ABSTRACT

This paper aims at pushing the clear relationship between software service composition and chemical dynamics a step forward. We developed a coordination model where services and clients are coordinated via a tuple space handling services as if they were interacting chemical substances: on the one hand, services get equipped with an “activity value” resembling chemical concentration and measuring their reactivity as imposed by the tuple space; on the other hand, services automatically compose via interaction ports resembling chemical bonding. The tuple space enacts a feedback loop that regulates and balances the activity level of (atomic or composite) services, decreasing it over time as in chemical decay, but reinforcing it each time the service is correctly used. This behaviour promotes service competition: losing (i.e. unused) services literally extinguish. Which services or service compositions survive competition is automatically decided solely based on resulting performance, i.e. the rate at which services are actually exploited.

Categories and Subject Descriptors
C.2.4 [Computer-Communication Networks]: Distributed Systems—Distributed applications

Keywords
Biochemical Tuple Space, Automated Service Composition

1. INTRODUCTION

Today and tomorrow’s computing scenarios are characterised by openness, dynamism and unpredictability: they increasingly call for tackling issues like self-adaptation, self-management, and toleration of short- and medium-term evolution without human intervention [21]. This trend is not only recognised as a motivation for several research fields – like pervasive computing, autonomic-computing, self-organising systems, and nature-inspired computing – but also influences traditional areas of software engineering, coordination models and languages being not an exception.

Several coordination models and mechanisms have been recently introduced along this idea: some of the most notable examples are reported in [11, 5, 4, 19, 18]. This paper develops on [18], where the notion of biochemical tuple space is introduced. In this model, tuples are associated with an activity/pertinency value (related to tuple weights in the probabilistic extension of LINDA introduced in [3]), which resembles chemical concentration and measures the extent to which the tuple can influence the coordination state—e.g., a tuple with low concentration would be rather inert, hence taking part in coordination with very low frequency. Chemical-like reactions, properly installed into the tuple space, evolve concentration of tuples over time exactly in the same way chemical substances would behave into a solution (i.e., following the reference stochastic model in [6]). Such reactions are meant to promote the exploitation of chemical patterns – either of natural or artificial chemistry – that can make interesting self-organisation properties emerge. A main benefit of this model is the ability of enacting a sort of “ecological” behaviour in coordinated systems [21]. Namely, chemical reactions can regulate the interactions of coordinated agents promoting their competition: those that lose such competition are extinguished, i.e. their ability of interacting is increasingly prevented by the coordinated system [18].

This paper shows how this coordination model can be refined to support a mechanism of “competitive self-composition” of services. A biochemical tuple space is used to coordinate the behaviour of an open set of service agents and client agents; it is programmed with chemical reactions that make services spontaneously compose: all the services (atomic or composite) compete with each other to serve requests – service activity level and matching are used by the tuple space to allocate requests. While all compositions can possibly take place, thanks to a positive/negative feedback only successful ones “survive” in the system, the others fading and being never actually selected. Technically, the interface of services is structured as a set of input/output ports, and composition is performed by linking matching pairs of input/output ports in two services.

The paper is organised as follows. Section 2 introduces the biochemical tuple space model. Section 3 describes the abstract idea underlying our proposal in terms of chemical reactions. Section 4 refines the language of chemical reactions and formalises the mechanisms of service composition and client-service interaction. Section 5 briefly describes an implementation based on TuCSoN, while Section 6 concludes, providing final remarks.
2. CHEMICAL TUPLE SPACES

The biochemical tuple space model presented in [18] is used here as ground for a new composition model for services—the reader should refer to this paper for further information about related works in the coordination context. Our discussion focuses on a simplified version of the model, abstracting away from details that are unnecessary in this paper: we stick to a single tuple space and neglect probabilities associated to tuple retrieval.

A tuple space is a repository of tuples (structured data chunks like records) that is used as a coordination medium provided to external “agents”: such agents coordinate their behaviour by accessing the tuple space through primitives \( \text{out}, \text{rd} \) and \( \text{in} \), used to insert, read, and remove a tuple, respectively. Operations \( \text{rd} \) and \( \text{in} \) can specify a tuple template (a tuple with wildcards in place of some of its arguments) and their execution is suspended until a matching tuple is found.

The basic idea of the proposed model is to attach to each tuple an integer value called “activity level”, which can be seen as a measure of the pertinency of the tuple and resembles chemical concentration—the higher it is, the more likely and frequently the tuple will influence system coordination. Activity level of tuples is dynamic (as pertinency typically is), and evolves using a chemical-like system coordination. Activity level \( n \) of tuples residing in the space; \( L \) is a chemical-like law (also called reaction) expressing transformation of tuple set \( \tau \) (reactants) into \( T \) (effects) with chemical rate \( r \); \( P \) defines processes that are sequences of actions, including a primitive to wait a delay time with markovian rate \( r \), as well as the set of primitive \( \text{LINDA} \) operations for inserting (out), removing (in) and reading (rd) tuples; finally, \( S \) is a system, which is modelled as a flat composition of processes, tuples, and laws. Note that all the above elements are considered as terms, e.g., \( \langle . \rangle \) is considered as a binary functor, and similarly for all the other constructs: hence, substitutions can be applied to any of them.

We find it useful to introduce a congruence relation \( \equiv \), stating when two systems are to be considered syntactically equal, and hence can be used one in place of the other:

\[
0 \mid S \equiv S', S \mid S' \equiv S'' \mid (S \mid S') \equiv S'' \mid (S' \mid S'') \equiv S''
\]

The former line states that operator “\( | \)” is associative, commutative, and absorbs 0 as usual, while second line states that a tuple can be either seen as joined into a single term, representing the whole “substance in the solution”, or split in two (or recursively more) terms down to tuples with activity value 1, as they were single “molecules”. Accordingly, to actually read the overall concentration of a tuple in a system, a partial operator “\( \equiv \)” is introduced, such that \( \tau(\equiv)S \) yields \( \tau(\equiv)S' \) if \( \tau(\equiv)S (\text{for any } m) \), and provides no result otherwise. Namely, when a system \( S \) is unified with \( \tau(\equiv)S' \) it means that tuple \( \tau \) has overall concentration \( n \) in \( S \), since we are sure that \( S' \) does not include any tuple with content \( \tau \).

Other than substitution, we also assume the existence of a (chemical-)matching function \( \mu \) for terms, that is application-specific rather than fixed. This should be such that \( \mu(\tau, \tau') \in [0,1] \); once a matching notion is defined in the application domain, matching gives 0 if \( \tau \) and \( \tau' \) do not match, 1 if they completely match (or are equal), and any value in between to represent partial matching. Matching function is in principle orthogonal to the concepts of logic instance and substitution: though, in our calculus the result of \( \mu(\tau, \tau') \) is used only if notation \( \tau/\tau' \) makes sense.

### Operational Semantics

The operational semantics of this calculus is given as a CTMC model (Continuous-time Markov Chains). A transition system \((S, \rightarrow)\) is defined where transitions are of the kind \( S \rightarrow S' \), meaning that system configuration \( S \) moves to \( S' \) with dynamics/likelihood expressed by rate \( r \); while finite rates (denoted as \( * \)) model a continuous-time transition, an infinite rate means instantaneous execution.

Transition relation is defined by the rules in Figure 1. Rule (OUT) states that as long as a process wants to perform an out, the tuple is immediately inserted in the space, and the process continuation \( P \) can carry on. Rule (RD) handles rd operations, which read any tuple \( t' \) that is an instance of the specified template: the process continuation \( P \) carries on after applying substitution \( t/\tau' \). Rule (IN) is similar to (RD) but it extracts tuples instead of just reading them. Rule (W) states that an agent executing a \( \text{wait}(r) \) simply waits a timeout with markovian rate \( r \). Note that the above rules and constructs are introduced only for the sake of completeness, for we do not focus much on internal agent behaviour in this paper.

Finally, rule (CHM) transforms tuple concentrations by a
chemical reaction: roughly, if the precondition $T_i$ is found in the space, then the rule can be applied that removes $T_i$ and replaces it with $T_o$. Such a rule is however a bit involved and requires a discussion. In the general case where a tuple set $T$ is found that is not equal to $T_i$, but rather a logical instance of it, note that $T$ will be transformed to $T_o$ modulo substitution as expected. However, note that $\{T_i/T\}$ could provide more solutions: for instance $\lfloor t(X) | t(Y) / t(1) | t(2) \rfloor$ could yield substitutions $\{X/1, Y/2\}$ or $\{X/2, Y/1\}$. For this case, rule (CHM) states that each different solution of substitution $\{T_i/T\}$ is allowed and leads to a different instantiation of the rule: hence, one will be chosen probabilistically depending on the markovian rate.

The markovian rate of rule (CHM) is given by $G(r, T, T|S) \ast \mu(T_i, T)$. The first factor computes the transition rate according to Gillespie’s algorithm; $G(r, T, T|S)$ is obtained by $r \ast count(T, T|S)$, where function $count(T, S)$ counts how many different combinations of tuples in $T$ actually occur in $S$, namely:

$$count(0, S) = 1$$

$$count(\tau(n) \oplus T, \tau(m) \oplus S) = \binom{m}{n} \ast count(T, S)$$

The second factor $\mu(T_i, T)$ makes chemical-matching enter the picture. Even though $T$ is an instance of $T_i$, for the application at hand it might be useful to promote or neglect the application of the reaction to $T$—the burden of this decision being delegated to matching function. A notable example in the context of this paper is that when $T$ is formed by the tuples of two services that might compose to each other: matching function will compute the appropriateness of such a composition, e.g., 0 means that there should be none. Accordingly, matching is ultimately used to properly tune chemical rate, whose actual value ranges from 0 to the rate as expressed by the reaction.

Finally note that a chemical reaction can provide an infinite rate, meaning that the reaction is instantaneously executed—if many of such rules are enabled for execution, one is chosen non-deterministically. Such a mechanism is useful to provide the tuple space with a functional reactive behaviour (e.g. to process certain tuples as they are inserted in the space). We assume that if matching is zero and $r$ is infinite, then the transition is not allowed—though this case is not considered in this paper.

### On Semantic Matching

As pointed out in previous section, chemical-matching function $\mu$ is key to promote or reduce the execution of chemical reactions. Though syntactic matching is the simplest form of matching one can adopt for $\mu$, it is easy to recognize that syntactic match can hardly deal with the openness requirement of modern software systems (e.g. pervasive services) — see also the criticisms on syntactic match-making as regards Web services [14]. Since a biochemical tuple space needs to accept the insertion of many different kinds of tuples, it would be impossible to completely foresee their syntactic structure at the time chemical reactions are designed. In addition, we want single chemical reactions to apply to a wide set of tuple kinds even though they do not belong to a common syntactic pattern, but rather adhere to a common ontological concept—e.g. they are all tuples concerning sport rather than food.

Accordingly, we recognise semantic matching as developed in recent research threads [14, 2, 17, 7] — a proper matching criterion for our coordination infrastructure. We then provide a biochemical tuple space with a semantic matching mechanism to retrieve tuples and apply chemical reactions that is: (i) application-dependent; (ii) not discrete (a tuple matches a template entirely or not at all) but returning a “vagueness” value between 0 and 1 [2]. Chemical reactions are then applied modulo match, and their actual chemical rate is affected by the degree of vagueness returned by function $\mu$, implemented according to semantic matching principles. As a simple example, suppose that, given chemical reaction $X \rightarrow Y + Z$, a tuple $t(100, john, 28)$ with concentration 50 is found that matches concept X with vagueness 0.4. As explained in previous section, the transition rate $r \ast \#t(100, john, 28)$ (i.e. $50 \ast r$) is actually multiplied by 0.40, meaning that the actual rate is 60% lower than expected, i.e. $r \ast 0.40$. While the maximum chemical rate of this reaction is $r$, $\mu$ guarantees tuples to be selected with a smaller rate: e.g. for $t(100, john, 28)$ such a chemical rate is $0.4 \ast r$, as if a specific instance $t(100, john, 28)$ 0.4 of the general reaction existed.

### 3. Abstract Model of Competitive Self-Composition

As an application of the chemical tuple space model, in this paper we focus on a competitive self-composition framework for services. In this scenario, a chemical tuple space is utilized to coordinate the behaviour of an open set of service agents and client agents. The tuple space is however programmed with chemical reactions that make services spontaneously compose, in the sense that tuples representing composition are automatically created and reified in the space: all the services (atomic or composite) compete with one another to serve requests – service activity level and matching are used by the tuple space to allocate requests – so that only successful services “survive” in the system, the other fading until being rarely used.

**Services as Chemical Components**

The implementation of the competitive self-composition framework over chemical tuple spaces starts from the idea of reifying the existence of a service through a proper service tuple, inserted in the space by the service agent as soon as it enters the system. The activity value of this tuple (e.g., initially equal for all services) measures the reactivity that the coordination system associates to the service: the higher it is, the more frequently the service will be exploited, and
conversely, if activity value is too low the service will end up being completely unused. On the other hand, clients manifest their intention of using a given service by injecting a request tuple, which differently from service tuple will always have activity level 1 (or 0 when the tuple is actually absent). The chemical-like behaviour we intend to enact is expressed informally by the following abstract reactions, which we orderly refer to as [DECAY, USE, COMPOSE]:

\[
\begin{align*}
\text{service}(S) & \xrightarrow{r_{\text{dec}}} 0 \\
\text{service}(S) & \xrightarrow{r_{\text{use}}} \text{service}(S) \\
\text{service}(S1) & \xrightarrow{r_{\text{join}}} \text{service}(\text{compose}(S1, S2))
\end{align*}
\]

Note that each reaction has a specific chemical rate that represents the maximum actual rate, as matching can reduce it depending on the actual structure of reactants—e.g. two services \(S_a\) and \(S_b\) may be likely to join or not. Accordingly, we refer to decay rate of a given service, use rate of a pair service/request, and join rate of two services, as the overall rate after applying the matching factor, namely, \(r \times \mu(T, T_i)\) by using the symbols introduced in Section 2.

**Competition**

Reaction [DECAY] states that any service decays. This is the most simple case of chemical reaction as described in [6], which causes the activity level of any service to decrease over time—each unit of activity level is handled as a molecule of a decaying chemical substance. For instance, in the simulation in Figure 2 (top-left), a tuple with initial concentration 1000 and decay rate 0.01 completely disappears after about 700 time units\(^1\).

Decay is a “negative feedback”, which is however meant to be balanced by the positive feedback of reaction [USE], stating that services and requests can be matched, so as to start an interaction process whose result is twofold: the request is consumed, and the service activity level is increased by one. Note that matching function takes care of associating a numerical match value to any service/request couple. As an example, suppose that while a service has concentration 1000 and decay rate 0.01, matching requests continuously arrive with rate 50 and use rate is 0.00005: the resulting behaviour is shown in Figure 2 (top-centre). The system reaches an equilibrium, where service has concentration 5000 (the ratio between arrival rate of requests and decay rate of services), and requests that are not served yet stabilise to few hundreds. Namely, after an initial bootstrap, the service concentration is proportional to the rate at which it serves requests, while the use rate is responsible of the service reactivity, hence of the number of unserved requests—for instance, with use rate 0.05 unserved requests remain bound to a few units.

Such simple reactions also intrinsically support service competition. Consider the above scenario, now with two services that match the same requests: one with a match leading to use rate 0.06, the other to use rate 0.04. As shown in Figure 2 (top-right), even though the services have initially even a slightly different use rate, the one with use rate 0.04 loses competition and fade until vanishing.

**Composition**

Reaction [COMPOSE] handles composition of services: it matches two services and, when selected, decreases by one their activity value, and increases by one the activity level of the composite service. Note that iteratively, the same rule can be applied to generate compositions of three or more services. The details of such a composition are abstracted here, and will be described in next section: surely, for two services to be composed, they should (at least partially) match according to function \(\mu\). Note that a composite service decays due to rule [DECAY] and is used by [USE] as in the case of atomic services, creating a positive-negative feedback loop providing a stabilisation.

To clarify the composition behaviour resulting from the three rules, we consider the following as a reference case study. Suppose two services \(S_f\) and \(S_h\) are defined: \(S_f\) is a service to book flights, while \(S_h\) allows one to book hotels. Specifically, after the user specifies the date and numbers of days he/she needs to spend in a place, \(S_f\) processes the request, returning a selection of flights available for a round trip around the required dates. Service \(S_h\) operates in the same way as far as hotel reservations are concerned. Figure 2 (bottom-left) shows the behaviour of the two services with initial activity level 100 and 30, serving requests as follows: they serve requests \(R_f\) and \(R_h\), concerning respectively requests for flights and hotels, both introduced with rate 25 and served with use rate 1.0. Moreover, there is also a third kind of request \(R_{fh}\), regarding reservations for both flights and hotels, introduced with rate 100, but served with rate 0.3 as the services only partially match such requests—0.3 could be understood as the fact that they can reply to only 30% of a cumulative request, or they can work 30% of the required time, or they can search the solution in a 30% portion of the search space, or the requester rated 3/10 the quality of result. All services have decay rate 0.01 as usual. The result is that both services reach an equilibrium at about activity level 7500. Figure 2 (bottom-right) shows what happens when the two services can compose leading to a new service \(S_{fh}\), which is supposed to be able to serve requests \(R_{fh}\) with use rate 1.0. The result is that \(S_{fh}\) now competes with \(S_f\) and \(S_h\) since it better serves composite requests \(R_{fh}\); \(S_f\) and \(S_h\) stabilise to activity values 3500, 5500, and 8000 respectively. We used a very low join rate \(10^{-8}\), so that without requests \(R_{fh}\), the activity value of \(S_{fh}\) remains bound to few units. Keeping that rate low is important to avoid generating high “concentrations” of composite services that are actually not useful.

We refer to “self”-composition since, even though the number of potential compositions of atomic services is very high, only those actually useful acquire a high activity value—the others remain a sort of noise in the tuple space.

### 4. CONCRETE MODEL

In this section we present, in terms of chemical reactions, a full formalisation of all application-independent aspects of the model, which can be used as a blueprint for implementation and for design-time system simulations.

**Chemical Reactions and Matching**

The syntax of tuples as can appear in chemical reactions is refined as follows. In the left-hand side of a chemical reac-
tion, a subterm can now come with a superscript numerical value. Accordingly, function \( \mu \) will work in an application-specific way by matching terms with same superscript, and “syntactically” everywhere else. Consider chemical reaction:

\[
\text{service(Id, Desc}^1) | \text{service(Id}', Desc'}^1) \quad \text{service(comp(Id, Id'), comp(Desc, Desc'))}
\]

Figure 2: Example simulations for: (top-left) decay; (top-centre) decay/use equilibrium; (top-right) competition without composition; (bottom-right) competition with composition.

Rather easy extension. Consider as an example the service for flight booking \( \text{sf} \) depicted in the left side of Figure 4 (A), which expects to receive a request for a flight from port \( i \), asks a backend agent (either a service or a client) with a message from port \( h1 \) to request a hotel reservation, receives the reply from \( h2 \), and then yields a reply to the requester through port \( o \); and similarly for \( \text{sh} \) in the right side, receiving a request to reserve hotels from \( x \), replying through \( y \). These services can be described by tuples:

\[
\text{service([sf(id502)], [in(i), out(o), out(h1), in(h2)], [in(o), out(h1), in(h2), out(o)])}.
\]

In our framework, two services compose first by linking the input port of a service with the output port of the other, as shown in Figure 4 (B); next, given one such composition, further links might be established, leading to a new composition with different configurations, as shown in Figure 4 (C). Service tuples representing such compositions are made by juxtaposition of their arguments. For cases (B) and (C) above, we have:

\[
\text{service([sf(id502), sh(id103)], [in(i), out(o), link(in(x), out(h1)), in(h2), out(y)], [in(o), out(h1), in(h2), out(o)], [in(x), out(y)])}.
\]

Note that the three arguments are sufficient to entirely describe the new configuration created out of a composition. Reaction [COMPOSE] in Figure 3 precisely describes the transition leading to the first service composition. It takes two matching service components \( C \) and \( C' \), two matching open ports \( B \) and \( B' \) in them (one input and one output), and creates the composed service. Note that this reaction decreases by one the activity level of components, and increases by one the activity level of the composite service. Similarly, reaction [LINK] creates a link in a single composite component. It should be noted that these two rules probabilistically enable any matching-compliant composition of

**Representation of Services and Compositions**

When a service enters the system managed by the tuple space, it injects a service tuple of the kind

\[
\text{service(ServiceIdentity, BondList, Protocol)}.
\]

\( \text{ServiceIdentity} \) is a list of unique service descriptions (one service description if the service is atomic), \( \text{BondList} \) is a list of terms expressing input/output unique ports (or bonds) of services (namely, its interface), and \( \text{Protocol} \) is a list of sequences of input/output ports (one sequence if this is an atomic service) representing the expected interaction history—providing a more expressive language for protocols, including e.g. non-deterministic choices, is a
services, from partial compositions like in Figure 4 (B), to simple (complete) compositions as in Figure 4 (C), from 1-to-many orchestration-like compositions, up to complex aggregates of several components (in principle, with no limit to their size)—the example introduced in Section 3 refers to Figure 4 (B); others can be studied similarly. Negative feedback is provided to all compositions through reaction [DECAY] as usual, while positive feedback is provided whenever clients successfully exploit a service, as described below.

**Matching Requests**

Clients look for services by injecting request tuples of the kind `request(ServiceDescription, AgentId)`. Such requests are matched with services by reaction [MATCH]; matching function takes care of matching `ServiceDescription` against the entire service tuple (i.e., considering the overall service configuration). The result of applying such a reaction is that the request is consumed, and a session tuple is created keeping track of: a newly generated identifier, the agent identifier, and the entire initial service information (denoted as `C, I, P` in the [MATCH] reaction). In particular, `P` stands for the protocol to be executed, and initially coincides with the service protocol: when this becomes `[]`, it means that the session is over and has been successful, hence, by reaction [COMPLETE] the session tuple is removed and the service activity level is increased by one.

**Client-Server Interaction**

As a session has been created, client and service can start exchanging messages through the tuple space, which keeps track of them by evolving the state of the protocol and by properly redirecting messages towards the components of composite services. In our framework this is achieved by instantaneous reactions [INPUT], [OUTPUT], and [IN-OUT], which all work in a fully syntactic way. Reaction [INPUT] checks whether an input message exists in the space that is directed to service session `X` and port `B`; in this case, if the session protocol correctly expects such a message, the protocol is updated and the tuple is turned into an accepted-input, meaning the service can (and should) retrieve and process it. Reaction [OUTPUT] is dual: a service produces a message that is turned into a produced-output, so that a client agent can retrieve it. Reaction [IN-OUT] handles direct exchange of a message between service and client. This is achieved by [IN-OUT] reaction, in the space that is directed to the protocol of the session of the service and the port `B` of the client. This direction is the opposite of what would happen in a traditional client-server interaction, where the client initiates the communication and the server responds. In our approach, the tuple space acts as a mediator, allowing the client to send messages directly to the service, bypassing the need for a client-server connection. This simplifies the interaction and reduces the overhead of establishing a new connection each time a request is made.

**Figure 3:** Chemical reactions for composition and use of services.

**Figure 4:** (A) Two services, (B) Composition after one linking transition, (C) Composition after two linking transitions.
two linked components: a produced-output tuple at one side directly becomes an accepted-input tuple at the other side. Finally, reaction [ABORT] handles the case where a client agent inserts an abort tuple to abruptly close a session, in which case – differently from reaction [COMPLETE] – there will be no increase of activity level.

Note that a minimal degree of access control is required to avoid service and client agents interacting in a wrong way through the tuple space: (i) server agents should publish a service tuple once (using an imposed initial concentration value), then monitor new sessions, consume accepted-input messages and produce output messages according to the declared observable behaviour; (ii) client agents can issue requests, send input messages and receive produced-output messages, and possibly abort sessions if the service is found not to be adequate—to avoid enacting positive feedback.

5. IMPLEMENTATION IN TUCSON

This section presents an implementation of the service chemical composition introduced in Section 4. The proposed implementation relies on a TuCSoN-based implementation of biochemical tuple space introduced in [19], which reports an extension of the classical TuCSoN tuple-centre model towards biochemistry by providing a tuple centre with a suitable set of ReSpecT reactions. These reactions make a tuple centre behave according to the operational semantics of a biochemical tuple space, as described in Section 2: put simply, they actually realise a TuCSoN-based biochemical tuple space. In a TuCSoN biochemical tuple space, chemical reactions are specified by tuples law(InList, Rate, OutList): e.g. a reaction \( x \rightarrow y \), involving tuples \( x \) and \( y \), is represented by \( \text{law}([x, y], 10, [y, y]) \). On top of this implementation, we defined all the chemical reactions for service composition and use, as described in Figure 3. For the sake of brevity, only the definition of reactions [COMPOSE] and [MATCH] is reported here: the other reactions have been defined along the same line.

\[
\chi[\text{COMPOSE}]
\text{law}([\text{service(Id1, Ms1, P1)}, \text{service(Id2, Ms2, P2)}], 10, \text{service(Id1, Id2)}, [\text{link(X1, X2)}, \text{Ms5}], [\text{P1, P2}]),
\text{match(X1, X2)},
\text{mset(Ms1, X1, Ms3)}, \text{mset(Ms2, X2, Ms4)},
\text{mset(Ms5, Ms3, Ms4)}).
\]

\[
\chi[\text{MATCH}]
\text{law}([\text{service(Sid, Ms, P)}, \text{req(Sdesc, Aid)}], 10, \text{service(Sid, Ms, P)}, \text{session(F, Aid, Sid, Ms, P)}),
\text{fresh(F)},
\text{match(service(Sis, Ms, Sdesc))}).
\]

As a sample value for \( r_{comp} \) and \( r_{case} \) we adopted 10. The two law tuples represent a slight extension of those in [19]. Indeed [COMPOSE] features, as additional parameters, a match tuple and a list of mset tuples: while the match tuple is exploited to realise the (semantic) matching mechanism to create links among matching open ports, the list of mset tuples defines operator \( \oplus \). To this end, the ReSpecT reactions in [19] – which define the actual engine of the biochemical tuple space – have been modified by introducing two new predicates: semantic-matcher and multiset.

Predicate multiset is a fully relational predicate with the signature:

\[
\text{multiset(?L1, ?Elem, ?L2)}.
\]

where \( \text{list L1} \) is \( \text{Elem} \oplus \text{L2} \). This predicate is adopted in the body of the extended ReSpecT reactions so as to provide the necessary support for those laws (e.g. [COMPOSE], [LINK], etc.) adopting operator \( \oplus \).

Predicate semantic-matcher implements the semantic matching mechanism, adopting the following signature:

\[
\text{semantic-matcher(+Desc1, +Desc2, -X1, -X2, -R)}.
\]

The predicate takes two service descriptions as inputs, returning a couple of matching ports and a vagueness value \( R \). As already explained, the vagueness value is bounded between 0 and 1. Similarly to predicate multiset, semantic-matcher is adopted in the body of the ReSpecT reactions as a support to those laws that exploit the matching mechanism, such as [COMPOSE], [LINK], [MATCH], and so on. For instance, as far as [COMPOSE] law is concerned, semantic-matcher takes the two service descriptions as input parameters, returning a couple of matching open ports, if any, and a vagueness value – 0 means that the two services cannot be composed. On the other hand, semantic-matcher is also adopted when [MATCH] is executed: in this case, it takes a service tuple and the service description within a req tuple, returning just a vagueness value, which specifies whether the request can be processed or not.

Finally, a third predicate fresh(-F) has been devised to define the superscript notation \( f \), used to generate new session identifiers. In other words, whenever [MATCH] is executed, fresh predicate is invoked within the body of the extended ReSpecT reactions, leading to the generation of a new ground term representing the identifier of a new session.

6. RELATED WORK

The work presented in the paper is strongly related to the field of automatic service composition [1, 12, 8]. Automatic service composition deals with the needs of users to search for appropriate service compositions that meet the required processes. Automatic composition is usually addressed by considering input and output interfaces of services, and a set of user-defined criteria that are exploited to automatically find suitable service compositions. To this end several algorithms and frameworks have been proposed for automatic service composition, ranging from techniques relying on AI [13] and distributed approaches [8] to frameworks for Web-service automatic composition [16, 12]. While these approaches are mostly syntactic – i.e. based on template-matching techniques – other techniques have been devised to deal with semantics [10] and dynamic scenarios – such as pervasive systems – where exact matching is not practical [9].

However, to the best of our knowledge, only few attempts have been made to address automatic composition of services by considering self-organisation as a viable approach to promote automatic composition based on self-aggregation and emergent service composition. For instance, in [15], the authors recognise that the dynamism and openness of emerging scenarios – e.g. those related to pervasive computing – require a dynamic and flexible service aggregation that cannot be tackled by traditional composition approaches. To this end, they propose a model for pervasive-service composition inspired from ecology. As such, they exploit semantics for service aggregation, and propose a sort of ecological substrate – coming from the underlying social and physi-
7. CONCLUSION AND FUTURE WORK

This paper presents the competitive self-composition framework for services, and provides a formal account of its mechanisms. Despite this model looks promising and provides a coherent ground, the road towards its full evaluation, realization, and application is inevitably longer than the scope of this paper. In particular, future works will be devoted to develop several activities: (i) tuning reactions, since the abstract reactions reported in Section 3 appear to suitably support the balance between composition and competition, it should be evaluated whether alternative schemata can provide a better engineering support, or a better fitting with chemical behaviour; (ii) tuning rates, as in the paper we never discussed how we can come up with proper decay, use, and compose rates, though their actual value is key to the resulting system dynamics; (iii) semantic matching, since the issue of finding proper mechanisms to tackle semantic matching is key for the success of our proposal, especially as far as open systems are concerned; (iv) developing case studies in the context of web service orchestration and pervasive service systems.

8. REFERENCES


