Alma Mater Studiorum · Università di Bologna

SECONDA FACOLTÀ DI INGEGNERIA CON SEDE A CESENA
Corso di Laurea in Ingegneria informatica

BioTuCSoN: biochemical extension of TuCSoN to support self-organising coordination

Tesi di Laurea in Sistemi Multi-Agente

Relatore:
Chiar.mo Prof. Andrea Omicini

Presentata da:
Marco Piraccini

Correlatore:
Ing. Stefano Mariani

Secondo appello, III sessione
Anno Accademico 2011/2012

"...limits, like fears,
are often just an illusion."
- MJ

Introduction

In the last twenty years, but more markedly in the last decade, we have witnessed an extraordinary technological evolution. The scientific progress in the field of hardware engineering has led to the development of *smart devices*, provided with an increasing computational power into a reducing skeleton. The rapid and continual spread of such kind of technologies, more and more commonly used by people in daily life, have brought to dramatic change the ICT landscape. A new scenario can be outlined, where computing systems are anywhere, embedded in environmental object, always connected, for example by means of wireless technologies, and always active to perform tasks on our behalf. It is known as *pervasive computing* paradigm, and its purpose can be briefly defined as design "machines that fit the human environment instead of forcing humans to enter theirs". Accordingly to this model, people could be connected with each other, or with environmental items, with the aim to retrieve useful informations concerning our own interests or contingent necessities. The possibility to have at your disposal specific knowledge in a given context is even more essential in a world bombarded with a terrific amount of data, and in which people have a short time to select and use it.

The delineated profile implies the creation of *complex systems* characterized by numerous interconnected elements, each one with a defined computational capability, that have to coordinate themselves and interact aiming to achieve global goals, other than personal ones. In order to be able to manage the new requirements, we direct our attention to the study of dynamics of biological *eco-systems*. Analysing them, it is possible to extract some recur-

ring and useful patterns; in particular, it is observed that, generally, natural *environment* is populated by a certain number of *individuals*, with a limited intelligence, that achieve an organised global behaviour arising from simple local interactions (phenomena called *emergency*).

Our work starts from studying a specific project, that takes inspiration from these principles, named SAPERE ("Self-aware Pervasive Service Ecosystem"). It is conducted by an European collaboration, that also involves the University of Bologna, and focussed on the development of a highly-innovative nature-inspired framework, suited for the decentralized deployment, execution, and management, of self-aware and adaptive pervasive services in future network scenarios[25]. After we have inferred the principal abstractions and architectural features of this approach, we want to evaluate how a nature-inspired software system can be realized, considering biochemical tuple spaces model. This analysis leads us to individuate the essential characteristics, necessary to build on its concrete implementation. So, starting from the existent tuple spaces infrastructure TuCSoN, we realize a first biochemical released(*BioTuCSoN*), reifying the principal concepts acquired by the previous studies. Comparative performance tests between TuCSoN and BioTuCSoN are, then, provided, in order to prove respective advantages in different situations. Our work ends evaluating a case study that points out the capabilities of the realized biochemical technology.

The thesis is organised as follows. In chapter one we present the SAPERE project, first through general considerations and then analysing, carefully, its structure and model. Chapter two provides a discussion on biochemical tuple spaces, analysing some background researches and, then, deducing from them essential abstractions and features. Chapter three describes TuCSoN technology that constitutes the basis from which we start to define BioTuCSoN. The related work process is explained in chapter four. Finally, chapter five shows an interesting case study, highlighting the results achieved by means of the weighted probabilistic behaviour of BioTuCSoN.

Contents

Introduction **i**

1 SAPERE project **7**

1.1 SAPERE approach 11

1.1.1 SAPERE logic architecture 11

1.1.2 The eco-laws framework 13

1.1.3 The adaptive displays use case 15

1.2 SAPERE basic abstractions 16

1.2.1 Model entities 16

1.2.2 Eco-laws 19

2 Biochemical Tuple Spaces **21**

2.1 Biochemical tuple spaces model 22

2.1.1 Space 22

2.1.2 Reactions 23

2.1.3 Chemical engine 24

2.1.4 Semantic matching 26

2.2 Some examples 27

2.2.1 Services competition 27

2.2.2 Pattern based on gradient 29

2.3 Mapping on TuCSoN 30

2.3.1 Model analysis 31

3 TuCSoN & ReSpecT **37**
3.1 TuCSoN 37
3.1.1 Model & Language 38
3.1.2 Architecture 41
3.1.3 Programming tuple centres 43
3.2 ReSpecT 43
3.2.1 ReSpecT language 44
3.2.2 ReSpectVM 44

4 Biochemical TuCSoN **49**
4.1 Motivations 49
4.2 Architectural requirements 50
4.3 Bio extension foundations 51
4.3.1 Bio tuple 51
4.3.2 TuCSoN code analysis 56
4.3.3 Bio primitives 60
4.3.4 Surrounding changes 70
4.4 BioTuCSoN performance 70
4.4.1 Test environment 71
4.4.2 Performance results 72
4.5 Bio ReSpecT 74
4.5.1 Our work 75

5 Case study **79**
5.1 Service ecosystem 79
5.1.1 General context 79
5.1.2 Specific scenario 80
5.2 Test system architecture 81
5.2.1 Tuple space programming 81
5.2.2 Services and clients 83
5.3 Simulations planned 84
5.3.1 NoiseAndCompetition 85

5.3.2 FeedAsDecay 88
5.3.3 TemporaryFeed 89
5.3.4 ServComp41Req 90

Conclusions & Future works **93**

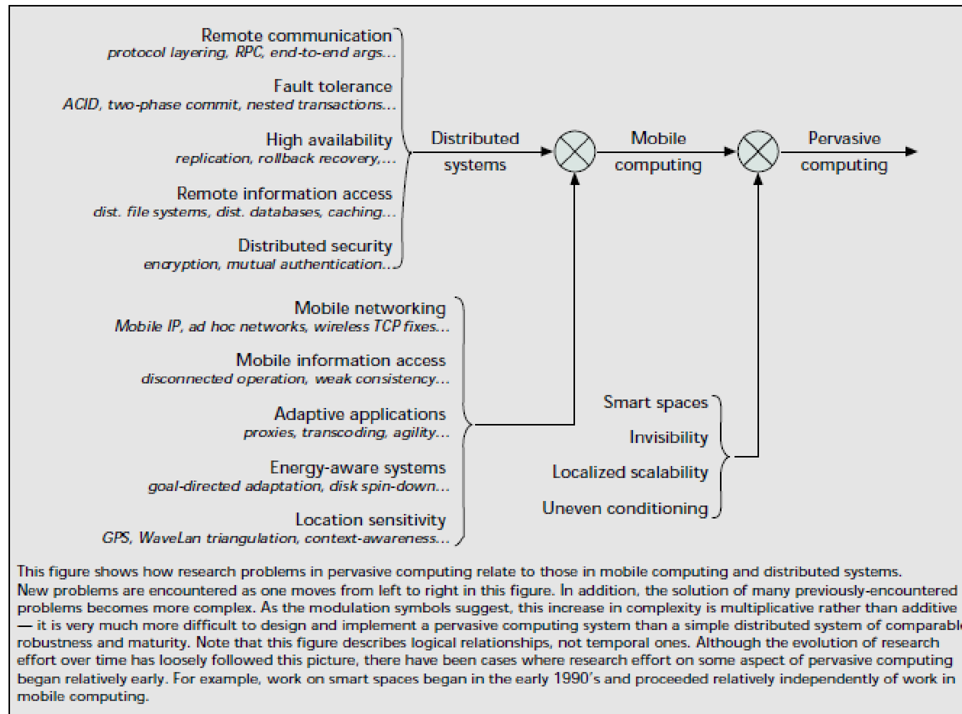
Chapter 1

SAPERE project

The expression "ubiquitous computing", then also called *pervasive computing*, was coined by Mark Weiser in the influential 1991 paper [23]. In those years the technological development was not so mature to really provide the described scenario, but now the situation has totally changed. The increasing spread of advanced computational devices, the huge progress concerning hardware solutions, that leads to integration in smaller and smaller skeletons computational capabilities in conjunction with more and more durable memories, are all elements that have revolutionized the ICT landscape. The strong integration between social environment and technology offers a wide set of possibilities for a better exploitation of the enormous amount of information, available thanks principally to internet. For example, we should desire that it is not users who search for information but the contrary. So, a public display could show only the useful information for the user in front of it, given some personal preferences saved on a user device and depending on a specific environment context.

But pervasive computing scenario, besides ensuring some interesting properties and applications, presents also challenges across computer science. Several issues have to be considered during designing and implementation stage and, some of them, have never been faced before. Actually, this approach does not come out of the blue, but is the result of previous evolution steps

that are *distributed system* and *mobile computing*. So, some problems can be faced referring to researches about these matters. We can outline the relationships between each evolution steps by means of the following taxonomy of issues [17].



This figure shows how research problems in pervasive computing relate to those in mobile computing and distributed systems. New problems are encountered as one moves from left to right in this figure. In addition, the solution of many previously-encountered problems becomes more complex. As the modulation symbols suggest, this increase in complexity is multiplicative rather than additive — it is very much more difficult to design and implement a pervasive computing system than a simple distributed system of comparable robustness and maturity. Note that this figure describes logical relationships, not temporal ones. Although the evolution of research effort over time has loosely followed this picture, there have been cases where research effort on some aspect of pervasive computing began relatively early. For example, work on smart spaces began in the early 1990's and proceeded relatively independently of work in mobile computing.

Now, we want to scan the new requirements introduced by pervasive computing scenario. First of all, such kinds of systems should deal with a high *dependability*, since they have to work continuously, ensuring a reliable use experience with minimal maintenance support. Then, a basic property is to support *self-** features, that are, principally, self-management and self-adaptation, so that pervasive infrastructures can survive contingencies without any human intervention and at limited management cost[20]. These are intrinsically related with *adaptivity*, another important characteristic, that expresses the necessity of adapting themselves automatically to changes. It is, also, to underline the role of the system to manage context-awareness and *situatedness*, i.e. for example, allowing display to show the right informa-

tion, at the right time, depending on environment status and user necessities. Moreover, in pervasive systems in every moment could appear new services, requests, users or physical items, so it is indispensable to satisfy requirements such as *openess*, *robust evolution* and *flexibility*.

Now, the question is: how can we design and realize a software system facing all these aspects?

A possible solution comes from the eco-system in which we human beings live. Latest researches address their studies towards other, apparently uncorrelated, science fields such as chemistry, biology, physics or ecology. By means of such inter-disciplinary approaches, it is possible to extract, from each area, models, methods and techniques useful in the design of pervasive software systems.

Each field of study entails specific metaphors, analysing the same category of problems at different levels of abstraction. We will see them one by one, starting from low-level [26].

Physical metaphor

The components of the system are modelled by physical particles, that live together into and are affected by a sort of virtual computational fields, which represents the coordination media. Particles' activity are determined by a set of rules that define how each of them is influenced by the field, following the value of its gradient. Depending on this information a particle can modify its status or move into the space. The virtual field can be simple (euclidean field) or complex (gravitational field).

Chemical metaphor

The eco-system is composed by computational atoms or molecules, that have internally the description of the characterizing properties, in a formal way. These, substantially, represents the role of each component into the space. The set of rules, that models eco-system behaviour, is shaped by a set of chemical reactions. They are formed by a list of reagents and a list of prod-

ucts. Once firing, a reaction can link together different molecules depending on some property values, insert or remove components and so on. The space is subdivided into several chemical compartments, through which molecules can move freely to ensure a global interaction.

Biological metaphor

It is considered to be a small biological environment. Components are simple cells or animals with limited intellectual capabilities. They act over the space based on some basic goal-oriented behaviour, that are influenced by chemical signals spread over the environment. This trace can be scattered by means of the individuals themselves, that so can be affected by the behaviour of others (mechanism named stigmergy). In this kind of system, the set of rules specifies how chemical signals are spread, how much time is required for them to evaporate and in what way they influence individuals' behaviours.

Ecological metaphor

It represents the higher level of abstraction and models individuals as animal species provided by some form of intelligence. Their behaviour is guided by a personal goal to achieve, as, for example, the research of resources necessary to survive. The ecological rules (or laws) are in charge of defining how individuals can find resources and in what conditions they can perform specific tasks (i.e. eat, reproduce,etc). It means, basically, that laws govern system dynamics ruling the interactions between individuals of the same and different species. Similarly to chemical systems, the shape of the world is typically organized around a set of localities, i.e. ecological niches, yet enabling diffusion of species across niches.

Now we will analyse the SAPERE project, that wants to face the new requirements of the pervasive scenario, referring to a nature-inspired solution.

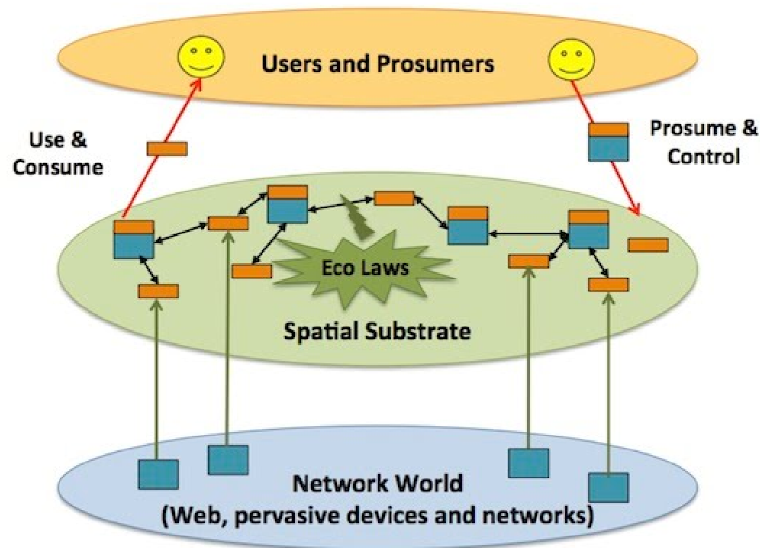
1.1 SAPERE approach

The European SAPERE project ("Self-aware Pervasive Service Ecosystems") was born in 2010 (and will end in September 2013) with the aim to develop "a highly-innovative theoretical and practical framework for the decentralized deployment and execution of self-aware and adaptive services for future and emerging pervasive network scenarios".

As we have said, SAPERE draws inspiration from natural ecosystems in order to tackle the new challenges induced by the pervasive computing scenario. In particular, the purpose is to face the problem, from the foundations, conceiving a new way of modelling pervasive systems, in order to consider and satisfy every requirement.

1.1.1 SAPERE logic architecture

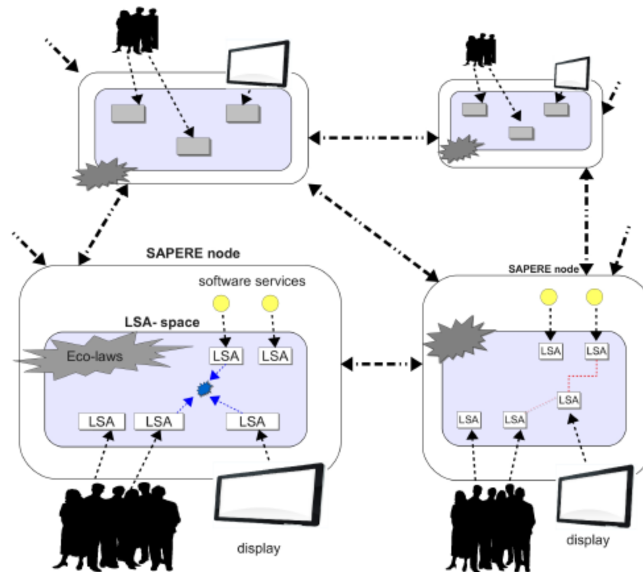
The analysed scenario concerns a distributed computational ecosystem in which a lot of services, data and devices engaged in very dynamic and flexible coordinated activities. Self-organisation is essential in order to ensure context-awareness and manage the coordination activities between components physically close to each other. Moreover, the system should provide the ability of supporting the communication between components, without their prior-knowledge, promoting an interaction pattern which is self-adaptive and self-managing. To deal with these issues, we refer to a nature-inspired solution, modelling the pervasive service environment as a non-layered spatial substrate, mapped above the actual pervasive network infrastructure[21]. The substrate embeds the basic laws of nature (named eco-laws) that rule system behaviour. System components (devices, users, software services) are modelled as individuals of different species. They interact and combine with each other, complying with the eco-laws and generally based on their spatial relationship, so as to achieve personal goals as well as global interests. Human users can interact with the eco-system acquiring data and services (as *consumers*) or, also, inserting requests or information (as "*prosumers*").



Each individual of the ecosystem has associated a semantic description by means of LSAs (Live Semantic Annotation). They are “live” and active annotations, strictly coupled with the described component, considering, also, the current situation and context. They are assigned at design time to individuals, performing the role of their observable interface and allowing dynamic forms of aware interactions, leading by semantic issues. More in concrete, it means that components evolve influencing and being influenced, internally, in their LSA description, by others individuals or by the environment and deciding with who interact, observing LSA information.

In any SAPERE node there is included a LSA-space in which self-adaptive coordination mechanisms take place so as to mediate the interaction between components. Whenever an individual comes close to a node, its LSA is, automatically, injected into the LSA-space of that node, entering into local coordination dynamics. Similarly, when the component goes away, its LSA is removed. In turn, also SAPERE nodes can be connected together based on physical or logical proximity. Each node can refer to or interact with another one through exchange of LSAs.

The eco-laws, that govern system behaviour, are inserted into SAPERE nodes. They are modelled by proper *chemical reactions*, involving LSAs of the individuals. Their firing condition is ruled by probabilistic and semantic aspects, and involves actions such as definition of new connection between entities, production/removal of LSAs or their spread over the network, from one node to another.



1.1.2 The eco-laws framework

Now we will see a possible language for eco-laws, focussing on syntactic structure of LSAs, properties of eco-laws and matching issues.

As first, we show, specifically, what we mean with **LSAs**. They are semantic annotations similar to RDF(Resource Description Framework) and they can be expressed as:

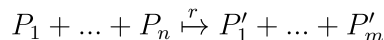
```
i : [ p1 = v1, ... , pn = vn]
```

where i represents the unique LSA identifier (over the entire eco-system), pi expresses property's name and vj the related value. There are, also,

some particular properties/values (starting with "#") that are managed in a specific way by the infrastructure, providing an automatic reification of some aspects of the environment inside the LSA-space. In particular, each LSA-space must contain one LSA of type #location for each neighbour, showing its *id* and its extimated distance *d*, and one of type #time indicating the current time. For example:

```
i : [type=#time,value=t]
i : [type=#neighbour,where=d,distance=d]
```

The **eco-laws** can be composed of two types of LSAs: LSA pattern and LSA ground. The first can include variables instead of some values while the latter can not contain any variables. A LSA ground L matches with a LSA pattern P if there exists a substitution of variables to values that applied to P gives L. An eco-law can be expressed as:



$$P_1 + ... + P_n \overset{r}{\mapsto} P'_1 + ... + P'_m$$

The left-hand side shows the list of *reagents*, while the right-hand side the list of *products* involved in the reaction. LSAs $L_1, ..., L_n$, that matches reagents $P_1 + ... + P_n$, are extracted from the space, saving the bindings between variables and values. So, they are replaced by the LSAs obtained applying the previous associations to products $P'_1 + ... + P'_m$. Rate r expresses the frequency at which the reaction fires. More precisely, the execution of an eco-law can be modelled as a CTMC (Continuos Time Markov Chain) transition with Markovian rate r. The application of an eco-law can be subdivided in two steps:

1. Iteratively, a reagent pattern P_i is non-deterministically selected from the eco-law and, accordingly, is retrieved from the space an LSA L_i, that matches P_i. The results of the unification is, then, applied to the remainder of the eco-law.

2. In case the iteration ends, the products set up the set of LSA to put into the space.

An eco-law can involve only LSAs of the same space and the products can be inserted only in the local space or in one of the neighbours. Such kind of constraints are verified through the #location properties.

Finally, we want to consider aspects concerning mechanisms of **matching**. First of all, we have to distinguish between *reagents pattern* and *products pattern*. The former type of template has to specify only the associations properties/values that allow to select the required LSAs. On the other hand, products pattern has to detail only the values for the properties that have to be modified. Moreover, the associations can be expressed not merely with "=" operator, but also with others, that allow, for example, to match a variable to more than one value ("=*"), or to filter LSAs depending on properties value ("has" and "has-not"), or to add/remove value from specific products' properties ("+=" and "-="). Controls about semantic match can be managed by adding a fuzzy predicate fp that returns, instead of yes/no, a value in [0,1]. This further operator can be used also for simple computations.

1.1.3 The adaptive displays use case

Here we want to shortly describe a potential use case (extracted by [21]), with the mere purpose to point out, concretely, how this infrastructure can be exploited.

We can imagine a public area, such as an airport, in which a lot of people wander around with their personal devices. We suppose that each device keeps user preferences and is equipped with proper sensors that perceive user behaviour. All around, in the public area, are spread several displays that visualize different kind of services and information. They should have to acquire data and preferences from the closest user, in front of it, with the aim to provide the best service in regard to user needs and environmental status.

SAPERE approach ensures these dynamics, distributing in the environment a proper number of nodes. Each one includes a LSA-space and is

located in a specific area, so that when an user comes in front of a display, one LSA-space must contain: LSA of display, LSA of the user device and LSA of the required service. For example, it could be provided one LSA-space for each display. Then to allow displays to visualize in every moment the right information, self-organising patterns are necessary . They are realized through three basic concepts: SAPERE architecture (just described), a proper set of rules, a probabilistic engine to execute them (following CTMC process) and, finally, a mechanism for semantic matching.

Now we, abstractly, define what kind of rules the system needs. First of all, it is necessary that a reaction allows an LSA to link to a user device with one of a displays, that expresses the will to visualize contextualized information. Similarly, it is essential to bind the LSA of a display with LSAs of every visualizing service suitable for that display. Specific properties, within LSAs of the displays, indicate what service has to be shown and towards who. Lastly, it is required a reaction that executes the visualization, connecting, actually, display and service and setting some relevant properties such as the current time and the display status.

1.2 SAPERE basic abstractions

Here, we want to infer the fundamental features and abstractions required by SAPERE infrastructure, and suggest a first potential mapping with existent technologies. From the previous description, it is clear that a tuple spaces approach could be a potential solution. We will try to explain why and how, referring, in particular, to TuCSoN technology (described in chapter 3).

1.2.1 Model entities

Analysing SAPERE model, it is possible to outline its architecture with the following tuple:

$$< Set(\sigma), Set(L), Set(F), Set(r), Set(\theta), \triangleright, loc, @ >$$

Variables

σ : unique identifier for node/space;

L : LSA;

F : LSA filter;

r : abstract rate for scheduling policies;

Functions

θ : expresses how to bind terms together;

$\triangleright$: $Set(F) \times Set(L) \rightarrow Set(L)$, update function based on filters;

loc : $Set(L) \rightarrow Set(\sigma) \cup \{*\}$, extracts the LSA's target location;

@ : $Set(L) \times Set(\sigma) \rightarrow Set(L)$, creates a clone of a LSA into another location.

The system is composed of several LSA-spaces, spread over the physical network. Any **LSA-space** is a multi-set of LSAs, characterized by a unique identifier (σ_i) system-wide. This aspect is, for example, naturally modelled in TuCSoN, through the concept of tuple centres, expressing the identifier as *tname@netid:portno*, where *tname* stands for the name of the tuple centre that is located into a TuCSoN node hosted by a network device *netid* on port *portno*.

The **LSAs**, that fill the space, are semantic tuples characterized by unique name system-wide and a list of items in the form properties-values. They can be expressed as: $i\langle p\bar{o}; p\bar{o}; ...; p\bar{o}\rangle$, where i represents LSA's identifier, p some kind of properties to which are associated one or more values ($\bar{o} = o_1, ..., o_n$). The values can be atomic values or, in turn, a list of elements properties-values. LSAs can be reified by means of TuCSoN tuples (namely Prolog atom) with some specific considerations. Specifically, it is required to ensure

uniqueness to tuple's name, for example considering some controls over the insertion operation, so as to merge tuples with the same names or prevent the addition of new ones, if they are incompatible with namesake tuples in the space. For example, a LSA of kind: $id911\langle src = s01; type = service; ...\rangle$, can be represent with a TuCSoN tuple: $id911(src(s01), type(service), ...)$.

A **filter** is a set of patterns, each one being, namely, a sort of LSA-template. Other than matching function, it makes available further operations over LSAs, such as assigning variables to some value obtained as a result of a specific expression or to a value that makes true some boolean predicate. Moreover, filters provide a characteristic syntax to express several relationships between properties and values. TuCSoN templates are quite similar to these, but do not provide the same level of expressiveness, and so should be extended with the required functionalities.

Binding and **update function** follow the rule of logic unification. Since TuCSoN communication language is logic-based, these aspects are naturally managed.

Location function allows to define the location which a given tuple belongs to, while **clone function** ensures a transactional move from the source space to the destination one. These can be implemented with the existent TuCSoN abstractions such as a specific tuple argument and reactions ReSpecT (that have a transactional semantics), or they can be inserted at infrastructural level, realizing proper extension of TuCSoN's Java code.

Finally, we have to consider how the system is configured. It is a multi-set of LSA-spaces, eco-laws and topological connections. The last ones define the neighbourhood structures for each space and they can be modelled in TuCSoN which proper tuples, possibly confined into a specific area of the tuple space. Eco-laws need to some important considerations, dealed with separately in the following.

1.2.2 Eco-laws

As we have stated, eco-laws in SAPERE are expressed as $R \xrightarrow{r} P$, where $R, P \in F$, and F represents a LSA filter. Accordingly with the previous observations about filters, we can simply model eco-laws in TuCSoN as:

law([Reagents],rate,[Products]),

where Reagents and Products ara both a list of TuCSon templates.

But how can they be used? The eco-system's dynamics involve three possible transitions: the application of an eco-law, named [REA], the diffusion of LSAs towards a specific LSA-space among neighbours, named ([DIFF]), or towards all of them, named ([BRO]). Now we, shortly, describe them individually:

[REA] : retrieves from the space the LSAs that match templates in R, then, considering the defined bindings, inserts the LSAs specified in P into the space.

[DIFF] : if, in some space σ, there is a LSA in which the value of location properties is $loc = \sigma_i$, with $\sigma \neq \sigma_i$, then removes such LSA from σ and inserts it into σ_i, if these spaces are defined connected in the system configuration.

[BRO] : if some space σ there is a LSA in which the value of location properties is $loc = *$, then removes such LSA from σ and inserts it into all the neighbouring spaces.

The operational semantics, just abstractly described, could be implemented in TuCSoN by means of a proper set of ReSpecT reactions, that could reify a sort of chemical engine allowing it to select and execute the eco-laws in a probabilistic way and at variable time intervals. To ensure better performance and better system usability there should be provided an integrated version of the chemical simulator, acting directly on Java code of TuCSon.

Chapter 2

Biochemical Tuple Spaces

Starting, again, from the pervasive scenario, previous described, we want now to focus on a possible solution, that takes inspiration from a nature-inspired approach, in particular applying the chemical metaphor, i.e. the biochemical tuple spaces model. Its purpose is to engineer the spatial coordination and self-organisation of distributed pervasive services, into today's complex system, by means of a specific computational model based on chemical reactions. Tuple spaces are extended with the ability of evolving tuples similarly to a chemical system; so tuples can be seen as chemical substances to which is associated a value that expresses their activity/pertinence into a given context. As for chemical solutions, they can move inside a single-compartment, namely a tuple space, or from one to another. Finally, the coordination rules are organised as chemical reactions, selected and executed by a proper chemical engine, whose behaviour has to allow the essential system properties of self-organisation, adaptivity and self-management.

Described the reference model[20] and discussed some examples[18], then we delineate the essential features typical of biochemical tuple spaces; so, we try to define a potential concrete mapping, taking TuCSoN as background technological infrastructure.

2.1 Biochemical tuple spaces model

The current ICT landscape offers a lot of sensing and actuating devices that leads to the possibility of developing computational environments, composed by pervasive services. They have to be situated and evolve themselves in relation with the social and physical context. Standard solutions, such as Service-Oriented Architecture (SOA), are not able to manage naturally the newly introduced features (situatedness, adaptivity, self-*,etc). So, in order to prevent complex and time-consuming implementations, we turn to chemical-inspired tuple spaces, that exploit their intrinsically distributed architecture and bio patterns to face the new requirements. The components of the system are coordinated as though they were molecules fluctuating into a distributed space under specific chemical rules. System stochasticity and dynamism issues, are supervised by a proper modality of selection and execution of the bio-laws. We show how the basic properties of pervasive systems are, naturally, satisfied by biochemical tuple spaces:

- *situatedness*: local state is defined by tuples into the space;
- *self-**: global self-organisation and adaptivity are satisfied by a proper set of chemical-rules, drawn from natural patterns (e.g. prey-predator system) and performed by a chemical simulator.
- *diversity*: semantic matching for reagents extraction and products insertion ensures to define general rules that can be applied in contingent or potentially new cases.

In the following we will illustrate how a distributed system based on biochemical tuples spaces can be constructed.

2.1.1 Space

The global distributed space is modelled as a sort of graph that defines specific relationships of connection between the nodes of the system. It means

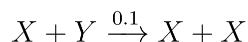
that each node (named *location*) can directly communicate only with a given set of neighbours. Any location hosts a tuple space, into which the local dynamics of interaction take place.

These are, principally, ruled by the evolution of tuples' *concentration* over time, as a consequence of the probabilistic application of chemical rules (or reactions). The value of concentration is modelled as an integer number, non-negative, that states the level of pertinence/activity of the related tuple in a given context. A high value implies a high probability for the tuple to be engaged in a chemical reaction thereby affecting system behaviour. Concentration varies over time as it changes its relevance in the environment.

As already stated, tuple spaces are connected to each other in order to simulate the process of chemical diffusion of molecules through membranes. A tuple, that is scheduled to be spread towards other neighbouring tuple centres, is called *firing tuple*. Each connection is realized by an one-way link, characterized by a rate r that measures the maximum transfer of tuples for time unit. The diffusion process can be affected by the computational field self-induced due to the same motion of the tuples. Each firing tuple defines a local gradient G and a local gradient δ in [0,+,-]. When δ is '0' the transfer rate r is not influenced by G, when δ is '+' the tuple tends to ascend G, while tends to descend it when δ is '-'. To reify the probabilistic evolution of the chemical processes, it is essential to introduce the concept of thermodynamic *noise*. It expresses the probability that a firing tuples moves off contrary to the information specified by the gradient δ, introducing a mechanism similar to simulated annealing.

2.1.2 Reactions

The local dynamics inside each tuple space as well as the global system behaviour are governed by proper set of rules, installed into each node. They are modelled as chemical reactions of the type:



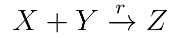
$$X + Y \xrightarrow{0.1} X + X$$

It means that two tuple x and y, that match respectively the reagents X and Y, are selected from the local space; so y decreases its concentration by one unit, while concentration of x increases by one. But to fully understand the semantics of the execution of reactions we have to analyse the behaviour of the chemical engine.

2.1.3 Chemical engine

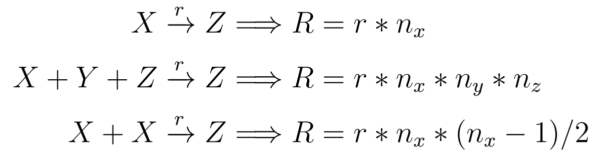
The evolution of the computational environment and of its components must occur with dynamics similar to those of a real biochemical system in order to satisfy "pervasive requirements". Since system behaviour is defined and governed by the set of rules inserted into each tuple centre, the previous assertion means that it has to be defined a proper way to manage such rules, designing a chemical engine that selects and executes them. Thanks to the work accomplished by Gillespie[8], we have at our disposal a stochastic formal meta-model that simulates the evolution of a biochemical system by means of a CTMC (Continuous-Time Markov Chain) computational system. It is substantially a variation of DTMC (Discrete-Time Markov Chain), in which edges, that represent system transitions, are labelled with probability instead of rates and do not require a continuous time to be triggered. A transition is the abstraction used to model a single evolutive step of the biochemical system, namely a chemical reaction. The associated rate represents the average firing frequency, computing the time interval between two consecutive occurrences of a transition with the negative exponential probability distribution.

Accordingly with *Gillespie's algorithm*, the probability of a reaction to be triggered is affected not only by the rate intrinsically associated, defined at design-time, but also by the concentration of its reagents. This value varies over time and so is a key element to achieve context-dependent properties. We illustrate this mechanism in detail. Consider a solution of substances X, Y and Z with n_x, n_y and n_z concentration values (or molecules in chemical acceptation), and a chemical reaction with intrinsic rate r:



$$X + Y \xrightarrow{r} Z$$

Its meaning is: one molecule of X binds with one of Y, transforming into a single new molecule of Z. It entails that n_x and n_y decrease by one, while n_z increases by one. The real rate (called *markovian rate*) R associated to the reaction is given by the intrinsic rate r multiplied by the number of possible combinations of molecules that cause the reaction, in the previous case: $R = r * n_x * n_y$. Other cases can be:



$$X \xrightarrow{r} Z \Longrightarrow R = r * n_x$$
$$X + Y + Z \xrightarrow{r} Z \Longrightarrow R = r * n_x * n_y * n_z$$
$$X + X \xrightarrow{r} Z \Longrightarrow R = r * n_x * (n_x - 1)/2$$

A possible algorithm[8] for the execution of chemical reactions is:

1. at each step calculate the markovian rate of all reactions $r_1, r_2, ..., r_n$, whose sum is S;
2. choose and apply one of them with probability: $P_i = r_i/S$
3. calculate time interval between execution steps: $\Delta t = log(1/\tau)/R$, where τ is a random number in [0,1].

This algorithm permits to obtain a system evolution suitable to model bio pattern[6]. For example, prey-predator dynamics can be modelled by: $X + Y \xrightarrow{r} X + X$, where a predator X first eats a prey Y and then generates a son. In particular, such a chemical engine allows to realize some useful spatial coordination and competition between the individuals of the system so that the most significant ones wins over others. In this way, the status of the system is always updated and evolve in relation to the contingent necessities of every moments. In the following, we will show some examples of these patterns.

2.1.4 Semantic matching

As discussed in the introduction, an essential aspect for pervasive environment is that at every time some new devices, services or requests can be introduced into the system. It is clear that a prior knowledge, at design-time, of every possible elements is impossible or, at least, impracticable. To manage this requirement, named *diversity*, is necessary that biochemical tuple spaces provide a function of semantic matching. We can synthesize its role saying that it wants to allow, for defining general reactions that can uniformly be applied to specific cases. An approach is to assume that each association between a reagent R and a tuple t leads not to a net value but fuzzy, i.e. a value in [0,1]. This value will be "high" if t has a strong matching with R, otherwise it is low. The matching degree affects the markovian rate so that if into the space there are only tuples less correlated with the reagents of a reaction r, the probability for r to fire is reduced. So for example, considering a reaction $X + Y \xrightarrow{r} Z$, the global rate, updated with the value of matching degree, can be expressed by: $G = r * n_x * n_y * \#x * \#y$, where $\#x$ and $\#y$ represent respectively the matching degree of tuple x with reagents X and the matching degree of y with Y.

To evaluate match degree, it is necessary to extend the system with the ability of performing semantic reasoning over tuples, that means to consider not only their syntactical structure but also their semantics. Several studies have been conducted on this critical topic; a possible solution based on ontology is proposed in [12]. It considers a tuple space as a knowledge repository structured as a set of tuples and, using *Description Logic*(DL), wants to define relationship between them. In particular, it describes the domain ontology through the notions of *concepts* and *roles*. The first denotes meaningful sets of individuals while the latter denotes relationships among individuals. Namely, semantic reasoning of DL aims to check whether individual belongs to a concept, providing as results a fuzzy value that stands for the match degree. As we have just realized, the implementation of a proper mechanism of semantic matching is not easy and implies specific and

deepened considerations to evaluate aside. For this reason we consider it orthogonal to our biochemical tuple space model and we will not do further analysis on it.

2.2 Some examples

Here we present some useful cases with the aim to explain the concrete possibilities of the biochemical tuple spaces approach. We refer, similar as in previous chapter, to the pervasive scenario of airport display infrastructure in which user devices, displays and services have to be coordinated to present information related to user needs.

2.2.1 Services competition

Local competition

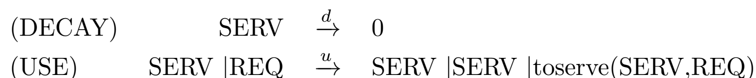
We start from considering a single tuple space, located in the node that hosts the display. The space acts as a coordination medium of services and users. As first purpose, we desire that less required services gradually disappear from the system, while services of interest emerge increasing their concentration. Such kinds of behaviour are possible activating competition dynamics between services, ruled by user requests. A possible interaction protocol could be:

1. a service provider inserts into the space a tuple service, specifying service id and the semantic description of its content;

2. a client puts a specific request;

3. tuple space has to bind service and request, considering some kind of semantic matching, and so creates a tuple toserve(service,request). This tuple is then read from service provider that computes the result and inserts it into the space by means of a tuple reply. Finally, the client retrieves reply ending the interaction.

The rules, that model this competition, are:



$$\begin{array}{llll} \text{(DECAY)} & \text{SERV} & \xrightarrow{d} & 0 \\ \text{(USE)} & \text{SERV}\ |\text{REQ} & \xrightarrow{u} & \text{SERV}\ |\text{SERV}\ |\text{toserve(SERV,REQ)} \end{array}$$

The rule (DECAY) decreases the concentration of a certain service that matches the reagent SERV, while rule (USE) aims to find a service and a request that matches respectively SERV and REQ, so retrieves them and creates a tuple toserve(SEV,REQ) increasing the concentration of the service considered. This positive feedback, in conjunction with the rule (DECAY), stands for simulate a prey-predator system ensuring to achieve a sort of struggle for survival between services.

Spatial competition

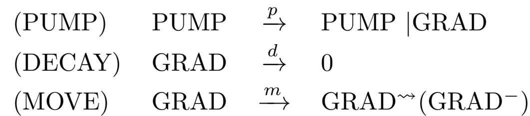
Now we consider a network of tuple spaces, to perform a spatial competition between services highlighting the context-dependent behaviour. We add to the previous set of rules, the next one:

$$\text{(DIFFUSE)} \quad \text{SERV} \xrightarrow{m} \text{SERV}^{\rightsquigarrow}$$

The rule (DIFFUSE) enables the spread of tuples from the source space to others, passing through neighbouring nodes. It models the concept of firing tuples previous described. This simple extension ensures an important spatial characteristic, i.e. services globally requested and useful can diffuse in the entire network while services with a local demand concentrate themselves only in a specific area. Moreover, this set of rules allows that if a new better service is added in a given node, it gradually replaces the old ones due to a better matching with *REQ* and so an increasing positive feedback.

2.2.2 Pattern based on gradient

Another interesting opportunity is represented by modelling spatial pattern based on the notion of computational gradient. It is a sort of force field induced, at the beginning, in a source node and then diffused around until each node settles its value in relation to the distance from the field source. This mechanism is useful to spread information from one node to its neighbours and iteratively to the entire environment, allowing also to make aware each space of the information diffusion path (backward node→source). Even now, we use the concept of firing tuples:



$$
\begin{array}{llcl}
\text{(PUMP)} & \text{PUMP} & \xrightarrow{p} & \text{PUMP} \mid \text{GRAD} \\
\text{(DECAY)} & \text{GRAD} & \xrightarrow{d} & 0 \\
\text{(MOVE)} & \text{GRAD} & \xrightarrow{m} & \text{GRAD}^{\leadsto}(\text{GRAD}^{-})
\end{array}
$$

The protocol requires to insert in the source node a token PUMP with concentration equal to one. This action leads it to start the generation of gradient GRAD, thanks to reaction (PUMP). This inflation is hindered by (DECAY) rule, so that the situation of saturation is avoided, making the gradient tends to an asymptotic limit. Finally, (MOVE) rule has the task of spread GRAD tuples towards neighbouring nodes, following, in the example, negative direction of the gradient, i.e. it tends to move where its concentration is lower. The gradient decreases its value in regard to the distance from the source because, progressively, the influence of pumping force is reduced and the decay rule has more effect. The point, beyond which the gradient is totally vanished, is called gradient horizon.

To show how to exploit patterns based on gradient, we consider the following typical scenario. A request, located in a node, pumps a gradient in order to obtain a specific service. Once the gradient arrives in the node that hosts the requested service, an answer is pumped, for a limited time, ascending the gradient until reaches the node source of the request. For achieving this behaviour, it is necessary to add to the previous three rules:

$$
\begin{array}{llcl}
\text{(ANSWER)} & \text{SERV} \,|\text{GRAD} & \xrightarrow{p} & \text{PUMP-A} \\
\text{(PUMP-A)} & \text{PUMP-A} & \xrightarrow{p} & \text{PUMP-A} \,|\text{ANSW} \\
\text{(DECAY-PA)} & \text{PUMP-A} & \xrightarrow{d'} & 0 \\
\text{(ASCENT)} & \text{ANSW} & \xrightarrow{m} & \text{ANSW}^{\rightsquigarrow}(\text{GRAD}^{+}) \\
\text{(DECAY-A)} & \text{ANSW} & \xrightarrow{d} & 0 \\
\text{(STOP)} & \text{ANSW} \,|\text{PUMP} & \xrightarrow{m} & 0
\end{array}
$$

The dynamics can be explained in detail as following. At a specific moment t_0 a request starts to pump a gradient(g), searching for a specific service(s). When g comes to the space containing s, here it is generated, thanks to (ANSWER), a pump-token that gets off the production of the reply ANSW, for (PUMP-A). The generation of pump-token goes on for a limited time, restricted by (DECAY-PA), after that the answer gradually fades-out for (DECAY-A). Meanwhile, ANSW ascends the gradient (ASCENT) until reaches request's source, where, interacts with the pump-token of the service, ending the production of the gradient (STOP).

2.3 Mapping on TuCSoN

LINDA appears as a suitable infrastructure to reify a system based on biochemical tuple spaces. LINDA's purpose is to coordinate the interaction among several parallel processes/agents, acting upon data that are stored as record with type fields, called tuples. A tuple space is a sort of repository of tuples, which is used as coordination medium to affect the behaviour of the external agents, supporting spatial and temporal uncoupling. This is possible thanks to the coordination language that allows an agent to insert a tuple through an *out* primitive, to retrieve it with *in* or to read it with *rd*. The primitives *in*/*rd* require as argument a tuple template, namely, a tuple with wildcards instead some of its argument. Basically, they have suspensive semantics, blocking their execution until a matching tuple is found. Anyway, their non-suspensive version is also provided, called *inp*/*rdp*, that simply fails if a matching tuple is not found at the first execution. TuCSoN

is a working technological platform based on LINDA model, providing in addition the possibility to specify the behaviour of a tuple space against some external or internal events. This programmable tuple spaces are called tuple centres and we can specify their actions by means of a meta-coordination language, defined through ReSpecT reactions. Its full description is treated in chapter three, here we just want to analyse if the notions and mechanisms provided by TuCSoN suffice to realize the biochemical tuple spaces model, and, otherwise, to propose a solution to fill the gap.

2.3.1 Model analysis

We will examine singularly the principal issues of the biochemical tuple spaces model [18], providing for each one specific evaluations and a potential mapping on TuCSoN platform.

Communication language

We start considering the foundations of the model, that is the communication language. The agents of the biochemical system can interact and coordinate themselves by means of tuples. At each tuples is associated a concentration value, proportional to its relevance in a given context. Concentration is dynamic and evolve over time due to the chemical laws.

Tuples in TuCSoN are simply logic atom, composed by a name and a list of arguments. So concentration cannot be naturally managed by TuCSoN, but it is necessary to plan some extension. For example, we can imagine to define a new entity, called *biotuple*, that, starting from TuCSoN tuple, wraps into itself both the tuple content and its concentration value. A possible syntax can be:

$$biotuple(\tau, n)$$

where, τ stands for the content of the biotuple, while n represents its concentration. For convenience, we can model τ simply as a TuCSoN tuple. It

means that the new entity has to handle properly only the adding concentration value. This extension has to be applied directly on source code of TuCSoN, to realize *biotuple* as basic abstraction. A first implementation, in pseudo-code, could be:

```
public class BioTuple{
    private LogicTuple ltuple;
    private long mult;
    public BioTUple(LogicTuple ltuple, int mult){
        this.ltuple = ltuple;
        this.mult = mult;
    }
    public void setLogicTuple(LogicTuple ltuple){
        this.ltuple = ltuple;
    }
    public void setMult(long mult){
        this.mult = mult;
    }
    public LogicTuple getLogicTuple()
        return ltuple;
    }
    public long getMult(){
        return mult;
    }
}
```

Following this solution, we can expand previous syntax as:

```
biotuple(tuple_name(arg1,arg2,...), #multiplicity)
```

The term *multiplicity* has unnecessarily the same meaning of concentration. With multiplicity we mean the quantity of a given tuple into a specific space. On the other hand, as we have stated, concentration expresses the level of activity/pertinence of a tuple into a local and limited environment. This evaluation can be applied simply considering the total tuple's quantity in the local space, and so multiplicity and concentration have the same meaning, or considering the tuple's quantity in relation to the total amount of other tuples. Last acceptation involves further computations but probably ensures better performance and a more interesting evolution of the system status. In this case, if we plan that each tuple has to contain the proper

value of concentration, will be necessary to update it at each insertion/removal operation, implying a considerable computational load. To avoid an excessive number of updates, we can think to maintain in each biotuple entity just the *multiplicity* value, leaving the correct evaluation of concentration to a specific function, used only when such value is required by the chemical engine to apply rules.

In the following we assume that there exists a matching function $\mu(\tau, \tau') \in [0, 1]$, returning 0 if τ and τ' do not match, 1 if they completely match, some internal value if they partially match.

Coordination language

The coordination language makes available a set of coordination primitives that allow the interaction between agents and tuple spaces. This set is similar to the one typical of LINDA and TuCSoN, but provided with the further function of managing tuples' concentration. So, *out* primitive has to specify the initial concentration of the inserted tuple, *in* can be either used to entirely remove a tuple (if no concentration is specified) or to decrease the concentration of an existing tuple, and so *rd* but without to remove anything. Their chemical equivalent actions are, respectively, to inject a chemical substance in a solution and to remove/observe a certain quantity of a substance.

The syntax for coordination primitives in biochemical tuple spaces model is the same as for TuCSoN primitives, but operational semantics is different[18]. We consider a DTMC model in which the execution of a primitive is represented as a transition with a specific probability(except *out* that fires outright). Its likelihood is given by the product between the degree of match $\mu(\tau, \tau')$ and, in case concentration is specified, the relative quantity of tuple required in relation to the total amount of a matching tuple. For example, given $\tau\langle n\rangle$, the tuple required with concentration n, and $\tau'\langle n+m\rangle$, a matching tuple with total amount $n+m$, a removal/reading operation, that involves these tuples, has a probability to fire computed as: $\frac{n+m}{n}\mu(\tau, \tau')$.

From this analysis we can assert that TuCSoN does not provide a suitable

mechanism to model biochemical primitives and to fill the gap should be planned a proper extension. In particular the work should be oriented in two directions. As first, should be realized a biochemical version of primitives in order to manage tuples' concentration. Then should be implemented a transition system, based on DTMC model, as previously described.

Topological structure

A pervasive computational system, built on biochemical tuple spaces model, has to present a well-defined topological structure. To simulate a complex biological environment, it is necessary to provide a network of tuple spaces, connected by means of neighbouring structures, that resemble biochemical compartments. Interaction between tuple spaces could follow, for example, the *linkability model* [22]. It can be achieved through a particular chemical law that takes one unit of some tuple and spreads it towards one of the neighbours, picked probabilistically.

This issue is feasible in TuCSoN quite naturally. Indeed, TuCSoN basically supports the definition of a distributed architecture (as the same name asserts: **Tu**ple **C**entres **S**pread over the **N**etwork). As we stated in the previous chapter, a tuple centre is uniquely identified by means of its name tname@netid:portno, that stands for a tuple centre tname located at node netid on port portno. So to define a neighbouring structure is sufficient to keep in each space the full name of tuple centres considered as "neighbours". Then, in order to communicate, it is just necessary to insert at the end of the name ?op, where op represents the primitive we want to excute on that node. The probabilistic choice of the neighbour could be performed through the uniform primitive urd available in TuCSoN (reads one tuple choosing it with uniform distributed probability among those that match the template). The proposed solution exploits the existent TuCSoN mechanisms and entails a certain delay due to the execution of uniform primitives. However, it is estimated to be a valuable approach because the diffusion of a tuple's unit involves only one uniform reading to pick the destination tuple centre and so

the overhead should be limited.

Chemical laws

The evolution of the system is governed by chemical laws (or reactions). Each tuple space is characterized by a proper set of reactions that affect tuples concentration over time in the same way as chemical substances evolve into chemical solutions. In order to simulate natural dynamics it is necessary to satisfy two requirements: the first is to define a set of laws that is inspired by bio patterns, the second is to provide a chemical engine that picks and executes the reactions simulating a real chemical system. This is possible following Gillespie's algorithm. In particular we can express the operational semantics of the execution of a chemical law as:

$$[\![[T_i \xrightarrow{r} T_o] | T | S]\!]_\sigma \xrightarrow{\mu(T_i,T)G(r,T_i,T|S)} [\![[T_i \xrightarrow{r} T_o] | T_o\{T_i/T\} | S]\!]_\sigma$$

This represents exactly the semantics described earlier. When reagents in T_i are found in the space, they are removed and replaced by the products in T_o considering the previous bindings. Generally, the matching between T and T_i provides more solutions and so in the evaluation of the markovian rate it is to count how many different combinations of tuples that match T_i, actually occur in S. This is considered in the factor $G(r, T_i, T|S) = r * count(T, T|S)$, where:

$$count(0, S) = 1$$
$$count(\tau\langle n\rangle \oplus T, \tau\langle m\rangle \oplus S) = \binom{m}{n} * count(T, S)$$

Clearly in TuCSoN a similar mechanism is not present. We have two ways to implement it. Initially, we can think about leaning on the existent tools and model a simulator programming tuple centres through ReSpecT reactions. This solution has the advantage of not entailing changes in the source code of TuCSoN but implies some problems during the implementation stage, in particular due to a problematic debugging. Moreover, since

ReSpecT reactions are used also for boot protocol and other general system operations, chemical laws, in such form, could lead to some tricky side-effects and to a low usability (users have to pay attention on using tuples that can compromise the working of the engine). Finally, it is a penalising approach, also, as regards performance, especially in the critical computation of global rate previous considerd. A possible solution, for this latter problem, could be to instantiate as many concrete laws as the possible combinations of matching templates/tuples, to calculate for each one its markovian rate and then to apply one of them probabilistically.

The other alternative is to implement a chemical engine acting directly on source code of TuCSoN, following the previous observations. In this way the simulator becomes a core abstraction of the model, offering better performance, clear separation of functionality and a good usability. On the other side, it is to consider some potential problems concerning time required for the realization and integration/compatibility issues.

Chapter 3

TuCSoN & ReSpecT

In this chapter we want to describe the technologies to which we have referred until now and which we will take as foundation to realize a first implementation of a biochemical tuple space, that are TuCSoN and ReSpecT. After we have discussed TuCSoN model and architecture, we will focus on ReSpecT from the point of view of its language and of its engine's working principles.

3.1 TuCSoN

TuCSoN (**Tu**ple **C**entres **S**pread **o**ver the **N**etwork) is a general purpose agent-oriented model and infrastructure for Multi-Agent System (MAS) coordination. TuCSoN is based on a coordination model providing *tuple centres* as first-class abstractions to design and develop general purpose coordination artifacts. Tuple centres are programmed through the ReSpecT logic-based specification language [13].

Agents can interact with tuple centres and coordinate themselves by exchanging tuples through a LINDA-like set of coordination primitives. This approach presents three key features, useful to handle a pervasive scenario: generative communication, associative access and suspensive semantics. *Generative communication* means that the information inserted into the space have an independent life with respect to the generator. It allows agents to be

uncoupled in space, time and name, i.e interacting agents have not necessary to know each other, to coexist in the same space or at the same time in order to communicate. *Associative access* means that access to information is based on tuple matching, considering their structure and content rather than their location or name. This characteristic leads to a sort of data-driven coordination allowing, potentially, to define knowledge-based coordination pattern. At last, *suspensive semantics* promotes coordination pattern based on knowledge availability, coping well with the issue of incomplete or partial knowledge typical of system continuously under evolution.

In the following, we will illustrate by means of what model and architecture TuCSoN allows these interesting features.

3.1.1 Model & Language

A TuCSoN system is a collection of TuCSoN agents that interact with ReSpecT tuple centres (the coordination media), located in a set of nodes potentially distributed over the network. Agents act as proactive entities that coordinate themselves by means of reactive entities (tuple centres). They are composed of two different spaces: a shared space for communication based on tuples (*tuple space*) and a *specification space* that contains the programmable logic of the related tuple centre.

Each tuple centre can be univocally identified, within the entire system, through their full name: tname@netid:portno. It locates a tuple centre tname (can be any Prolog-like first-order logic ground term) available on node netid:portno, where netid stands for the IP number of the DNS entry of the device hosting the node and portno is the port number where TuCSoN coordination service listens the invocations for the execution of coordination primitives.

The interaction between agents and tuple centres occurs through a specific *coordination language* executing *coordination operations*. In turn, they rest on TuCSoN *communication language* that includes tuple language and tuple template language. Since the TuCSoN coordination medium is the logic-

based ReSpectT tuple centre, both languages are logic-based allowing as instances any first-order logic Prolog atom.

Each coordination operation is composed of two stages: first an agent requires an operation on a specific tuple centre target (*invocation*), then, after the tuple centre has computed the result, it replies to the agent including all the information concerning the execution of the required primitive. The syntax to invoke an operation `op` on a tuple centre `tname@netid:portno` is: `tname@netid:portno?op`. In this way it is possible to invoke a primitive also on external tuple centres (i.e not in the same node of the agent). In particular, TuCSoN provides nine basic coordination primitives[16], that are [1]:

out(*Tuple*) writes *Tuple* in the target tuple space; after the operation is successfully executed, *Tuple* is returned as a completion;

rd(*TupleTemplate*) looks for a tuple matching *TupleTemplate* in the target tuple space; if a matching *Tuple* is found when the operation is served, the execution succeeds by returning *Tuple*; otherwise, the execution is suspended to be resumed and successfully completed when a matching *Tuple* will be finally found in and returned from the target tuple space;

in(*TupleTemplate*) looks for a tuple matching *TupleTemplate* in the target tuple space; if a matching *Tuple* is found when the operation is served, the execution succeeds by removing and returning *Tuple* ; otherwise, the execution is suspended to be resumed and successfully completed when a matching *Tuple* will be finally found in, removed and returned from the target tuple space;

rdp(*TupleTemplate*) predicative (non-suspensive) version of *rd*(*TupleTemplate*); if a matching *Tuple* is not found, the execution fails (operation outcome is FAILURE) and *TupleTemplate* is returned;

[1] *Tuple* belongs to tuple language, while *TupleTemplate* belongs to tuple template language

inp(*TupleTemplate*) predicative (non-suspensive) version of *in*(*TupleTemplate*); if a matching *Tuple* is not found, the execution fails, no tuple is removed from the target tuple space and *TupleTemplate* is returned;

no(*TupleTemplate*) looks for a *Tuple* matching *TupleTemplate* in the target tuple space; if no matching tuple is found when the operation is served, the execution succeeds, and *TupleTemplate* is returned; otherwise, the execution is suspended to be resumed and successfully completed when no matching tuples can any longer be found in the target tuple space, then *TupleTemplate* is returned;

nop(*TupleTemplate*) predicative version of *no*(*TupleTemplate*); if a matching *Tuple* is found the execution fails and *Tuple* is returned;

get reads all the *Tuples* in the target tuple space and returns them as a list; if no tuple occurs in the target tuple space at execution time, the empty list is returned and the execution succeeds anyway;

set(*Tuples*) overwrites the target tuple spaces with the list of *Tuples*; when the execution is completed, the list of *Tuples* is successfully returned;

Later, the necessity to manage more than one tuple with a single primitive and with good performance leads to the supplement of **bulk primitives**: *out_all*, *rd_all*, *in_all*, *no_all*. Substantially, they return, not one tuple, but all the tuples that match the template; otherwise, if no tuples match, will be returned the empty list (bulk primitives never fail).

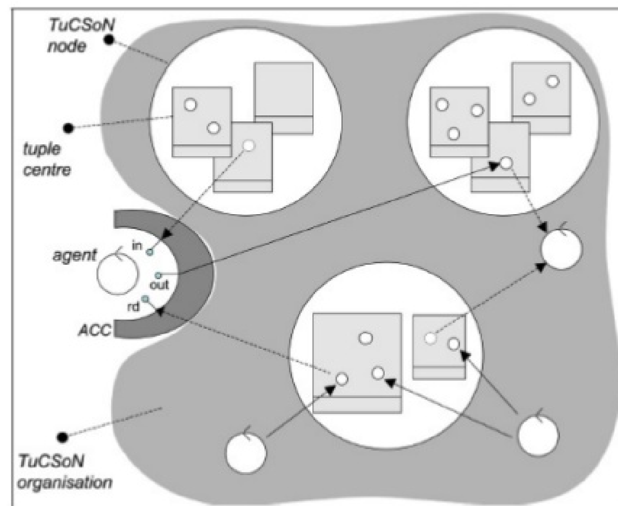
Another important extension is represented by **uniform primitives**: *urd*, *uin*, *urdp*, *uinp*, *uno*, *unop*. Their purpose is to insert a probabilistic behaviour within agents' coordination. In particular, they replace the don't care non-determinism of LINDA-like primitives with a uniform probability distribution. This set of primitives is important especially as potentially allowing to model stochastic behaviour of some nature-inspired patterns.

Finally, a further useful primitive is *spawn*. It deals with activating some kind of computation Java or Prolog, local to the tuple centre where it is in-

voked. This operation presents a non-suspensive semantics starting a parallel computational activity that is carried out asynchronously w.r.t the caller. It has two arguments, that are the activity to perform (a Prolog atom including Prolog theory and goal or a Java class) and the identifier of the tuple centre where to execute it. Through such primitive it is possible to handle complex computational activities related to coordination, managing them by means of a standard sequential computation instead of a sequence of time-consuming coordination primitives.

3.1.2 Architecture

A TuCSoN system is a collection of TuCSoN nodes that host TuCSoN services. Each node is characterized by a network device (netid) and by a network port (portno) where the service listens to incoming requests. This solution allows the presence of multiple nodes on a single device, as long as each one is listening on a different port.



A TuCSoN agent has at any time the possibility of invoking a primitive on any tuple centre available on the network through: tname@netid:portno?op. So the TuCSoN *global coordination space* is composed by all the tuple centres located in all the nodes of the system. On the other side, the TuCSoN *local*

coordination space for a network device netid is defined at any time as the set of all the tuple centres made available by all the nodes hosted by this device. Each node defines a default tuple centre (called default) and a default port (20504). If the network device's address is not specified, the execution of the primitive refers to the local coordination space. For example, if it is specified only op, this primitive is invoked on default tuple centre of local node on port 20504.

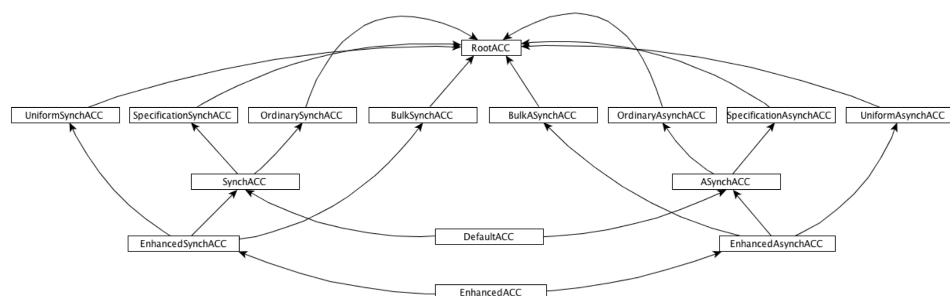


Figure 3.1: Overall view over TuCSoN ACCs

From the point of view of agents, the idea is to structure tuple centres in organisation and govern the access to them associating at each agent specific role. Thus, an agent can perform only a limited number of actions that are allowed to him. This solution is based on the *Role-Based Access Control* (RBAC) model and requires to provide a special tuple centre in which to maintain RBAC rules. In order to model such hierarchy of agents we refer to the notion of *Agent Coordination Context* (ACC). It is a runtime and stateful interface that is used by agents to invoke primitives on tuple centres of a specific organisation. Substantially, ACCs rules the interaction between agents and tuple centres; actually TuCSoN stands, only, at a first stage towards the full implementation of this approach. In particular, TuCSoN provides three basic ACC each one both in synchronous and asynchronous version, that enable the usage respectively of the three principal sets of primitives, i.e. Ordinary ACCs, Bulk ACCs and Uniform ACCs. In the synchronous version the agent invokes the primitive and then blocks waiting for its completion;

on the other hand, in the asynchronous version, the agent, after invocation, does not block but is asynchronously notified of the operation's completion. However, TuCSoN presents other ACCs that enable access to ReSpecT specification and other useful combinations. The overall view over TuCSoN ACCs is showed in the figure above.

3.1.3 Programming tuple centres

A tuple centre is a tuple space enhanced with the possibility to program its behaviour in response to some external or internal events. An agent can define it through the TuCSoN meta-coordination language and by executing meta-coordination primitives. Similarly as before, meta-coordination language is composed by specification language and specification template language that actually coincide, since TuCSoN coordination medium is the logic-based ReSpecT tuple centre. Admissible tuples for these languages are expressed through the syntax: reaction(E, G, R) $\in$ specification language and reaction(ET, GT, RT) $\in$ specification template language, where we consider E, G, T as Prolog term ground(without variables) and ET, GT, RT as Prolog term containing potentially some variables. Any TuCSoN meta-coordination operation is invoked by a source agent on a target tuple centre, to which is delegated its execution. Syntax and phases for invocation are the same as for the coordination operations. There are 9 meta-coordination primitives, that perfectly match the 9 basic coordination primitives: *out_s*, *rd_s*, etc. This ensures an uniform access both to tuple space and to the specification space in a TuCSoN tuple centre. But to really understand the behaviour of tuple centre against a given specification, we have to analyse ReSpecT language.

3.2 ReSpecT

ReSpecT (**R**eaction **Spec**ification **T**uples) is a logic-based coordination language that enables tuple space programming, actually distributed as a part of TuCSoN middleware. It has a twofold nature: it allows to associate events to

reactions (specification language) and to execute them as local computations (reaction language). We will illustrate both roles in the following.

3.2.1 ReSpecT language

As specification language, ReSpecT allows event to be declaratively associated to reactions by means of specification tuples of kind: reaction(E, G, R). Its semantics is: given a ReSpecT event Ev, a specification tuple $reaction(E, G, R)$ asociates a reaction $R\theta$ to Ev if and only if $\theta = mgu(E, E_v)$ and a guard predicate G is true.

In particular, event E stands for any TuCSoN primitive (except for get_s and set_s), guard G represents a logical condition that has to be satisfied (quod vide [13], Table 5) while reaction R can be any TuCSoN primitive (as before) or any Prolog computation or any combinations of the two.

3.2.2 ReSpectVM

As reaction language, ReSpecT provides a support to the execution of reactions. Here, we want to describe its internal behaviour and architecture to understand how tuple centres actually work. This analysis is important because in the next chapter we will present a biochemical extension of TuCSoN & ReSpecT, that deals, also, with the ReSpecT engine.

First of all, we report the informal semantics of ReSpecT virtual machine based on paper [13]. The main cycle of a tuple centre works as follow. Whenever the invocation of a tuple centre primitive by either an agent or a tuple centre is performed, an (admissible) ReSpecT event is generated, and reaches its (the primitive) target tuple centre, where it is automatically and orderly inserted in its InQ queue. When the tuple centre is idle (that is, no reaction is currently being executed), the first event ϵ in InQ (according to a FIFO policy) is moved to the multiset Op of the requests to be served: this stage is called the *invocation* phase of the event ϵ. Consequently, reactions to the invocation phase of ϵ are triggered by adding them to the multiset Re of the

triggered reactions waiting to be executed.

All triggered reactions in *Re* are then executed in a non-deterministic order. Each reaction is executed sequentially, with a transactional semantics, and may trigger further reactions, again to be added to *Re*, as well as new output events representing link invocations: such events are added to the multiset *Out* of the outgoing events, and then moved to the tuple-centre outgoing queue *OutQ* at the end of the reaction execution, if and only if successful.

Only when *Re* is finally empty, requests waiting to be served in *Op* are possibly executed by the tuple centre, and operation/link completions are sent back to invokers. This may give rise to further reactions, associated to the *completion* phase of the original invocation, and executed again with the same semantics specified above for the invocation phase. Thus, the main cycle of an ReSpecT tuple centre is finally concluded.

ReSpecT state machine

Let's see how the ReSpecT engine manages such events and queues at software level.

We start from RespectVM class, cornerstone of the entire process. It is a thread in charge of listening new events generated against some agents' request. Its principal role is to ensure the continuous and cyclic behaviour of the ReSpecT engine, inserting the generated events into the *InQ* queue.

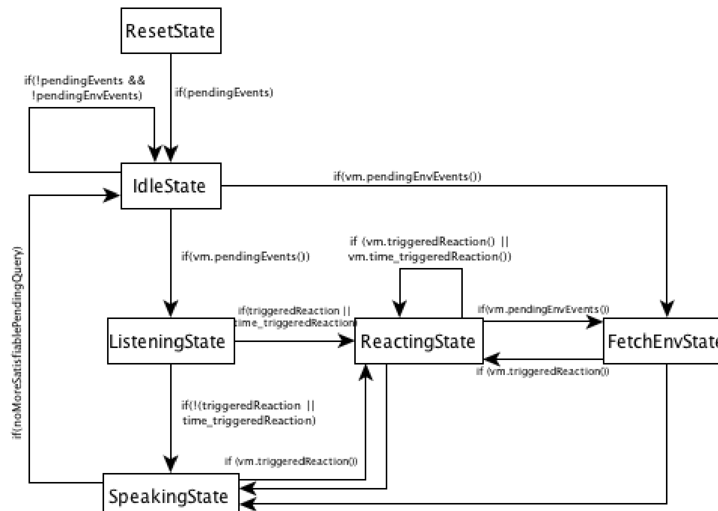
The operations expressed before in the description of the informal semantics are reified by means of a sort of state machine. It is composed by six states, that are:

- ResetState: represents the initial state, considered only at the system boot;

- IdelState: represents the condition in which there are not any events to manage or reaction to perform, it is called idle state of the system;

- ListeningState: at this point the system is waiting for an event coming from an external agents;
- ReactingState: state reached when there are some triggerable reactions;
- FetchEnvState: state reached when the system perceives some environmental events;
- SpeakingState: fundamental state in charge of managing the execution of primitives requested by agents.

At each state corresponds an namesake class that implements its behaviour. Here below we present a diagram that shows clearly the structure of the ReSpecT state machine.



Another key class is represented by RespctVMContex that deals with the low-level interfacing of TuCSoN and ReSpecT operations. It has two principal tasks. The first is to actually manage the insertion/removal/reading of tuples in the tuple space. The second is to make available the functionalities to verify whether the current primitive leads to generate an event that implies the firing of some reaction (with a matching event triggering). In chapter

four we describe the work made on this class to achieve a bio extension for TuCSoN and ReSpecT.

If at least one reaction is found, after evaluating that the predicate *guard* is satisfied, the reaction is performed considering the association variables/-values defined in the previous matching(between primitive's event and reaction's event). The real execution of the reaction's body is delegated partly to the Prolog engine, as regards Prolog computations, and partly to the class Respect2PLibrary that represents a TuProlog library through which defines the behaviour of ReSpecT primitives.

Chapter 4

Biochemical TuCSoN

4.1 Motivations

As we have seen before, TuCSoN is an interesting technology that, potentially, allows to realize several applications in different scenarios. One of these is biochemical coordination. In particular, TuCSoN lends itself to model nature-inspired patterns, thanks to its architecture characterized by programmable tuple centres. Agents can coordinate themselves and interact uncoupled in space and time, leaving information in the tuple space. Tuple centres can be programmed with a specific behaviour against some internal or external events. Moreover, support to distributed communication is intrinsically provided, allowing interaction between tuple spaces located in different nodes of the system. These characteristics make TuCSoN very similar to the natural environment where organisms live, interact and organise themselves, achieving global complex patterns, from simple individual behaviour (for example ACO pattern).

Taking inspiration from the considerations of previous chapters, we want to define a biochemical version of TuCSoN, extending the original one with abstractions and mechanisms specific to biochemical tuple spaces. In particular, the aim is to realize an independent release, focusing on architectural aspects such as maintainability and extensibility.

4.2 Architectural requirements

Before explaining the way adopted to implement BioTuCSoN, it is important to underline some aspects considered during the work process.

First of all, we have to point out that TuCSoN, for its academic nature, is a software system continuously under evolution. In theory, every day some bugs could be fixed, new features could be added or some mechanisms changed. Potential extensions would have to consider this issue in order to simplify the work of developers, i.e. reduce as much as possible the time required to carry the changes from the standard version to the new one. For these reasons, in the design of bio extension, we analysed the different alternatives and chose those that minimize variance by the approach of original TuCSoN, exploiting as much as possible the current mechanisms. This is what we mean with the term *maintainability*.

In the same way, the requirement of *extensibility* is important. The work we are going to describe, is only a first step toward the full BioTuCSoN implementation. Biochemical tuple spaces are complex systems that require several new aspects and abstractions to be managed [18].

Firstly, the principal concept to introduce is *concentration*. It is a characteristic value of each tuple that expresses its level of pertinence/activity. This information is essential to execute uniform primitives such as *uin*, *urd*, etc, that select probabilistically the matching tuple according to its concentration. In theory, selection probability should be affected also by the degree of match between template and tuple, a fact that implies semantic aspects. The presence of concentration cause to modify the behaviour of every TuCSoN primitive, named *bio primitive* in BioTuCSoN, aiming to allow them to manage this value in a proper way.

In the second place, we have to consider also *topological* issues such as linking neighbouring tuple spaces, retrieving information about reachability of neighbours, sending tuples between different spaces and update tuple properties in relation to their moves.

Finally, an aspect, that is fundamental to ensuring the working principle

of biochemical coordination, is the *chemical engine* (or chemical simulator). As seen in the initial chapters, it is an algorithm that cyclically retrieves, in a stochastic way, the coordination rule to perform, computing probabilistically even the time interval between steps. A solution could be to refer to Gillespie's algorithm, implementing it as a Java function integrated into TuCSoN code.

Principally, our work deals with analysing, defining and implementing basic aspects of biochemical tuple spaces, i.e. concepts of *bio tuple* and *bio primitive*. Then, wanting to ensure the programmability of bio tuple space, we plan and realize some changes to ReSpecT engine (*BioReSpecT*). For now, topological aspects are considered realizable through the current TuCSoN mechanisms, while chemical engine is not considered. Indeed, we want to prove whether exploiting the features of this first version of BioTuCSoN & BioReSpecT, we can model and obtain some interesting system properties, including especially self-organisation and situatedness.

4.3 Bio extension foundations

Now we illustrate the process that brings us to define and implement the two principal abstractions of BioTuCSoN: bio tuples and bio primitives. We present the choices we took, explaining the reasons, the advantages and eventually the disadvantages involved.

4.3.1 Bio tuple

In this section we show how we worked to introduce bio tuples abstraction, that is, namely, the communication language of BioTuCSoN.

Similarly as TuCSoN, BioTuCSoN communication language is, theoretically, composed by *bio tuple language* and *bio tuple template language*, both logic-based. The difference is that while a *bio tuple* must be completely specified (i.e ground Prolog atom, with a defined concentration value), a *bio template* can be partially specified (i.e. Prolog atom can contain variables

and/or concentration value can be not specified). Although these are different characteristics we choose to define only one class that models both languages. This simplifying solution leads to only one setback, that is to check that *bio out* has a bio tuple (and not a bio template) as argument.

As already stated, *bio tuple* is a standard tuple enriched with a value, called *concentration*(or multiplicity), that specifies the level of its relevance in a given context. TuCSoN models tuples with LogicTuple class into package alice.logictuple, so now we focus on this part of code. The easiest way to introduce this abstraction is to define an entity that integrates, into itself, LogicTuple and a numerical value representing the concentration. It is a positive integer (different from zero) that can reach high values in most scenarios. Consequently, we can plan to define a class, naming it BioTuple, that has two class fields, one of type LogicTuple and the other of type long representing multiplicity. Now, we have two alternatives.

The first is to let BioTuple to implement the generic interface Tuple (also implemented by LogicTuple). In theory, this appears a possible and coherent solution because BioTuple and LogicTuple can be seen as different realizations of Tuple interface, each one with specific characteristics. However, this way leads to some practical problems at implementation time: Tuple is actually a fictitious interface, without any declaration of fields or methods. Moreover, in TuCSoN code every operation over tuples requires as input parameter, or returned value, objects of type LogicTuple, and not of type Tuple as we could suppose. So if we want to follow this way, we have to replace every occurrence of LogicTuple with BioTuple, a trivial operation but time consuming and not so efficient. As an alternative, we can define correctly Tuple interface using it in place of LogicTuple. But even this expedient does not resolve the problem of previous substitutions. Both described solutions have then a substantial incoherence with the architectural requirements expressed before. To satisfy the maintainability property it is important to modify as little as possible the structure of TuCSoN code while the previous ways need a lot of small changes.

The second alternative, the chosen one, is to realize a BioTuple class as an extension of LogicTuple. In this way, a bio tuple is seen as a standard tuple with, in addition the concentration value, exactly as it was described before. Its implementation can be simplified by delegating to the inherited methods some basic operations and defining from scratch only the characterizing aspects. TuCSoN code is affected in a minimal way because, wherever it is specified a LogicTuple, we can pass an instance of BioTuple thanks to polymorphism. So we can exploit the most of existent TuCSoN mechanisms. Specific controls of type (through instaceof) and explicit casts are then needed at the time to choose the actual operation(bio or standard primitive) to execute. Shown below is an UML diagram that represents the adopted solution, highlighting the overridden methods by BioTuple class.

BioTuple constructors exploit functions of super class to set in the right way the content of the bio tuple, that is at last a Prolog term. It can be passed in several forms such as functor and arguments (String, TupleArgument,...) or without functor name or simply as a Prolog term (Term). The only new task of bio tuple constructors is to set the specific multiplicity value (long mult) of that instance.

Now we have to clarify one important detail. The concept of *multiplicity* is similar but not the same as *concentration*. In the first case, we mean the overall amount of one specific bio tuple into the space. Concentration is a more complicated aspect that implies spatial consideration, i.e it is a measure of pertinence/activity of a tuple in a given context obtained computing the ratio between its multiplicity and the sum of the quantity of all tuples into the considered space. The introduction of the concept of concentration, in its full meaning, is relevant most of all for chemical engine, and so it is left to future works. In the following we use multiplicity and concentration with the same intent.

Each bio constructor throws an InvalidMulitplicityException, specifically created, in case that the multiplicity argument is equal or less than zero.

The unique constructor noteworthy is BioTuple(Term): its roles is to create a bio tuple, retrieving information from only a Term. This means that there is a necessary sort of parsing stage to evaluate if the term is defined coherently according with the structure of a bio tuple and then to extract the information about tuple content and its multiplicity.

```
public BioTuple(Term t) throws InvalidMultiplicityException{
  Struct s_t = (Struct)t.getTerm();
  if(s_t.getName().equals("biotuple") &&
        s_t.getArity()==2 && s_t.getArg(1).isGround()){
    info = new TupleArgument(s_t.getArg(0));
    long m = ((Number)s_t.getArg(1).getTerm()).longValue();
    if(m<=0)
      throw new InvalidMultiplicityException();
    this.mult = m;
  }else if(s_t.getName().equals("biotuple") &&
        s_t.getArity()==2 && !s_t.getArg(1).isGround()){
    info = new TupleArgument(s_t.getArg(0));
```

```
    }else{
      throw new InvalidMultiplicityException();
    }
}
```

Not originally planned, this constructor is defined during the work process because it is useful in several situations. In particular, it is used in three strategical classes:

- TupleSet : class that wraps the business logic of every primitive and the basic tools of TuCSoN infrastructure;
- Tucson2PLibrary : a tuProlog library that makes available TuCSoN primitves also to tuProlog agents;
- Respect2PLibrary : a tuProlog library that defines the behaviour of ReSpecT primitives, used inside ReSpecT virtual machine.

In every previous case, the utility of this constructor is to let developers to manage bio tuples, building as they were logic tuples, and so promote the concept of maintainability.

Other than constructors, we have to add some methods. First of all, we add setMultiplicity(long m) and getMultiplicity(), useful to set and retrieve the concentration value of a given bio tuple instance. To show bio tuples in a string format we redefine toString(), representing it as:

$$biotuple(\langle LogicTuple\rangle, \langle \#multiplicity\rangle).$$

Again, to convert a bio tuple into a Prolog term we realize specific methods: toTerm(), overriding the inherited one, and toTerm(long), used in TupleSet and defined with the aim to promote maintainability, ensuring to manage bio tuple similarly as logic tuple. Specifically, the second method returns a bio tuple as a Prolog term setting its multiplicity at the specified value.

Then we define two static methods parse(String,long) and parse(String) to allow developers to create a bio tuple simply passing the Prolog term as

string and multiplicity as long value. The second one is useful when we want to define a bio tuple template, without ground concentration value. They are principally used by TucsonAgent (programmed with Java) to define bio tuples. Now we present some examples. The first deals with the building of a bio tuple ground, i.e without any variable:

```
//biotuple(hello(world),8)
BioTuple tuple = BioTuple.parse("hello(world)", 8);
```

Then, we show how to create a bio tuple template with variable argument and variable concentration:

```
//biotuple(hello(X),0)
BioTuple template = BioTuple.parse("hello(X)");
```

When, in the string representation, the multiplicity field assumes the value of zero, this means it is not specified and so is considered variable. As we will see later, a variable multiplicity can be used in all primitives, except for *out* (namely in every retrieving/reading primitive), implying that concentration is ignored when matching between tuple and template is performed. The constructor BioTuple(TupleArgument,Long) allows to manage both situations: if multiplicity is not specified, the second parameter is set to null, otherwise, it is set to the given value.

Even parseCLI(String tuple) was not planned at the first design of BioTuple. This function is used in CLIAgent class to parse the string obtained by the command line returning the relative BioTuple, always with the aim to ensure a similar management of bio tuples and logic tuples, .

Finally, isMultGround() is a simple method that checks whether or not the concentration value is set.

4.3.2 TuCSoN code analysis

Before explaining the bio updates from the point of view of requirements (semantics), design (architectural choices) and implementation (technical details), we point out some interesting aspects of TuCSoN behaviour, to better

understand the operations flow, critical sections and, generally, the context we are going to consider.

Communication dynamics

At an high abstraction level, we can see the execution of a TuCSoN primitive as follow: a *TuCSoN agent*, located in a specific network node, makes a request that is accepted by a *TuCSoN tuple centre*, situated in the same or in a different network node. It performs the required operations and then gives back the result to the caller. These two entities have a dual behaviour. To make request TuCSoN agent invokes the *function* corresponding to the desired primitive and then waits for the reply from the contacted tuple centre through a *control thread*. On the other side, TuCSoN tuple centre waits for requests from agents through a specific *thread*; once come, it invokes proper *functions* to compute the result and finally returning it.

As already mentioned, agent and tuple centre could be situated in different spatial location, so their interaction occurs through the network. The protocol followed to establish a connection between an agent A and a tuple centre T is:

1. As first step, A must authenticate itself and overcome a security protocol (for now only theoretical); if it succeeds, an entity is created called ACCProxyAgentSide, which is delegated the communication tasks. The authentication is valid at a system level, i.e. it is necessary only the first time an agent joins the system making the first request.

2. The initial stage expects that ACCProxyAgentSide communicates with a node-side thread called WelcomeAgent which waits for requests to pass on to another class named ACCProvider. This entity analyses request's type, sender and content. In case everything is fine and the request is not an exception, it creates an ACCProxyNodeSide, dual node-side entity to ACCProxyAgentSide, in charge of supporting the communication with A. In this stage, it is also established the session in which the following

interaction between ACCProxyAgentSide and ACCProxyNodeSide takes place.

3. Real comunication between A and T happens actually through the entities ACCProxyAgentSide and ACCProxyNodeSide into a specific session. If A wants to interact with other tuple centres, he leans on the same ACCProxyAgentSide, repeating from the communication with WelcomAgent and maintaining one session for each tuple centre contacted.

As regards the internal behaviour of a TuCSoN node, synthetically, it has to check whether or not an operation has as target itself. A negative answer implies that the operation is passed on the specified tuple centre target. In the other case, operation is inserted into a input event queue that will later be scanned by ReSpecT engine to extract events to handle. Then, the computed result is put into an output event queue, from which ACCProxyNodeSide takes the information through which it builds the reply message. These two buffers allow tuple centre to manage more than one suspensive primitives at time.

Operations flow

Now we analyse the operations flow necessary to perform a TuCSoN primitive, considering the connection between agent and tuple centre has been already established. That is, the observation of interaction between ACCProxyAgentSide and ACCProxyNodeSide. It is useful to explain the context in which we apply changes.

From agent-side, the request for the execution of a specific TuCSoN primitive is performed by sending a message (TucsonMsgRequest) towards a defined tuple centre target. This message is built starting from a specific instance of TucsonOperation class related to the required primitive. The principal operations that lead to message dispatch is confined in the method doOperation into the ACCProxyAgentSide. At this stage, the agent waits for a reply message (TucsonMsgReply) from the tuple centre, through a control thread instance of Controller internal class. Once the reply arrives, the control thread checks

if the execution ends well and if the information obtained are coherent with the request. If so, it signals the correct termination to ACCProxyAgentSide thread, which has suspended waiting for that. Finally, the latter returns the operation result to the agent.

From node-side, the operations flow are more complex. We can subdivide the primitive management into two principal activities reify with as many threads:

- ACCProxyNodeSide supervises and implements communication protocol to interact with other system subject;
- RespectVM models ReSpecT engine that actually executes the primitives.

ACCProxyNodeSide waits requests from agents/tuple centres cyclically reading from the input stream. Once they arrive, the thread checks the type of request and invokes an appropriate function to insert into a input event queue the related event. We can outline the classes involved in this flow as following:

$$ACCProxyNodeSide \rightarrow TupleCentreContainer \rightarrow$$
$$OrdinaryAsynchInterface \rightarrow RespectTC \rightarrow RespectVMContext$$

On the other side, RespectVM thread merely fires cyclically (after two consecutive calls) execute() method into SpeakingState class. Its fundamental role is to extract, if there are, one by one events from input buffer and depending on the type of specified operation performs the functions required to get results. These, again, call methods of RespectVMContext class, that in turn exploits functions of TupleSet. The latter is the topic class where the business logic is confined, i.e. the code that defines the basic behaviour of all primitives. Principally we operate here to realize "bio changes". As before, we show the class involved synthetically:

$$RespectVM \rightarrow TupleCentreVMContext \rightarrow SpeakingState \rightarrow$$
$$RespectVMContext \rightarrow TupleSet$$

4.3.3 Bio primitives

Now we want to describe the updating process towards bio primitives. The semantic we consider for them is drawn from [18] with some differences. There all retriving/reading primitives are divided into two types, one managing template with ground multiplicity (*in*, *rd*, ...) and one template with multiplicity as variable (*inv*, *rdv*, ...). Moreover, all of them have a probabilistic behaviour influenced by concentration values. We consider a different structure. To ensure one-one mapping with TuCSoN and so a better system maintainability, we merged into a single primitive the management of the different template types and planned bio extension of the most of TuCSoN primitives, following TuCSoN name conventions. That means we defined also bio primitives with non-probabilistic behaviour. Finally, it is not taken into account any semantic match issue and, for the moment, every primitive fires instantaneously.

We achieve bio extension of all synchronous primitives except *get*, *set* and the bulk ones, without considering timeout. For their realization we followed the same logic of original primitives and left unchanged as much as possible TuCSoN structure, with the aim to respect the previous architectural requirements. Thanks to have modelled BioTuple as extension of LogicTuple, we could exploit the existent mechanisms, using polymorphism, and so confine the changes only to the business logic, i.e. to the code that actually implement the behaviour of the primitives.

We want to realize an indipendent BioTuCSoN version, that means that a user can only handles bio abstractions. For this reason, instead of defining a new interface for bio primitives, we modify the existing one, so that its functions required as arguments no more LogicTuple but BioTuple. In particular, we act on OrdinarySynchACC and UniformSynchACC. These simple changes, that reverberate on ACCProxyAgentSide, allow bio primitives to be managed at agent-side.

On node-side, as we have seen before, we were able to relegate the changes only at the final stage of the operations flow; more specifically into the Re-

spectVMContext and TupleSet classes. These aspects will be analysed carefully for each primitives.

In the following we expose singularly the fundamental bio primitives in which we make substantial design choices, while the others, whose behaviour were derived by the previous one, will be described shortly. We performed one-one mapping between primitives and bio primitives, so their name does not change. Since that, from this point forward we consider only bio TuCSoN extension, and so when we talk about tuples or primitives, we actually refer to bio tuples and bio primitives.

Bio out

It is shown below a *bio out* example, through TuCSoN agent code.

```
long mult = 5;
BioTuple tuple = BioTuple.parse("test(bio)",mult);
ITucsonOperation op = acc.out(tid, tuple, null);
```

Bio out: writes tuple in the target tuple space tid; if in tid there are a bio tuple that matches tuple, merges them together summing up their concentration values. Third parameter (null), representing timeout, is ignored for the moment.

Observations. At design time we evaluated some aspects. In standard TuCSoN, tuple space is reified, in TupleSet class, by a LinkedList. Supposing to insert, here, also bio tuples we will have in the same data structure instances of two different kinds, a fact that leads to a bit confusion when we want to retrieve them. Indeed, we should provide controls of type at each list reading, modifying RespectVMContext class. Moreover *bio out* needs additional checks compared with standard *out* and so it is impossible to unify the code of this two operations. To overcome these problems, we defined a new LinkedList⟨BioTuple⟩, named bioTuples, splitting clearly standard space from bio space. Due to this division, we have to update TupleSet inserting further control methods, i.e. all the necessary methods to acquire information about

the status of the two distinct spaces. So for example, we have to implement isEmptyLogicSet() and isEmptyBioSet(), and so on.

As for bio out primitive, the changes are confined into add(LogicTuple t) method of TupleSet, so that RespectVMContext remains unchanged. In this method we plan, as first operation, to check the instance type through instanceof command to choose between *bio out* code or *out* code[1]. Essentially, more than standard version, *bio out* has to update concentration values considering the merging with an eventual matching tuple. *Bio out* needs as argument a ground tuple with ground multiplicity. Otherwise the request is not performed by agent, blocked thorough a specific control at invocation time. Here below, we show the code that models bio out behaviour.

```
public void add(LogicTuple t){
  if(t instanceof BioTuple){
    BioTuple bioT = (BioTuple) t;
    if(bioTuples.size()==0)
      bioTuples.add(bioT);
    else{
      ListIterator<BioTuple> l=bioTuples.listIterator();
      while (l.hasNext()){
        BioTuple tu=l.next();
        if (bioT.match(tu)){
          l.remove();
          long oldValue = tu.getMultiplicity();
          try {
            tu.setMultiplicity(oldValue + (bioT).getMultiplicity());
          }catch (InvalidMultiplicityException e) {
            e.printStackTrace();
          }
          l.add(tu);
          return;
        }
      }
      bioTuples.add(bioT);
      if (transaction)
        bioTAdded.add(bioT);
    }
  }else{
    tuples.add(t);
    if (transaction)
```

[1]It is necessary to preserve works of standard primitives because TuCSoN infrastructure starts using LogicTuple.

```
      tAdded.add(t);
  }
}
```

Controls over transaction field are necessary to ensure the transactional behaviour of ReSpecT reactions, as we will see later.

Bio in

Here we show two *bio in* examples, through TuCSoN agent code. The first deals with a template with ground multiplicity, while the second considers a template without specifying any concentration value.

Bio in with multiplicity:

```
long mult = 5;
BioTuple template = BioTuple.parse("test(X)",mult);
ITucsonOperation op = acc.in(tid, template, null);
```

looks for a tuple matching template in the target tuple space tid that has a concentration equal or greater than mult. If such a tuple is found when the operation is served, the execution succeeds by removing from it the indicated quantity and returning this tuple with concentration equal to mult. Otherwise, the execution is suspended to be resumed and successfully completed when a matching tuple, with the previous requirements, will be finally found in, removed and returned from the target tuple space as explained before.

Bio in without multiplicity:

```
BioTuple template = BioTuple.parse("test(X)");
ITucsonOperation op = acc.in(tid, template, null);
```

looks for a tuple matching template in the target tuple space tid. If such a tuple is found when the operation is served, the execution succeeds by removing this tuple entirely and returning it. Otherwise, the execution is suspended to be resumed and successfully completed when a matching tuple will be finally found in, removed and returned from the target tuple space.

Observations. Even now we decide to relegate all changes into TupleSet to

preserve from transformations RespectVMContext. In particular we change getMatchingTuple(LogicTuple t) that previously implemented only standard *in*. Here, we specify the behaviour of the three basic case: standard *in*, *bio in* with and without multiplicity. First separation is made always through instanceof while the second one through BioTuple's method isMulGround().

Bio rd

As before, we present two examples for *bio rd* extracted from TuCSoN agent code. Ultimately, the only difference compared to *bio in* is that *bio rd* does not remove matching tuple.

Bio rd with multiplicity:

```
long mult = 5;
BioTuple template = BioTuple.parse("test(X)",mult);
ITucsonOperation op = acc.rd(tid, template, null);
```

looks for a tuple matching template in the target tuple space tid that has a concentration equal or greater than mult. If such a tuple is found when the operation is served, the execution succeeds by returning it with concentration equal to mult. Otherwise, the execution is suspended to be resumed and successfully completed when a matching tuple, with the previous requirements, will be finally found in and returned from the target tuple space as explained before.

Bio rd without multiplicity:

```
BioTuple template = BioTuple.parse("test(X)");
ITucsonOperation op = acc.rd(tid, template, null);
```

looks for a tuple matching template in the target tuple space tid. If such a tuple is found when the operation is served, the execution succeeds by returning it with own overall concentration. Otherwise, the execution is suspended to be resumed and successfully completed when a matching tuple will be finally found in and returned from the target tuple space.

Observations. Similarly as before, we worked only into TupleSet class mod-

ifying readMatchingTuple(LogicTuple t), this time without removing matching tuple.

Bio out, *bio in* and *bio rd* are the basic primitives of BioTuCSoN (as well as *out*, *in* and *rd* are for TuCSoN). The other ones are a sort of their variation and so in the following we will point out only the distinctive characteristics of each primitive. We want to underline that every primitive manages templates with or without multiplicity, so the previous use cases can be extended to the other primitives simply replacing primitive's name.

Bio uin

Bio uin has exactly the same semantics of *bio in* save that the selection and the extraction of matching tuple is probabilistic. This means that if two or more tuples match the specified template, one is removed and returned with probability given by its multiplicity.

```
ITucsonOperation op = acc.uin(tid, template, null);
```

Observations. In this case we modify both RespectVMContext and TupleSet. We assign to RespectVMContext the task to verify whether or not template multiplicity is set. This choice is taken principally due to considerations linked to code readability. Unifying in a single function the management of both cases, generates too complex code, since the probabilistic behaviour requires additional operations compared to *bio in*. So we decided to make a clear separation designing two different method into TupleSet:

- getUniformMatchingTuple(BioTuple templ): manages template without multiplicity
- getUniformMatchingTupleGround(BioTuple templ): manages template with multiplicity

Now we illustrate how we worked presenting the "ground version" and explaining its behaviour through several steps. At first we have to select

from the bio space, all the tuples that match the specified template (templ). So we iterate over bioTuples list, but before applying Prolog matching, we check if the considered tuple has a multiplicity at least equal to that of the template. This expedient allows to achieve better performance since the matching operation, which is relatively time-consuming, is performed only on suitable tuples. So we insert all tuples that pass the two previous controls into a temporary list (tmp).

```
...
while (l.hasNext()){
    BioTuple tu=l.next();
    long multTu = tu.getMultiplicity();
    if(multTempl<=multTu){
      if (templ.match(tu)){
        multTot += tu.getMultiplicity();
        tmp.add(tu);
  }}}
...
```

After that we evaluate list size. If it is empty, the function returns null, activating *bio uin* suspensive semantics. Otherwise, if it contains only one element, we can immediately return it paying attention to remove the right quantity, as specified by template multiplicity.

```
...
if(tmp.size() == 0) return null;
else if(tmp.size() == 1){
  BioTuple t = tmp.getFirst();
  while(l.hasPrevious()){
    BioTuple tr = l.previous();
    if(t.toString().equals(tr.toString())){
      long multTr = tr.getMultiplicity();
      if(multTr == multTempl){
        l.remove();
        if(transaction)
          bioTRemoved.add(tr);
        }else if(multTr > multTempl){
          l.remove();
          if(transaction)
            bioTRemoved.add(tr);
          try {
            tr.setMultiplicity(multTr-multTempl);
          }catch(InvalidMultiplicityException e){
            e.printStackTrace();
```

```
          }
          l.add(tr);
          if(transaction)
          bioTAdded.add(tr);
        }
        AbstractMap<Var,Var> v = new LinkedHashMap<Var,Var>();
        try{
          return new BioTuple(tr.toTerm(multTempl).copyGoal(v, 0));
        }catch(InvalidMultiplicityException e){
          e.printStackTrace();
        }
}}}
...
```

Finally, if tmp has more than one element, we have to implement a probabilistic behaviour in order to select and retrieve with higher probability tuples with higher multiplicity values. We consider the following algorithm: as first it is computed a random value (r) from zero to the overall sum of multiplicities of matching tuples (multTot), so we iterate over tmp list, updating from time to time counter value, i.e. adding to it, at every turn, the value of multiplicity of the considered tuple. If counter is equal or greater than r, we extract from bio space the considered tuple, otherwise we repeat previous step. Since counter, at last iteration, is necessary equal to multTot, it is ensured that at least one element is retrieved.

```
...
else if(tmp.size()>1){
  long r = (long)(Math.random()*multTot);
  long counter = 0;
  int i = 0;
  BioTuple tuple;
  for(BioTuple t : tmp){
    tuple = tmp.get(i);
    i++;
    counter += tuple.getMultiplicity();
    if(counter >= r){
      while(l.hasPrevious()){
        BioTuple tr = l.previous();
        if(tuple.toString().equals(tr.toString())){
          long multTr = tr.getMultiplicity();
          if(multTr == multTempl){
            l.remove();
```

```
            if(transaction)
              bioTRemoved.add(tr);
          }else if(multTr > multTempl){
            l.remove();
            if(transaction)
              bioTRemoved.add(tr);
            try{
              tr.setMultiplicity(multTr-multTempl);
            }catch(InvalidMultiplicityException e){
              e.printStackTrace();
            }
            l.add(tr);
            if(transaction)
              bioTAdded.add(tr);
          }
          AbstractMap<Var,Var> v = new LinkedHashMap<Var,Var>();
          try {
            return new BioTuple(tr.toTerm(multTempl).copyGoal(v, 0));
          }catch(InvalidMultiplicityException e){
            e.printStackTrace();
          }
}}}}}}
...
```

Bio urd

Bio urd has exactly the same semantics of *bio rd* save that tuple reading is probabilistic. This means that if two or more tuples match the specified template, one is selected with probability given by its multiplicity.

```
ITucsonOperation op = acc.urd(tid, template, null);
```

Observations. Previous considerations are valid. We acted on RespectVM-Context inserting the control over template multiplicity, and on TupleSet defining as before two methods to manage templates with/without multiplicity.

Derived primitives

After we have defined *bio out*, *bio in*, *bio rd*, *bio uin* and *bio urd* the basic operations of the others primitives is actually already implemented. So we do

not have to make particular changes, but we exploit the existent mechanisms to perform non-suspensive semantics and negative variations. For these reasons, in the following, we only expose briefly the semantics of each primitive. As before the primitive structure which we refer to is:

```
ITucsonOperation op = acc.<prmitive>(tid, template, null);
```

Bio no looks for a tuple matching template in the target tuple space tid considered also multiplicity constraint if this value is specified in template. If no one tuple is found the operation is served, the execution succeeds, and template is returned; otherwise, the execution is suspended to be resumed and successfully completed when no suitable tuples can any longer be found in the target tuple space, then template is returned.

Bio nop predicative version of *bio no* (non-suspensive semantics); if a suitable[2] tuple is found the execution fails (operation outcome is FAILURE) and tuple is returned.

Bio inp predicative version of *bio in*; if a suitable tuple is not found the execution fails, no tuple (neither partially) is removed from the target tuple space and template is returned.

Bio rdp predicative version of *bio rd*; if a suitable tuple is not found the execution fails and template is returned.

Bio uinp uniform version of *bio inp*, i.e probabilistic extraction affected by multiplicity value and non-suspensive semantics.

Bio urdp uniform version of *bio rdp*, i.e probabilistic reading affected by multiplicity value and non-suspensive semantics.

Bio uno/Bio unop uniform version of *bio no* and *bio nop*.

[2]with *suitable* we mean that the tuple matches template and satisfy multiplicity constraint if its value is specified

4.3.4 Surrounding changes

Since we modify the primitive signatures into OrdinarySynchACC and UniformSynchACC interfaces, we have to enable their management from tuProlog agents, updating Tucson2PLibrary, and from command line, updating CLIAgent.

In order to allow tuProlog agents to use bio primitives, as first thing we change Prolog theory that defines all operators and predicates available. To ensure a syntax consistent with the previous examples, we split in two each predicate, assigned one to template/tuple with ground multiplicity and one to template with no multiplicity. Consequently, we define their behaviour through specific Java functions (one for each predicate).

On the other side, to enable bio primitives application from command line interface, we have to work on CLIAgent class. The request is read from input stream and parsed by TucsonOpParser, so that it is possible to identify the kind of primitive required and its tuple argument obtained as string value. In turn, the string represents tuple has to be parsed in order to create a proper BioTuple. This operation is done by parseCLI(String) function of BioTuple class just introduced. The solution adopted permitted us to manage bio primitIves exactly as standard primitives.

4.4 BioTuCSoN performance

After we have introduced bio primitives and tested that their behaviour is coherent with the specified semantics, we focus on evaluating BioTuCSoN performance. We take as a basis of comparison standard TuCSoN primitives, in order to appreciate the advantages or disadvantages of the new bio version. Not all primitives are considered but only the basic ones, because the performances of others can be simply derived. So we proceed as follows: first of all we describe the test planned, then we expose the achieved results with some explicative graphs.

4.4.1 Test environment

We want to analyse the performances of the five principal primitives (*out*, *in*, *rd*, *uin*, *urd*) in standard and bio version, i.e. evaluate their execution time values. For bio evaluations we consider both templates with and without multiplicity. So we have to supervise five operations for TuCSoN and ten for BioTuCSoN. The test environment is the same in both situations and sets up in order the following actions. As first, we test *out* primitive filling the space with *nTupleTot* tuples divided into *nType* different types. For BioTuCSoN test we consider tuples in form of:

$$biotuple(test\langle type\rangle(i), i),$$

while for TuCSoN:

$$test\langle type\rangle(i),$$

where, $type \in [1, 2, ..., nType]$ and $i \in [1, 2, ..., nTuple]$, having that $nTuple = \frac{nTupleTot}{nType}$.

Now we can start to monitor the computational time for the others primitives following a standard pattern: after selecting a specific type of tuple (*type*), performs *nIter* iterations of *rd*/*in*/*urd*/*uin* primitive, searching any tuple matches the template

$$biotuple(test\langle type\rangle(X), M)$$

if multiplicity is not specified, or

$$biotuple(test\langle type\rangle(X), r)$$

if it is, where r is a random number chosen in $[1, 2, ..., nTuple]$. On the other hand, for TuCSoN we use the template

$$test\langle type\rangle(X).$$

Since we have defined a parametric test environment, now we are able to perform several tests shaping, to our taste, the content of the space. In

order to evaluate primitives' behaviour in different conditions we plan some test benches each one with a specific purpose. However, in all simulations we keep the value of *nIter* fixed to 10, because its function is only to reduce the variability of a single execution of a primitive and to obtain a significant amount of time that can be compared. As consequence, the other tests concern about a sort of trade-off between the total number of tuples (*nTupleTot*), the number of tuples for each type (*nTuple*) and the number of type (*nType*).

In particular, we analyse two situations. As first, we hold *nType* fixed to 1 and increase of 1000 unit, at each new simulation, the value of *nTupleTot*, starting from 5000 up to 10000. The purpose is to verify primitives' behaviour filling the space with more and more matching tuples. Then, we evaluate how primitves manage "noise" tuples, setting as constant $nTuple = 1000$ and increasing, this time, *nType*. It means that reading/retriving primitives can choose at every simulations from 1000 matching tuples in a space filled of $1000 \times (nType - 1)$ "noise" tuples.

To be thorough, we list here the principal technical specific of the PC in which the simulations take place. Processor: Intel(R) Core(TM)2 Duo CPU, 2.00 GHz; RAM: 4 GB; system: 32-bit Operating System.

4.4.2 Performance results

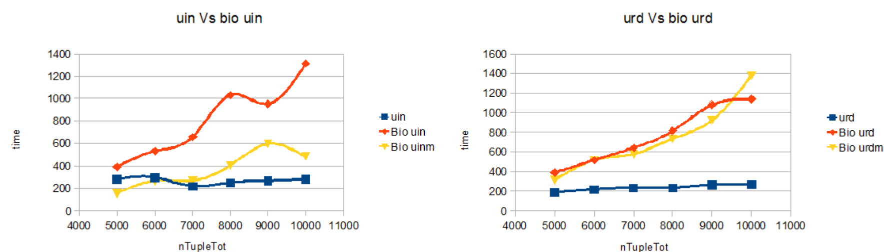
Now the achieved results are shown, describing it briefly in general and reporting in detail the comparison between uniform primitives of the two versions. In summary we can say that non-uniform primitives, both bio and standard, exhibit a good behaviour in all simulations, maintaining stable execution times that level off good values. They do not seem to be significant affected by the different conditions of the space. Comparing bio in/rd with and without multiplicity, we can notice that the first one takes slightly longer to be carried out, probably due to the more operations required to manage the concentration values.

More interesting considerations can be done in regards to uniform prim-

itives. In this case, their execution times alter substantially on varying of tuple space composition. We analyse separately the previous two situations.

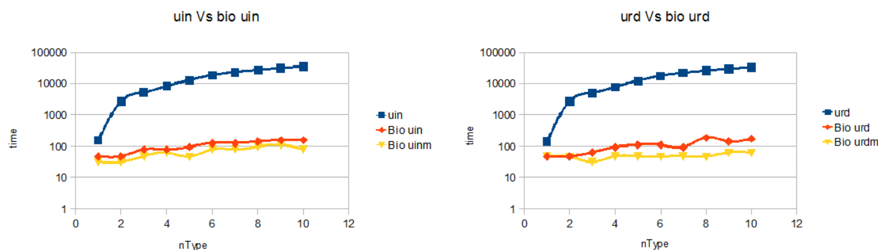
Test bench: increasing matching tuples

The increase of the number of matching tuples does not influence significantly the behaviour of standard uniform primitives that keep at a constant value its execution time. On the other hand, we can identify a tendency for bio uniform primitives to deteriorate their performances when they have to manage an higher multitude of matching tuples. Both for *bio uin* and *bio urd*, it is visible that "ground" versions settle to lower values of execution times in respect to "variable" versions. The gap between them appears different depending on the kind of bio uniform primitive considered. This is not a structural trend but it is probably due to their probabilistic behaviour. Indeed, the execution time of ground versions is deeply influenced by the value of multiplicity that we choose at random in the tests. These considerations can be spotted in the two following graphs.



Test bench: "noise tuples"

In this test bench, an opposite situation take shape. Standard uniform primitives manage with a big effort the presence of an increasing number of non-matching tuples, while the bio ones limit the decay of performances. It is important to underline that the next graphs report a logarithmic scale on Y axis, i.e. bio primitives largely overcome standard ones in these operative conditions.



4.5 Bio ReSpecT

So far, we have focussed the attention on implementing an independent bio version of TuCSoN, realizing its communication language (*bio tuple*) and its coordination primitives (*bio primitives*). Now, we want to consider and describe the bio extension of the coordination media, allowing ReSpecT to manage bio primitives, i.e. programming tuple centres to react to and respond with them. We have to highlight that meta-coordination language and meta-coordination primitives are left unchanged. It means that, even in our bio ReSpecT extension, the tuple centres are programmed by means of specific logic tuples, called specification tuples, whose form is reaction(E,G,R) and through standard meta-primitives. Basically, what we want to realize can be formally expressed as follows. Given a BioReSpecT event Ev, a specification tuple reaction(E,G,R) associates a reaction $R\theta$ to E_v if and only if $\theta = mgu(E, E_v)$ and guard predicate G is true, and where:

E(Event) : any BioTuCSoN primitive previous considered.

G(Guard) : same as ReSpecT guard.

R(Reaction) : any BioTuCSoN primitive previous considered.

The key issues to evaluate are related principally with the matching between event and bio primitives. Now bio primitives involve bio tuples, and so we have to establish an appropriate matching semantics, considering how manage the multiplicity value. More in concrete, for example, considering a reaction of the kind:

```
reaction(out(biotuple(test,MT)), response, in(biotuple(test,MR))
```

where MT and MR could be a ground or variable multiplicity, and an event such as:

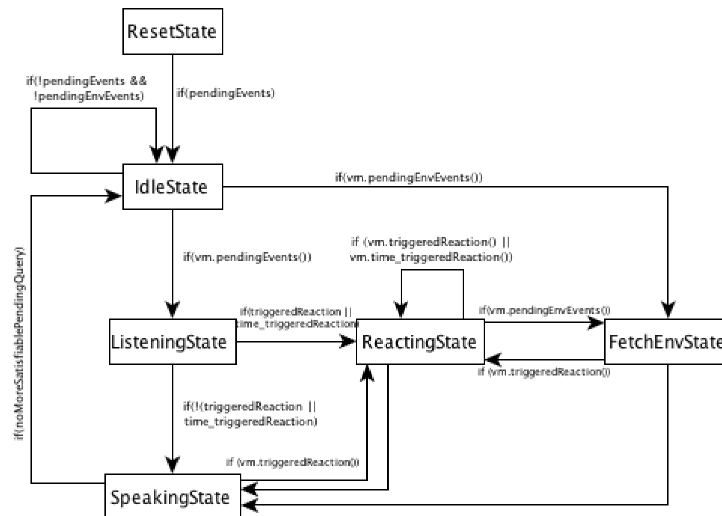
```
out(biotuple(test,M))
```

it is questionable whether such kind of event implies the firing of that reaction. We have three basic possibilities: approve the matching when $M \geq MT$ or $M \leq MT$ or, finally, only in case $M = MT$. In our implementation we consider the last solution, evaluating it as the more linear and consistent with the previous works.

After we have defined when a reaction can fire, since the guard predicates are exactly the same as for ReSpecT, we have only to focus on allowing BioReSpecT engine to elaborate bio primitives.

4.5.1 Our work

Here, we illustrate the undertaken work process to realize BioReSpecT, following the code flow triggered by the reactions. For a more detailed description of ReSpecT engine behaviour you can refer to chapter 3. However, to clarify we report, again, its state machine.



The core class, in which reactions are managed, is RespectVMContext. As we seen, this class, other than dealing with low-level management of primitives, has also the function of checking the presence of triggerable reactions by means of the essential method fetchTriggeredReactions(Event ev). It is invoked each time is necessary, to check eventual triggering reactions against a specific event. Each event can be of three types: *input*, *output* or *internal*. In every case, the role of this method is to analyse the operation related to *ev* and so builds a proper Prolog term, called currentReactionTerm. In turn, it will be used to create another Prolog term representing the reaction template, related with *ev*. This reaction term has to be searched in the Prolog theory that implements the programmable behaviour of the tuple centre. The reaction term is on the form of:

```
reaction( <currentReactionTerm> , Guard , Body )
```

where, *Guard* and *Body* are, for now, variables. Indeed, the only intent is to looking for a reaction that presents as *Event* something that matches with currentReactionTerm.

So as first intervention, we have to build, in the right way, currentReactionTerm. We want also to alter as little as possible RespectVMContext, always for ensure maintainability. For these reasons, the solution adopted is simply to override the method toTerm(), defining a specific version for bio tuples. So, thanks to Java polymorphism, it is possible to not change at all the method considered, because it will be execute the right toTerm() version depending on the native class of the instance involved in the operation.

As for bio primitives, we commit to the existent mechanisms all necessary operations that ensure the correct system behaviour and concentrate our labour on making available bio primitives for ReSpecT engine. It means taking Respec2PLibrary into account. This class represents a tuProlog library defining the behaviour of ReSpecT primitives, used inside ReSpecT virtual machine. Since the entire system gets started on by means of reactions involving logic tuples, we have to ensure the correct management both of bio and standard primitives. So, briefly, for each primitive it is verified the type

of the specific tuple involved and, depending on its nature, the proper code is performed. The control is made through an utility method inserted at the end of this class. Specific examples about BioReSpecT are illustrated in the following chapter.

Chapter 5

Case study

To prove the capabilities of BioTuCSoN & BioReSpecT we consider a case study extracted from the paper *Biochemical Tuple Spaces for Self-organising Coordination* [19].

5.1 Service ecosystem

5.1.1 General context

We want to realize a basic infrastructure that models a space in which pervasive services compete and interact following simple nature-inspired rules. These rules have to be selected and executed in a probabilistic way, as specified by Gillespie's algorithm discussed earlier. The aim is to achieve a global and complex self-organising behaviour arising from a limited and elementary set of rules.

The idea is to reify this infrastructure through biochemical tuple spaces disseminated on the network, each one with proper, general-purpose chemical rules. The agents of the system, such as *services*, *clients* and *devices*, coordinate them self interacting through such distributed coordination medium. The relevance of the information is modelled by the concept of concentration. A higher value stands for a higher pertinence of that service. The system dynamics as well as the services survival is governed by the evolution

of concentration over time.

The case study considered here presents a simple scenario in which agents compete and interact into a single node, with the mere intent of appreciating the capabilities of BioTuCSoN & BioReSpecT in relation to self-organisation and local competition of services.

5.1.2 Specific scenario

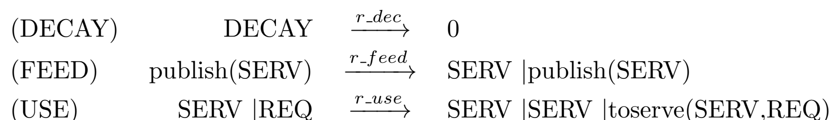
As expressed before, we now consider a simple scenario in which a single tuple space mediates the interaction between services and users in an open and highly-dynamic system. This means that there is no prior knowledge about what kind of services will be deployed and how much they will be requested and used.

In this condition, we want to check that: less requested services fade until eventually disappearing from the system and, on the other hand, the most useful services increase their concentration over time. Also, we want to verify that in the competition between two services, the most efficient wins over the other one. Services and clients interaction is regulated by the following protocol:

$$\begin{aligned} def D_s :=& out(\sigma, publish(service(ids, desc))).call D'_s \\ def D'_s :=& in(\sigma, toserve(service(ids, desc), request(Idc, Req))). \\ & out(\sigma, reply(Idc, Rep)).call D'_s \\ def D_c :=& out(\sigma, request(idc, req)).in(\sigma, reply(idc, Rep)) \end{aligned} \tag{5.1}$$

Service agents publish their service with *out* primitive, only one time, at the beginning. Then they enter in a loop to serve requests from clients, retrieving the request associated with their own service and inserting in the space the reply computed. Dually, client agents put requests in the space, waiting for the reply. It is responsibility of the system infrastructure to create *toServe* tuples that bind service and request, based on some criteria ideally affected by a certain semantic match degree.

The tuple space has to be programmed with the following rules to allow the desired self-orginising behaviour:



$$
\begin{array}{lrcl}
\text{(DECAY)} & \text{DECAY} & \xrightarrow{r_dec} & 0 \\
\text{(FEED)} & \text{publish(SERV)} & \xrightarrow{r_feed} & \text{SERV}\ |\text{publish(SERV)} \\
\text{(USE)} & \text{SERV}\ |\text{REQ} & \xrightarrow{r_use} & \text{SERV}\ |\text{SERV}\ |\text{toserve(SERV,REQ)}
\end{array}
$$

5.2 Test system architecture

After we had focussed the reference scenario, we realized a software system to really execute tests on BioTuCSoN & BioReSpecT. The system, or eco-system, is composed, essentially, by the following agents:

- **TCConfigurator** : its role is to configure the node where simulations happen, i.e. sets reactions, publishes rules and then starts off the simulation.

- **ServiceAgent** : its role is to publish a specific service and to provide replies to the related requests; there are as many ServiceAgent-s as the number of services.

- **ClientAgent** : its role is to make requests; each request is modelled as a single client, inserting in the space at a specific rate.

5.2.1 Tuple space programming

First of all, we have to illustrate the tuple spaces programming logic. It is wrapped in the class *TCConfigurator*, that extends *TucsonAgent*. (DECAY), (FEED) and (USE) rules are reified by specific ReSpecT reactions that, according to [18], have to be selected and executed in a probabilistic way. To simulate the chemical engine behaviour we exploit the concept of multiplicity (or concentration) typical of BioTuCSoN. The idea is to put in the space as many bio tuples as the number of rules, each one associated with a specific

multiplicity value that corresponds to its own rate. For example:

$$biotuple(rule(decay), 1)$$
$$biotuple(rule(feed), 100)$$
$$biotuple(rule(use), 50)$$

So, each simulation step is started through a probabilistic read that selects the rule to execute:

$$urd(biotuple(rule(R), C))$$

The time interval between steps are computed, non-deterministically, with

$$\Delta t = \frac{\log(1/\tau)}{R},$$

extracted by Gillespie's algorithm (τ:random number in [0,1]).

We have to clarify that this is a very simple approximation of chemical engine. Some relevant aspects are not considered here. For example rule rates are not affected by reagent concentrations and so a rule can be selected even if in the space there are not its reagents. Or, also, it is not provided any support of semantic match. Although we recognise these omissions, the approach adopted is considered adequate for our purpose.

In particular, tuple centre is programmed with the following reactions:

$$reaction(out(biotuple(selection, X)), response, (in(biotuple(selection, X)), urd(biotuple(rule(R), Rate)), out(biotuple(current_rule(R), Rate)))) \tag{5.2}$$

$$reaction(out(biotuple(current_rule(decay_service), Rate)), response, (in(biotuple(current_rule(R), Rate)), uin(biotuple(service(Ids, Desc), 1)))) \tag{5.3}$$

$$reaction(out(biotuple(current_rule(decay_publish), Rate)), response,$$
$$(in(biotuple(current_rule(R), Rate)), uin(biotuple(publish(S), 1)))) \quad (5.4)$$

$$reaction(out(biotuple(current_rule(feed), Rate)), response,$$
$$(in(biotuple(current_rule(R), Rate)), urd(biotuple(publish(S), C)),$$
$$out(biotuple(S, C)))) \quad (5.5)$$

$$reaction(out(biotuple(current_rule(use), Rate)), response,$$
$$(in(biotuple(current_rule(R), Rate)), urd(biotuple(service(Ids, Desc), C)),$$
$$uin(biotuple(request(Idc, Desc), 1)), out(biotuple(service(Ids, Desc), 1)),$$
$$out(biotuple(toserve(service(Ids, Desc), request(Idc, Desc)), 1)))) \quad (5.6)$$

(5.2) is triggered at the beginning of each step in order to select the reaction to execute. Others simply map (DECAY), (FEED) and (USE) rules. (DECAY) rule is subdivided in two reactions (5.3) and (5.4) to allow the fading both of publications and services.

5.2.2 Services and clients

The active entities of the system are represented by *ServiceAgent* and *ClientAgent*. Both extend *TucsonAgent* class and their behaviour is consistent with the previous specification (5.1). Clients make requests probabilistically; requests' rates are one of the simulation parameters.

In addition, we have defined an agent that stands for monitoring the space and, in particular, the evolution of service concentration over time.

5.3 Simulations planned

Here we explain the simulations planned, their structure and aims. Consequently, we present some graphs obtained from plotting the results of the most significant instances.

We have defined four type of tests that coincide with as many again classes. Each one have the function of establishing the connection with the TuCSoN node and then launching the agents with the proper set of parameters distinctive of its specific simulation.

The first, called **NoiseAndCompetition**, presents some services associated with a low demand and two highly popular services. The aim is to check that the unpopular services settle their multiplicity value over time at a low level while, in competition between the other two, the most required service win.

Then, in the **FeedAsDecay** test, we set the same value for (DECAY) and (FEED) rates. In this case, it is necessary to start with a high multiplicity value for each service to ensure their initial diffusion. The purpose is to check that the multiplicity value of the services increases over time mostly due to the rule (USE).

As a third test we present **TemporaryFeed** that simply makes available the rule (FEED) for a limited number of iterations. The aim is to check that the multiplicity value of less required services vanishes over time.

Finally, the **ServComp41Req** test wants to simulate two services that compete for replying the same kind of request. One service is more efficient than the other, and the first should win the contest.

After several preliminary simulations, we set the value of some critical parameters that will remain unchanged in all tests. A proper number of simulation steps is setted in 10000: it is sufficient to identify significant trends in the evolution of concentration value and not overly time consuming. Another important parameter is the rate at which clients make requests. If it is too high the system cannot manage all of them and crashes. On the other hand, a too low rate implies slow system dynamics and does not put

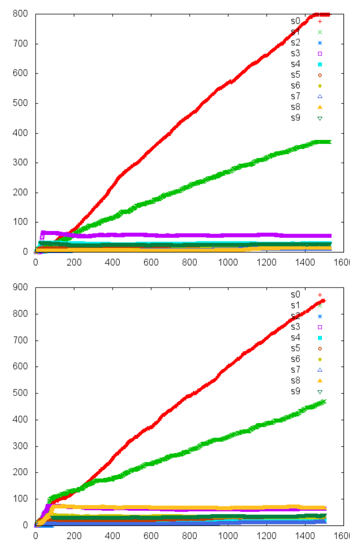
the infrastructure under enough stress. We suppose that the time interval between requests is 1 second (expressed in millisecond, $rate = 1 \div 1000 = 0.001$).

The parameters are presented in the following order: first we show the available *services* with the associated *probability request* (i.e. probability that a client asks exactly that service), then *rate request* and number of *simulation iterations*, fixed parameters; after we consider the number of iterations in which (FEED) rule is available (**null** value means that is always available, it is setted not **null** only in TemporaryFeed test). Finally, we report the rates of the rules, one of the most critical and influential parameters.

5.3.1 NoiseAndCompetition

Low noise

```
# Services -> Req.prob
#     s9 -> 1.0%
#     s8 -> 1.0%
#     s7 -> 1.0%
#     s6 -> 1.0%
#     s5 -> 1.0%
#     s4 -> 1.0%
#     s3 -> 1.0%
#     s2 -> 1.0%
#     s1 -> 30.0%
#     s0 -> 62.0%
# Rate request = 0.001
# N.iterations = 10000
# (FEED) iter. = null
# Rate rules :
#     decay publish = 1
#     decay service = 2
#     feed = 50
#     use = 25
```



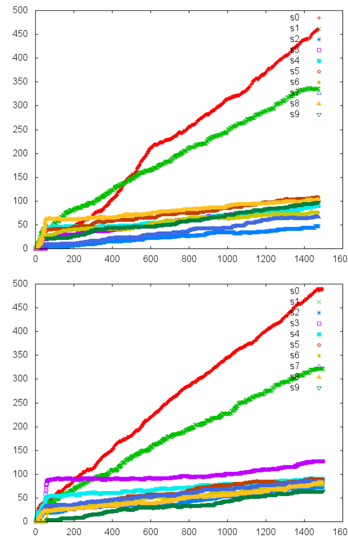
We report two graphs obtained from two simulations with the same set of parameters. In this case, the request's probability for secondary services is very low (1%) while, as concerning the primary ones, one has double request's probability than the other. The trend appears well-defined, with the noise

due to less requested services confined to low value.

In the first graph it is interesting to highlight that, at the initial stage, service s3 obtains a high concentration value, likely due to a consecutive choice of rule (FEED) for that service. However this initial fluctuation does not compromise the system dynamics and concentration value settles at a low level. In this sense, the system can be defined as "stable".

High noise

```
# Services -> Req.prob
#     s9 -> 5.0%
#     s8 -> 5.0%
#     s7 -> 5.0%
#     s6 -> 5.0%
#     s5 -> 5.0%
#     s4 -> 5.0%
#     s3 -> 5.0%
#     s2 -> 5.0%
#     s1 -> 25.0%
#     s0 -> 35.0%
# Rate request = 0.001
# N.iterations = 10000
# (FEED) iter. = null
# Rate rules :
#     decay publish = 1
#     decay service = 2
#     feed = 50
#     use = 25
```

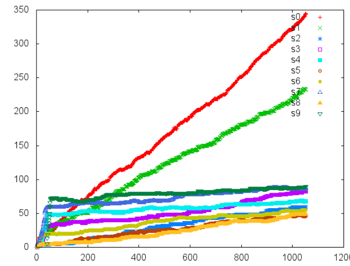


Here noise level was slightly increased, but we can still observe that it does not damage the correct evolution of service concentration. However, the trend is now less clear than before. Even now initial fluctuations do not compromise the following system behaviour.

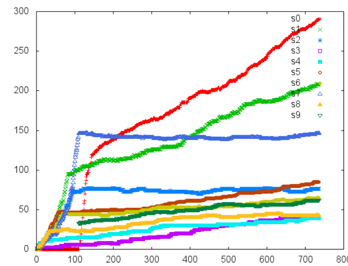
(FEED) rate Vs (USE) rate

Maintaining the previous request probabilities (s0 : 35%, s1 : 25%, others 5%) and the same values for rate request, n.iterations and (FEED) iter., we want to analyse the evolution of service concentration changing only the ratio between (FEED) rate and (USE) rate.

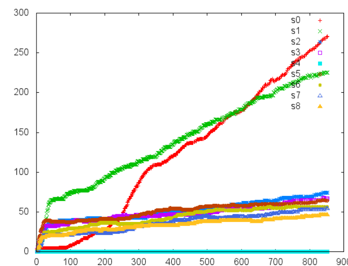
```
# Rate rules :
#     decay publish = 1
#     decay service = 2
#     feed = 50
#     use = 50
```




```
# Rate rules :
#     decay publish = 1
#     decay service = 2
#     feed = 50
#     use = 100
```




```
# Rate rules :
#     decay publish = 1
#     decay service = 2
#     feed = 100
#     use = 50
```

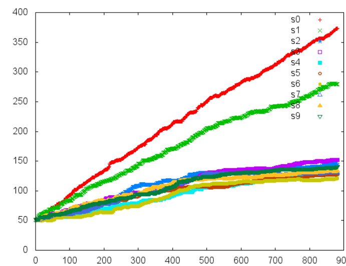


We have to pay special attention to this series of graphs. At first sight we might be quite surprised. A greater (USE) rate might lead us to think that more requested services would be rewarded while increasing the value of ratio($= use_rate \div feed_rate$) in favour of the (USE) rule, the trend seems to be less well-defined. But, actually, looking carefully, we can notice that this unexpected behaviour appears only at the initial stage and it is correct. A high (USE) rate implies that a small initial fluctuation, due to a probabilistic choice of (FEED), will accentuate, rewarding the services just published. In spite of this wrong parameter setting, the system appears "stable", even if it takes longer to stabilize the correct trend.

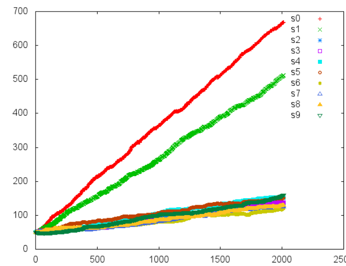
5.3.2 FeedAsDecay

Here we want to check the system dynamics when (FEED) rate is equal to (DECAY) rate. Since there are two rules that reify (DECAY), we present three different settings of parameters, trying to explore some interesting combinations. The parameters, that are not specified, have the same value as before.

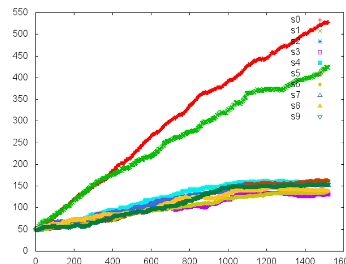
```
# Rate rules :
#     decay publish = 1
#     decay service = 10
#     feed = 10
#     use = 100
```




```
# Rate rules :
#     decay publish = 1
#     decay service = 1
#     feed = 1
#     use = 50
```




```
# Rate rules :
#     decay publish = 5
#     decay service = 5
#     feed = 10
#     use = 50
```



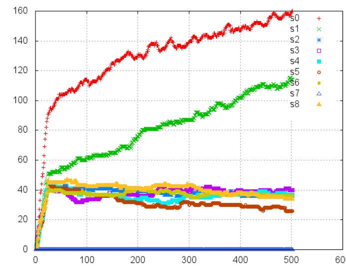
As we can observe from the graphs, all services start with a specific concentration value (specifically 50) that is setted for each one at the first publication. This is necessary to ensure their initial diffusion in spite of (FEED) rule and (DECAY) rule have similar rates.

The equilibrium between (FEED) rate and (DECAY) rate implies the disappearance of the initial fluctuations, a fact that confirms the previous assertions.

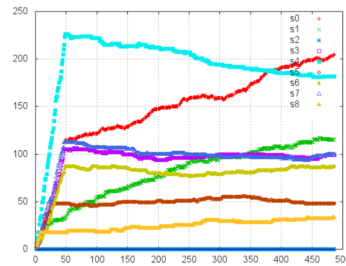
5.3.3 TemporaryFeed

We report here results of simulations in which (FEED) rule is available for a limited number of steps, specifying only the changed parameters.

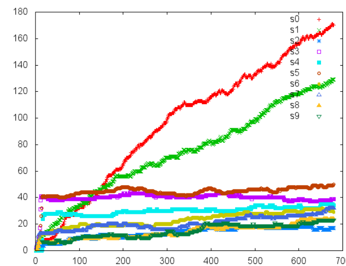
```
# (FEED) iter. = 500
# Rate rules :
#     decay publish = 1
#     decay service = 2
#     feed = 200
#     use = 50
```



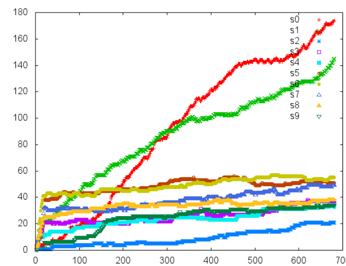

```
# (FEED) iter. = 1000
# Rate rules :
#     decay publish = 1
#     decay service = 2
#     feed = 200
#     use = 50
```




```
# (FEED) iter. = 3000
# Rate rules :
#     decay publish = 1
#     decay service = 10
#     feed = 100
#     use = 50
```




```
# (FEED) iter. = 5000
# Rate rules :
#     decay publish = 1
#     decay service = 10
#     feed = 100
#     use = 50
```



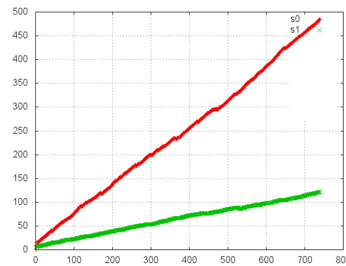
The initial fluctuations are very accentuated in the first two simulations ($feed_rate = 200$) while less emphasized in the last two ($feed_rate = 100$).

However, in all four cases, a slight downward trend of concentration value is visible related to less requested services when (FEED) rule is made unavailable. On the other hand, the two popular services are not affected by the elimination of the rule and continue to increase their concentration thanks to (USE) rule.

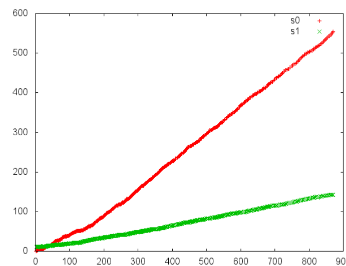
5.3.4 ServComp41Req

These simulations deal with the competition of two equiprobable services for serving the same kind of request. We want to check that the more efficient service increases its concentration to the detriment of the other one.

```
# Services -> Req.prob
#     s1 -> 50.0%
#     s0 -> 50.0%
# Rate request = 0.001
# N.iterations = 10000
# (FEED) iter. = null
# Rate rules :
#     decay publish = 1
#     decay service = 2
#     feed = 100
#     use = 50
```




```
# Services -> Req.prob
#     s1 -> 50.0%
#     s0 -> 50.0%
# Rate request = 0.001
# N.iterations = 10000
# (FEED) iter. = null
# Rate rules :
#     decay publish = 1
#     decay service = 2
#     feed = 50
#     use = 100
```



In both simulations the most efficient service wins the competition. Service *s1* takes three times as long as service *s0* to reply to the same kind of request, which means that *s0* manages more clients' requests than *s1*. To allow that concentration value increases according with the number of

served clients, a small change in the system is necessary. In particular we have to ensure that a service is not rewarded when the related tuple *toserve* is chosen but effectively when the service makes the reply available. For this purpose we modify the rule (USE) removing the rewarding insertion of tuple *service(Ids,Desc)* and assigning this task to *ServiceAgent* after it has put the reply into the space.

Conclusions & Future works

The achieved results prove that BioTuCSoN & BioReSpecT provide interesting mechanisms to model systems characterized by some basic form of self-organisation and situatedness. In particular, we have demonstrated that this infrastructure faces well *local competition* between services, satisfying one of the behaviours described in chapter 2 concerning biochemical tuple spaces model.

Further tests would be necessary to inspect the capabilities of our implementation as regards spatial competition or gradient-based patterns. Actually, the software developed to realize the case study was designed also to support a distributed architecture composed by several tuple centres, in different network nodes, that interact spreading probabilistically their tuples according to a specific rule named (DIFFUSE). However, due to some infrastructural problems, it was not possible to realize concrete tests on this issue.

It is also valuable to point out that we have obtained good results in spite of the fact that we have not exploited the functionalities proper of a chemical engine. This means that our extension, associated with some probabilistic methods of rules' execution, suffices to implement simple forms of coordination based on nature-inspired patterns.

We can, also, consider BioTuCSoN & BioReSpecT as a foundation from which to start to define a full version of the biochemical tuple spaces model. A first aspect is the integration of a chemical engine, following Gillespie's algorithm, as basic mechanism of the infrastructure, for example adding a

specific Java class that provides such functionalities. Another key issue is to enable some kind of semantic reasoning for matching operations. A possible solution could refer to the notion of domain ontology using description logic to define relationship among its elements. Finally, according to paper [18], it is necessary to associate to bio primitives a probabilistic behaviour based on the described DTMC model. This aspect could involve some significant changes to BioTuCSoN core.

In conclusion, we can assert that our work, starting from the analysis of background researches on nature-inspired computational models, leads to the implementation of a stable and independent bio extension of TuCSoN that reveals good results as for local competition and self-organisation of services. It is a first step towards a full implementation of the biochemical tuple spaces model, that is a potential approach, nature-inspired, to deal with the new requirements of current pervasive environments. The idea behind these researches and this thesis is to change the relationship between human and technology. Users no longer exploit technology, but technology exploits users' preferences to shape the status of the environmental (computational) artifacts in order to satisfy, in every moment, the necessities of the people around. This vision entails a better management of the information, providing their more convenient presentation, so that to enrich social context and, ultimately, to improve daily life.

Bibliography

[1] Mario Bravetti, Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. Quantitative information in the tuple space coordination model. *Theoretical Computer Science*, 346(1):28–57, 23 November 2005.

[2] Scott Camazine, Jean-Louis Deneubourg, Nigel R Franks, James Sneyd, Guy Theraula, and Eric Bonabeau. *Self-organization in biological systems*. Princeton University Press, 2003.

[3] Paolo Ciancarini, Andrea Omicini, and Franco Zambonelli. Coordination technologies for internet agents. *Nordic Journal of Computing*, 6(3):215–240, 1999.

[4] J.L. Fernandez-Marquez, J.L. Arcos, G. Di Marzo Serugendo, and M. Casadei. Description and composition of bio-inspired design patterns: The gossip case. In *Engineering of Autonomic and Autonomous Systems (EASe), 2011 8th IEEE International Conference and Workshops on*, pages 87 –96, april 2011.

[5] Jose Luis Fernandez-Marquez, Josep Lluis Arcos, Giovanna Di Marzo Serugendo, Mirko Viroli, and Sara Montagna. Description and composition of bio-inspired design patterns: the gradient case. In *Proceedings of the 3rd workshop on Biologically inspired algorithms for distributed systems*, BADS '11, pages 25–32, New York, NY, USA, 2011. ACM.

[6] JoseLuis Fernandez-Marquez, Giovanna Marzo Serugendo, Sara Montagna, Mirko Viroli, and JosepLluis Arcos. Description and composition

of bio-inspired design patterns: a complete overview. *Natural Computing*, pages 1–25, 2012.

[7] Luca Gardelli, Mirko Viroli, and Andrea Omicini. Design patterns for self-organising systems. *Multi-Agent Systems and Applications V*, pages 123–132, 2007.

[8] Daniel T Gillespie. Exact stochastic simulation of coupled chemical reactions. *The journal of physical chemistry*, 81(25):2340–2361, 1977.

[9] Fausto Giunchiglia, Mikalai Yatskevich, and Pavel Shvaiko. Semantic matching: Algorithms and implementation. *Journal on Data Semantics IX*, pages 1–38, 2007.

[10] Marco Mamei, Ronaldo Menezes, Robert Tolksdorf, and Franco Zambonelli. Case studies for self-organization in computer science. *Journal of Systems Architecture*, 52(8):443–460, 2006.

[11] Elena Nardini, Mirko Viroli, Matteo Casadei, and Andrea Omicini. A self-organising infrastructure for chemical-semantic coordination: Experiments in TuCSoN. In Andrea Omicini and Mirko Viroli, editors, *WOA 2010 – Dagli oggetti agli agenti. Modelli e tecnologie per sistemi complessi: context-dependent, knowledge-intensive, nature-inspired e self-**, volume 621 of *CEUR Workshop Proceedings*, pages 117–125, Rimini, Italy, 5-7 September 2010. Sun SITE Central Europe, RWTH Aachen University.

[12] Elena Nardini, Mirko Viroli, and Emanuele Panzavolta. Coordination in open and dynamic environments with tucson semantic tuple centres. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2037–2044. ACM, 2010.

[13] Andrea Omicini. Formal respect in the a&a perspective. *Electronic Notes in Theoretical Computer Science*, 175(2):97–117, 2007.

[14] Andrea Omicini and Enrico Denti. From tuple spaces to tuple centres. *Science of Computer Programming*, 41(3):277–294, 2001.

[15] Andrea Omicini and Luca Gardelli. Self-organisation & mas an introduction, 2010.

[16] Andrea Omicini and Stefano Mariani. *TuCSoN coordination Model & Technology A Guide*, 2013.

[17] M. Satyanarayanan. Pervasive computing: vision and challenges. *Personal Communications, IEEE*, 8(4):10 –17, aug 2001.

[18] Mirko Viroli and Matteo Casadei. Biochemical tuple spaces for self-organising coordination. In John Field and Vasco T. Vasconcelos, editors, *Coordination Languages and Models*, volume 5521 of *LNCS*, pages 143–162. Springer, Lisbon, Portugal, June 2009. 11th International Conference (COORDINATION 2009), Lisbon, Portugal, June 2009. Proceedings.

[19] Mirko Viroli and Matteo Casadei. Chemical-inspired self-composition of competing services. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew Palakal, Chih-Cheng Hung, and Dongwan Shin, editors, *25th Annual ACM Symposium on Applied Computing (SAC 2010)*, volume III, pages 2029–2036, Sierre, Switzerland, 22–26 March 2010. ACM.

[20] Mirko Viroli, Matteo Casadei, Sara Montagna, and Franco Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Transactions on Autonomous and Adaptive Systems*, 6(2):14:1–14:24, June 2011.

[21] Mirko Viroli, Elena Nardini, Gabriella Castelli, Marco Mamei, and Franco Zambonelli. A coordination approach to adaptive pervasive service ecosystems. In *2011 5th IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2011)*, pages 114–119, SASO

2011, Ann Arbor, MI, USA, 7 October 2011. IEEE CS. 1st Awareness Workshop "Challenges in achieving self-awareness in autonomous systems" (AWARE 2011).

[22] Mirko Viroli, Andrea Omicini, and Alessandro Ricci. Engineering mas environment with artifacts. In *2nd International Workshop Environments for Multi-Agent Systems(E4MAS 2005)*, pages 62–77. AAMAS, 2005.

[23] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, 1991.

[24] George Wells. Coordination languages: Back to the future with linda. In *Proceedings of the Second International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT05)*, pages 87–98, 2005.

[25] Franco Zambonelli, Gabriella Castelli, Laura Ferrari, Marco Mamei, Alberto Rosi, Giovanna Di Marzo Serugendo, Matteo Risoldi, Akla-Esso Tchao, Simon Dobson, Graeme Stevenson, Yuan Ye, Elena Nardini, Andrea Omicini, Sara Montagna, Mirko Viroli, Alois Ferscha, Sascha Maschek, and Bernhard Wally. Self-aware pervasive service ecosystems. *Procedia Computer Science*, 7:197–199, December 2011. Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET 11).

[26] Franco Zambonelli and Mirko Viroli. A survey on nature-inspired metaphors for pervasive service ecosystems. *International Journal of Pervasive Computing and Communications*, 7(3):186–204, 2011.

Ringraziamenti

Innanzitutto vorrei ringraziare il professore Andrea Omicini, relatore della tesi, che ha saputo ascoltare le mie richieste e assegnarmi un progetto molto interessante, relativo a tematiche che mi hanno affascinato fin dalla prima volta che ci sono state presentate. Ha avuto un ruolo chiave nella definizione del processo di sviluppo del progetto, tracciando la strada da seguire e mettendo di volta in volta in luce gli aspetti principali da affrontare. Un grosso ringraziamento va anche al correlatore, l'ingegnere Stefano Mariani, che mi ha supportato durante tutti questi mesi di lavoro. Sempre disponibile a rispondere con chiarezza e pazienza ad ogni mio dubbio, a risolvere ogni problema. Temo avrá parecchio da lavorare i prossimi mesi con tutta la pubblicitá che gli ho fatto.

Ringrazio la mia famiglia. I miei genitori che mi hanno messo nelle condizioni migliori per potermi concentrare nello studio e allo stesso tempo divertirmi, che mi hanno sempre supportato e sono sempre stati molto vigili e attenti che non mi distraessi troppo. I miei fratelli e le loro famiglie, in particolare i miei nipoti Betta, Lucy, Pier ("il Nano") e Cotta (in ordine crescente di etá per essere rigorosi) che mi hanno permesso di approfondire la mia cultura su DisneyChannel, Patty, Violetta, etc.. ma anche che con il loro "delirio" hanno reso piú "vive" e divertenti le giornate di studio barricato in casa.

Vorrei anche ringraziare i miei amici, colleghi e compagni di progetto Whites, Mandu, Corza con cui ho condiviso difficoltá ma anche tante soddisfazioni e momenti divertenti. Un ringraziamento speciale va a Davide, in-

sieme a cui ho fatto praticamente tutti i progetti, per tutte le cose che mi ha insegnato, per aver sopportato i miei momenti di follia pre-esame, per tutte le discussioni sull'universitá, politica, donne, per aver dato un contributo sostanziale ad una migliore cura del paesaggio nelle colline cesenati. Senza queste persone sarebbe stato molto piú difficile arrivare fin qui, e sicuramente sarebbe stato molto meno divertente.

Infine un ringraziamento va ai miei amici extra-universitari, che hanno contribuito a ricostruire e riempire la mia vita in un momento di cambiamento e grazie ai quali non ho nemmeno mai pensato (qualcuno riderá) di "ricascare" nel passato. La mia squadra/e di basket, per la straordinaria esperienza di Spoleto, ma in generale per ogni momento con loro... sono dei pazzi... non credo esistano persone piú divertenti e "imbarazzanti" di loro!! ... E per ultimi i famigerati BBoys, troppi da elencare (lo pensa anche il mio freezer!!), ma tutti a loro modo unici.. per tutti i week-end, sempre diversi, in posti esotici e lontani.. sono come una seconda famiglia ed é bello sapere di poter contare su un gruppo cosí.