, R.: Labeled transition systems. Process Algebras for Petri Nets. MTCSAES, pp. 15–34. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-55559-1_2
17. Kalasapur, S., Kumar, M., Shirazi, B.A.: Dynamic service composition in pervasive computing. IEEE Trans. Parallel Distrib. Syst. **18**(7), 907–918 (2007). https://doi.org/10.1109/TPDS.2007.1039
18. Lemos, A.L., Daniel, F., Benatallah, B.: Web service composition: a survey of techniques and tools. ACM Comput. Surv. **48**(3), 1–41 (2015). https://doi.org/10.1145/2831270
19. Louvel, M., Pacull, F.: LINC: a compact yet powerful coordination environment. In: Kühn, E., Pugliese, R. (eds.) COORDINATION 2014. LNCS, vol. 8459, pp. 83–98. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-43376-8_6
20. Martin, D., et al.: OWL-S: Semantic markup for web services. W3C Member Submission 22 (2004)

21. Murphy, A.L., Picco, G.P., Roman, G.C.: LIME: a coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Softw. Eng. Methodol. (TOSEM) **15**(3), 279–328 (2006). https://doi.org/10.1145/1151695.1151698
22. Omicini, A.: On the semantics of tuple-based coordination models. In: 1999 ACM Symposium on Applied Computing (SAC 1999), 28 February–2 March 1999, pp. 175–182. ACM, New York (1999). https://doi.org/10.1145/298151.298229
23. Omicini, A., Zambonelli, F.: Coordination for Internet application development. Auton. Agent. Multi-Agent Syst. **2**(3), 251–269 (1999). https://doi.org/10.1023/A:1010060322135
24. Talantikite, H.N., Aissani, D., Boudjlida, N.: Semantic annotations for web services discovery and composition. Comput. Stand. Interfaces **31**(6), 1108–1117 (2009). https://doi.org/10.1016/j.csi.2008.09.041
25. Talib, M.A., Yang, Z.: Semi-automatic code generation of static web services composition. In: Student Conference on Engineering, Sciences and Technology, pp. 132–137. IEEE, January 2005. https://doi.org/10.1109/SCONES.2004.1564784
26. Vallée, M., Ramparany, F., Vercouter, L.: A multi-agent system for dynamic service composition in ambient intelligence environments. In: PERVASIVE 2005, Advances in Pervasive Computing, vol. 191, pp. 175–182. Austrian Comp. Soc. (OCG) (2005)
27. Viroli, M.: On competitive self-composition in pervasive services. Sci. Comput. Program. **78**(5), 556–568 (2013). https://doi.org/10.1016/j.scico.2012.10.002