Alma Mater Studiorum - University of Bologna

DEIS - DEPARTMENT OF ELECTRONICS, COMPUTER SCIENCE AND SYSTEMS
PhD Course in Electronics, Computer Science and Telecommunications

XXIII CYCLE – SCIENTIFIC-DISCIPLINARY SECTOR ING-INF/05

SEMANTIC COORDINATION THROUGH PROGRAMMABLE TUPLE SPACES

Candidate:
Dott. Ing. ELENA NARDINI

Supevisors:
Chiar.mo Prof. Ing. ANDREA OMICINI

Tutor:
Chiar.mo Prof. Ing. ANTONIO NATALI

Advisor:
Prof. Ing. MIRKO VIROLI

PhD Course Coordinator:
Chiar.ma Prof. Ing. PAOLA MELLO

FINAL EXAMINATION YEAR 2011

Acknowledgement

I would like to thanks all the people that have supported my research activity during the last three years. First of all my thanks go to my family: my father Carlo, my mother Patrizia and my sisters Elisa and Eleonora. They have always trusted me, despite the difficult moments. I also would like to thanks Enrico and Ambra. They are more friends than colleagues. Then, thanks to Andrea, Mirko and Antonio, which have driven me during this hard walk.

Finally, thanks to Andrea. Even if you arrived only one year ago, you have had a fundamental role. . .

Contents

Abstract

Two of the main features of today complex software systems like pervasive computing systems and Internet-based applications are *distribution* and *openness*. Distribution revolves around three orthogonal dimensions: *(i) distribution of control*—systems are characterised by several independent computational entities and devices, each representing an autonomous and proactive locus of control; *(ii) spatial distribution*—entities and devices are physically distributed and connected in a global (such as the Internet) or local network; and *(iii) temporal distribution*—interacting system components come and go over time, and are not required to be available for interaction at the same time. Openness deals with the *heterogeneity* and *dynamism* of system components: complex computational systems are open to the integration of diverse components, heterogeneous in terms of architecture and technology, and are dynamic since they allow components to be updated, added, or removed while the system is running.

The engineering of open and distributed computational systems mandates for the adoption of a software infrastructure whose underlying model and technology could provide the required level of *uncoupling* among system components. This is the main motivation behind current research trends in the area of coordination middleware to exploit *tuple-based coordination models* in the engineering of complex software systems, since they intrinsically provide coordinated components with *communication uncoupling*.

An additional daunting challenge for tuple-based models comes from knowledge-intensive application scenarios, namely, scenarios where most of the activities are based on knowledge in some form—and where knowledge becomes the prominent means by which systems get coordinated. Handling knowledge in tuple-based systems induces problems in terms of syntax – e.g., two tuples containing the same data may not match due to differences in the tuple structure – and (mostly) of semantics—e.g., two tuples representing the same information may not match based on a different syntax adopted. Till now, the problem has been faced by exploiting tuple-based coordination within a middleware for knowledge intensive environments: e.g., experiments with tuple-based coordination within a Semantic Web middleware (surveys analogous approaches). However, they appear to be designed to tackle the design of coordination for specific application contexts like Semantic Web and Semantic Web Services, and they result in a rather involved extension of the tuple space model.

The main goal of this thesis was to conceive a more general approach to semantic

coordination. In particular, it was developed the model and technology of *semantic tuple centres*. It is adopted the *tuple centre* model as main coordination abstraction to manage system interactions. A tuple centre can be seen as a programmable tuple space, i.e. an extension of a LINDA tuple space, where the behaviour of the tuple space can be programmed so as to react to interaction events. By encapsulating coordination laws within coordination media, tuple centres promote *coordination uncoupling* among coordinated components. Then, the tuple centre model was semantically enriched: a main design choice in this work was to try not to completely redesign the existing syntactic tuple space model, but rather provide a smooth extension that – although supporting semantic reasoning – keep the simplicity of tuple and tuple matching as easier as possible. By encapsulating the semantic representation of the domain of discourse within coordination media, semantic tuple centres promote *semantic uncoupling* among coordinated components.

The main contributions of the thesis are: *(i)* the design of the semantic tuple centre model; *(ii)* the implementation and evaluation of the model based on an existent coordination infrastructure; *(iii)* a view of the application scenarios in which semantic tuple centres seem to be suitable as coordination media.

Chapter 1

Introduction

1.1 Background and Motivation

Software systems like pervasive computing systems and Internet-based applications are mainly characterised by *distribution* and *openness* [100]. Distribution revolves around three orthogonal dimensions: *distribution of control*—systems are characterised by several independent computational entities and devices, each representing an autonomous and proactive locus of control; *spatial distribution*—entities and devices are physically distributed and connected in a global (such as the Internet) or local network; and *temporal distribution*—interacting system components come and go over time, and are not required to be available for interaction at the same time. Openness deals with the *heterogeneity* and *dynamism* of system components: complex computational systems are open to the integration of diverse components, heterogeneous in terms of architecture and technology, and are dynamic since they allow components to be updated, added, or removed while the system is running.

The engineering of open and distributed computational systems mandates for the adoption of a software infrastructure whose underlying model and technology could provide the required level of *uncoupling* among system components. This is the main motivation behind current research trends in the area of coordination middleware to exploit *tuple-based coordination models* in the engineering of complex software systems, since they intrinsically provide coordinated components with *communication uncoupling*—the interested reader can find a historical perspective in [69], and further details in the references therein.

Tuple spaces are information spaces structured as sets of tuples, which are structured and ordered chunks of information. In tuple-based coordination models, system components interact, communicate, synchronise and coordinate via tuple spaces by inserting, reading and consuming tuples. According to the *generative communication* paradigm [35], tuples are permanently written in a tuple space through the *out* primitive, then made available to any process for reading/consuming through *rd*/*in* primitives, respectively. Tuples are read and retrieved *associatively*: in order to either read or consume a tuple, a *tuple template* – that is, a description of a tuple set – has to be specified. When a tuple matching the template is found in the tuple space, it is returned as the result for the tuple request. Generative communication and associative access make tuple-based coordination models particularly interesting for the engineering of distributed, open and knowledge-intensive systems. Generative communication promotes *communication uncoupling* among system components: tuple producers and consumers can interact with no prior knowledge about each other, thus leading to *space*, *time* and *name uncoupling*—which fit well with open and distributed application scenarios. Associative access to tuples along with synchronisation based on tuple availability in the tuple space promotes *knowledge-based coordination* where system components coordinate based on tuple information content and structure—which fits well with knowledge-intensive application scenarios.

After the original formulation within the LINDA model [35], a number of implementations and extensions have been developed and proposed in the literature. Among the proposals aimed at providing effective technologies supporting the tuple-based coordination model, remarkable examples are Sun's JavaSpaces [33] and GigaSpaces [37] Many other proposals have instead focussed on extending the tuple-based coordination model beyond its original limitations. Among the many different classes of extended tuple-based models, two are notable for their impact on the engineering of complex computational systems: those focussing on the *programmability of the behaviour* of the tuple-based coordination abstraction, and those enhancing tuple-based communication with *semantics*.

The behaviour of the original LINDA tuple spaces – represented by their state transition in response to the invocation of the standard coordination primitives – is set once and for all by the model, and cannot be tailored to the specific application needs [67]. As a consequence, any coordination policy not directly supported by the standard behaviour of the coordination abstraction – the tuple space – has to be charged upon system components, which hence grow in complexity—thus hampering the effectiveness of the coordination model especially in open scenarios [70]. In order to overcome such a limitation, a number of tuple-based coordination models and languages have been proposed that extend the original tuple-space model by allowing the behaviour of the tuple spaces to be programmed so as to embed coordination policies within the coordination media. By encapsulating coordination policies in the coordination media makes it possible to free coordinated components from the burden of the shared, social aspects of coordination. So, each component can be designed around its individual coordination concerns: as a result, components of a coordinated system can more easily come and go, having no need to be aware of the global coordination policies, which are charged upon the shared coordination abstractions. In short, tuple-based models featuring behaviour programmability promote *coordination uncoupling* between coordinables, which can be designed so as to coordinate independently of each other since global coordination is delegated to the coordination media.

Associative access to tuples in LINDA tuple spaces is based on a *tuple matching mechanism* which is purely syntactic. Although this might appear as a marginal aspect of tuple-based coordination models, it however imposes to coordinated components a design-time awareness of the structure and content of tuples: ultimately, components coordinated through a tuple space should be designed altogether—thus clearly working against the basic requirements for openness. In order to face this issue, current research in the area of coordination models and languages has focussed on ontology languages to semantically describe information in tuple spaces. The most relevant approaches aimed at augmenting tuple spaces with semantics – going altogether under the name of *semantic tuple space computing* – are presented in [64]. Semantically-enhanced tuple-based coordination models overcame the limitations of standard tuple spaces at least in two ways. On the one hand, they have the potential to free coordinated components of the hassles of a rigidly design-time defined syntactic communication, by charging the coordination abstractions of the burden of semantic interpretation. On the other hand, they promote richer and more expressive forms of communication and – mostly – coordination, where coordination policies can be in principle specified also based on information-based criteria. In short, tuple-based models featuring semantic support promote *semantic uncoupling* between coordinables, since they allow communication and coordination while delegating the common representation of the domain elements and relationships to the shared coordination media.

Behaviour programmability and semantic support in tuple spaces are both essential requirements for open, distributed and knowledge-intensive systems. However, none of the approaches provided by literature accounts for both. Moreover, as already shown by *Artificial Intelligence* literature [99, 51, 83], real-world information is often imprecise and vague, and should be handled as such. Although several works discuss how to represent vague/fuzzy knowledge, we found only the work of Balzarotti et al. [4] exploiting fuzziness to describe knowledge in tuple spaces—there, a tuple space framework called LighTS is presented, where fuzzy templates can be built in terms of fuzzy fields. For example, the template `A(Temperature is Hot, Distance is Far)` can be defined in order to obtain a tuple `A` in which the field `Temperature` and `Distance` are considered respectively *hot* and *far* by following a defined membership function associated to the fuzzy concepts `Hot` and `Far`. However, LighTS does not support semantic matching. Accordingly, seen the importance of the previously highlighted aspects, we aim at providing a new coordination abstraction including the following key elements: *(i)* tuple-based coordination for communication uncoupling, *(ii)* behaviour programmability for coordination uncoupling, and *(iii)* semantic and fuzziness support for semantic uncoupling.

1.2 Contribution

The main contributions of the thesis are:

- The definition of the tuple space model. In particular, we show the model features that are suitable for distributed and open contexts. Then, starting from the original formulation of tuple spaces within the LINDA model, we describe the main extensions of the original model classifying them in two categories that are notable for their impact on the engineering of complex computational systems: models focussing on the *programmability of the behaviour* of the tuple space and models enhancing tuple-based communications with *semantics*.

- The definition of the semantic tuple centre model. In particular, we first show the main objective of the work, that is the design of a new coordination abstraction gathering the features characterising the two kinds of tuple space extensions: *(i)* tuple-based coordination for communication uncoupling, *(ii)* behaviour programmability for coordination uncoupling and *(iii)* semantic and fuzziness support for semantic uncoupling. Then, we show how to reach this objective, first *(i)* starting from tuple centres as our coordination model for ensuring communication and coordination uncoupling and then *(ii)* enriching with semantic and fuzziness the model in order to provide a general-purpose and customisable semantic coordination media for semantic uncoupling, thus obtaining the new coordination abstraction *semantic tuple centre*. In particular, as shown in the following chapters, we choose tuple centres – tuple spaces enhanced with a *behaviour specification* – as a starting point, because, differently from other tuple-based coordination media provided in literature, they: *(i)* provide behaviour specifications that can associate any event possibly occurring in the tuple centre to a set of computational activities called *reactions*; *(ii)* do not rely on the definition of new primitives to add new coordinative behaviours, so they do not require any prior knowledge shared among coordinated components; *(iii)* provides a general-purpose coordination media that can be used to engineer the whole interaction space of a software system; *(iv)* are conceived as runtime first-class abstractions.

  After a description of the tuple centre model, we will provide an abstract architecture of the new coordination abstraction.

- Implementation of the semantic tuple centre model. Starting from the defined semantic tuple centre model, will be described the implementation of the model based on the TuCSoN coordination infrastructure, which is the only infrastructure supporting tuple centres. The implementation in TuCSoN will be evaluate on its current performance in order to understand its applicability in real application contexts.

- Extension of the tuple centre model with fuzziness. We show the important role of the fuzziness in describing the knowledge stored in tuple centres in open context. In particular, we show an extension of the semantic tuple centre model and of the implementation of the new model in TuCSoN.

- Application Scenarios. We show two main important application scenarios in which semantic tuple centres seems to represent a suitable coordination media. The first application scenarios is in the context of e-Health, that is, Healthcare supported by software systems. In particular we will focus on the interoperability of Electronic Health Record (EHR) fragments – medical information that are stored in a digital format over different healthcare institutions – belonging to an environment that is *distributed* and *open* and where the *security support* represents a fundamental requirement to protect the patient privacy. We will show how it is possible to extend the solutions proposed in literature by tacking the semantic version of TuCSoN as inspiration model, in order to augment their effectiveness in building EHR services, in particular as far as interoperability is concerned.

  The second application scenario is in the context of the pervasive services. Addressing this scenario calls for finding infrastructures promoting a concept of pervasive "eternality", namely, changes in topology, device technology and continuous injection of new services have to be dynamically tolerated as much as possible, and incorporated with no significant re-engineering costs at the middleware level [101, 94]. As far as the coordination of such services is concerned, it will increasingly be required to tackle self-organisation (supporting situatedness, adaptivity and long-term accommodation of diversity) as an inherent system property rather than a peculiar aspect of the individual coordinated components. As typical in self-organising computational mechanisms, a promising direction is to take inspiration from natural systems (e.g. physical, chemical, biological, social [94]), where self-organisation is intrinsic to the basic "rules of the game". Focussing on chemical natural systems, we will first shows the concept of chemical tuple spaces [91] – tuple spaces programmed with coordination rules resembling chemical reactions – as suitable coordination media for situated and adaptive pervasive computing [77, 17, 46, 56]. Then, we will show how a distributed architecture for chemical tuple spaces [91] can be implemented in TuCSoN providing semantic ReSpecT tuple centres.

1.3 Thesis Outline

Accordingly, to the above discussions, the reminder of the thesis is organised as follows. Chapter 2 will review the main background of this paper—namely, tuple-based coordination, the concept of tuple space programming, and existing semantic extensions to tuple-based models . Chapter 3 introduces the proposed semantic tuple centre model and its connections to the Description Logics. In particular, we first provide a description of the new model and of its main ingredients. Then, we show an abstract architecture of the coordination abstraction. Chapter 4 will show how to implement the semantic tuple centre model in the TuCSoN infrastructure by extending the supported ReSpecT tuple centres with semantic techniques. Moreover, the chapter will discusses the implementation and evaluates its current performance. Chapter 5 will show a first application scenario in which the TuCSoN infrastructure, semantically extended, it is suitable in order to coordinate EHR fragments. Chapter 6 will provide an extension of the semantic tuple centre model with fuzziness, showing a possible implementation in the TuCSoN infrastructure. Chapter 7 will show a first application scenario in which the TuCSoN infrastructure, semantically extended, it is suitable in order to coordinate pervasive services. Finally, Chapter 8 will concludes the thesis work outlining directions for future works.

Chapter 2

Tuple Spaces

The engineering of open and distributed computational systems mandates for the adoption of a software infrastructure whose underlying model and technology could provide the required level of *uncoupling* among system components. This is the main motivation behind current research trends in the area of coordination middleware to exploit *tuple-based coordination models* in the engineering of complex software systems, since they intrinsically provide coordinated components with *communication uncoupling.* Besides the original formulation of the tuple space model in LINDA, a number of extensions were proposed in literature in order to overcome the limit shown by the model. This chapter, after a brief introduction of the basic elements describing the original tuple space model, surveys the main extensions of the model that are in the direction of the *programmability of the behaviour* of the tuple-based coordination abstraction, and of the enhancing tuple-based communication with *semantics.*

2.1 Basic of Tuple Spaces

Tuple spaces are information spaces structured as sets of tuples, which are structured and ordered chunks of information. In tuple-based coordination models, system components interact, communicate, synchronise and coordinate via tuple spaces by inserting, reading and consuming tuples. According to the *generative communication* paradigm [35], tuples are permanently written in a tuple space through the *out* primitive, then made available to any process for reading/consuming through *rd*/*in* primitives, respectively. As shown in Figure 2.1, tuples are read and retrieved *associatively*: in order to either read or consume a tuple, a *tuple template* – that is, a description of a tuple set – has to be specified. When a tuple matching the template is found in the tuple space, it is returned as the result for the tuple request.

Generative communication and associative access make tuple-based coordination models particularly interesting for the engineering of distributed, open and knowledge-intensive systems. Generative communication promotes *communication uncoupling* among system

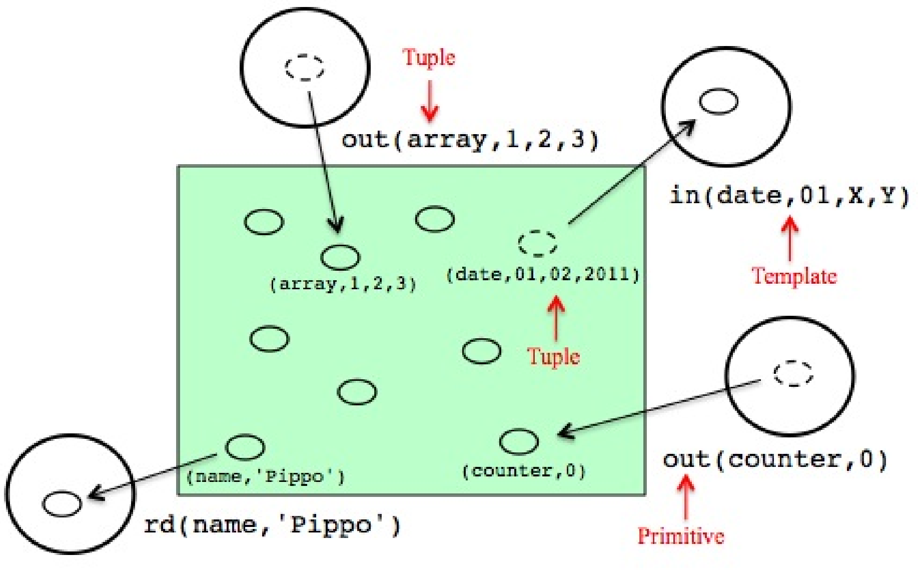


Figure 2.1: The Tuple Space Model

components: tuple producers and consumers can interact with no prior knowledge about each other, thus leading to *space*, *time* and *name uncoupling*—which fit well with open and distributed application scenarios. Associative access to tuples along with synchronisation based on tuple availability in the tuple space promotes *knowledge-based coordination* where system components coordinate based on tuple information content and structure—which fits well with knowledge-intensive application scenarios.

After the original formulation within the LINDA model [35], a number of implementations and extensions have been developed and proposed in the literature. Among the proposals aimed at providing effective technologies supporting the tuple-based coordination model, remarkable examples are Sun's JavaSpaces [33] and GigaSpaces [37]

Many other proposals have instead focussed on extending the tuple-based coordination model beyond its original limitations. Among the many different classes of extended tuple-based models, two are notable for their impact on the engineering of complex computational systems: those focussing on the *programmability of the behaviour* of the tuple-based coordination abstraction, and those enhancing tuple-based communication with *semantics*.

2.2 Behaviour of Tuple Spaces

The behaviour of the original LINDA tuple spaces – represented by their state transition in response to the invocation of the standard coordination primitives – is set once and for all by the model, and cannot be tailored to the specific application needs [67]. As a consequence, any coordination policy not directly supported by the standard behaviour of the coordination abstraction – the tuple space – has to be charged upon system components, which hence grow in complexity—thus hampering the effectiveness of the coordination model especially in open scenarios [70].

In order to overcome such a limitation, a number of tuple-based coordination models and languages have been proposed that extend the original tuple-space model by allowing the behaviour of the tuple spaces to be programmed so as to embed coordination policies within the coordination media. The very notion of *programmable coordination medium* was first explicitly expressed in [20], from where the concept of *tuple centre* was developed. Tuple centres are tuple spaces whose behaviour can be determined through a specification language defining how a tuple centre should react to incoming/outgoing communication events [67]. Unlike tuple spaces, the behaviour of tuple centres can be programmed with *reactions* so as to encapsulate coordination policies within the coordination medium. ReSpecT is a logic-based language allowing the specification of a tuple centre behavior through a set of first-order logic tuples [66]. Since ReSpecT is Turing-equivalent any computable coordination policy required by the specific application scenario can be in principle embedded within a ReSpecT tuple centre.

A number of similar approaches are rooted upon the Java distributed programming framework. IBM's T Spaces [96] are Java-based tuple spaces that can be re-programmed in Java so as to provide coordinated components with the ability to define new communication primitives and associate them to any T Space: thus, interacting entities are supposed to share some knowledge about the syntax and the semantics of new coordination primitives. MARS [14] is a coordination middleware providing a single reactive Java-based space, modelled by extending Sun's JavaSpaces. Tuples in MARS are Java objects, whereas reactions are built as standard Java methods.

Among the declarative approaches, *law-governed Linda* (LGL) [58] allows coordination media to be programmed through a logic-based language. LGL makes it possible to redefine the effect of a communication primitive by associating local proxies to each coordinated component, and making it possible to program them so as to change the semantics of communication operations. On the other hand, EgoSpaces [47] is a middleware built upon the *context* abstraction, allowing an individual component to limit the portion of the environment it interacts with. EgoSpaces tuple spaces are then databases of tuples representing context information. Similarly to LGL, EgoSpaces allows coordinated components to be associated to local *views* by which they can access tuples stored in a tuple space. Such views are built through a declarative specification making it possible to program its reactive behaviour.

The LighTS framework [4] is designed to provide the minimal set of features implementing a standard tuple space, while, at the same time, offering the building blocks required for customising and extending its coordinative behaviour. In particular, LighTS allows the tuple matching mechanism to be redefined so as to trigger reactions, specified by the programmer, when a tuple matching a given template is manipulated through an operation. Lime [60] is an infrastructure retaining the original philosophy and goals of Linda while adapting them to the support of agent mobility through the reactive programming of tuple spaces. A Lime *reactive statement* has the form `T.reactsTo(s,p)`, where `s` is a code fragment containing non-reactive statements to be executed when a tuple matching the pattern `p` is found in tuple space `T`. Finally, TOTA [55] is a tuple-based middleware supporting field-based coordination for pervasive-computing applications. In TOTA, the tuple space behaviour can be programmed by associating to each tuple its *propagation rule* over the network.

Since the behaviour of their coordination media can be programmed to encapsulate coordination policies, the above coordination models make it possible to free coordinated components from the burden of the shared, social aspects of coordination. So, each component can be designed around its individual coordination concerns: as a result, components of a coordinated system can more easily come and go, having no need to be aware of the global coordination policies, which are charged upon the shared coordination abstractions. In short, tuple-based models featuring behaviour programmability promote *coordination uncoupling* between coordinables, which can be designed so as to coordinate indepentendly of each other since global coordination is delegated to the coordination media.

2.3 Semantic in Tuple Spaces

Associative access to tuples in standard tuple spaces is based on a *tuple matching mechanism* which is purely syntactic. Although this might appear as a marginal aspect of tuple-based coordination models, it however imposes to coordinated components a design-time awareness of the structure and content of tuples: ultimately, components coordinated through a tuple space should be designed altogether—thus clearly working against the basic requirements for openness. In order to face this issue, current research in the area of coordination models and languages has focussed on ontology languages to semantically describe information in tuple spaces. The most relevant approaches aimed at augmenting tuple spaces with semantics – going altogether under the name of *semantic tuple space computing* – are presented in [64].

Among them, *Triple Space Computing* (TSC) [29] provides an extension of tuple-based model with Semantic Web technology. In particular, TSC proposes an extension of the classical flat data-model adopted for tuples, relying on RDF to extend the reach of tuple-based coordination to the Semantic Web domain. In the TSC model, tuples are associated with URIs, and can be interlinked so as to form graphs. Moreover, tuples are triples of the form <*subject predicate object*>—as typical of the RDF approach. Among the others, one of the main advantages of adopting Semantic Web technology is represented by the possibility to provide tuple spaces with semantic matching algorithms.

Born as an extension of TSC, *Conceptual Spaces* (CSpaces) [57] is an independent initiative targeted at studying the applicability of semantic tuple space computing in scenarios other than the Web, such as Ubiquitous Computing, EAI, and so on. To this end, CSpaces features a knowledge container, an organisational and a coordination model, a model for semantic interoperability, a security and trust model, a knowledge visualisation model, and an architecture model—for a thorough description, we forward the interested reader to [64].

Semantic Web Spaces [85] has instead been devised as a middleware for the Semantic Web, where clients can exploit Semantic Web data to access and manipulate knowledge, so as to coordinate their activities. Originally conceived as an extension of XMLSpaces [97], Semantic Web Spaces provides tuple spaces able to support the exchange of tuples in the form of RDF triples, relying on RDF Schema reasoning capabilities as far as tuple matching is concerned. As a consequence, the resulting middleware can be regarded as a first step towards the adoption of tuple-based coordination in the Semantic Web.

Finally, sTuples [50] is targeted at pervasive computing settings. There, given the heterogeneous and dynamic nature of pervasive environments, the combined adoption of Semantic Web and tuple spaces have been recognised as a viable solution not only to semantic interoperability issues, but also with respect to temporal and spatial uncoupling. To this end, semantic tuples – based on JavaSpace object tuples – are provided that extend data tuples, and tuple template matching is extended via a semantic matching mechanism on top of object-based matching. In addition, sTuples features agents residing in the tuple space with the goal of performing user-centric services, such as tuple recommendation.

In the overall, semantically-enhanced tuple-based coordination models overcame the limitations of standard tuple spaces at least in two ways. On the one hand, they have the potential to free coordinated components of the hassles of a rigidly design-time defined syntactic communication, by charging the coordination abstractions of the burden of semantic interpretation. On the other hand, they promote richer and more expressive forms of communication and – mostly – coordination, where coordination policies can be in principle specified also based on information-based criteria. In short, tuple-based models featuring semantic support promote *semantic uncoupling* between coordinables, since they allow communication and coordination while delegating the common representation of the domain elements and relationships to the shared coordination media.

2.4 Summary

In this chapter we provided an overview of the tuple space models showing their peculiarity in the context of complex software systems, which are mainly characterised by the *openness* and *distribution* dimensions. In particular, we first shown the original model provided by Linda and its feature of providing *communication uncoupling* among coordinables. Then, we put in evidence the two main limitations of the model: *static behaviour* of the tuple space and *syntactic matching* between tuples and templates. Finally, we provided an overview of the model extensions overcoming such limits adding *coordination* and *semantic uncoupling* among system components.

Chapter 3

Semantic Tuple Centres

The previous chapter provided an overview of the tuple space model and of the several model extensions towards *behaviour programmability* and *semantic support*. Starting from such an overview, this chapter will put in evidence the lack of support of both the extensions and its importance in the context of distributed and open application scenarios. Thus, it will be provided a new coordination abstraction based on the following key elements: *tuple-based coordination*, *behaviour programmability* and *semantic support*. Then, after having compared the different tuple-space-based models provided by the literature, it will be chosen the tuple centre model as starting point for the modelling of the new coordination abstraction. In particular, it will be shown which are the ingredients needed to extend the original tuple centre model in order to support the key elements and its new abstract architecture supporting such ingredients.

3.1 Towards Semantic Tuple Centres

Behaviour programmability and semantic support in tuple spaces are both essential requirements for open, distributed and knowledge-intensive systems—however, none of the approaches reviewed in Section 2.2 and Section 6.3 accounts for both.
This is why in this work we aim at designing a new coordination abstraction including the following key elements as shown in Figure 3.1:

Tuple-based coordination – since it promotes communication uncoupling, that it is suitable for distributed and open contexts.

Behaviour programmability – in order to encapsulate all coordination rules in coordination media and not in coordinables, guaranteeing coordination uncoupling among them.

Semantic support – in order to semantically represent the knowledge stored in tuple spaces supporting for semantic uncoupling. In particular, by introducing an *ontol-*

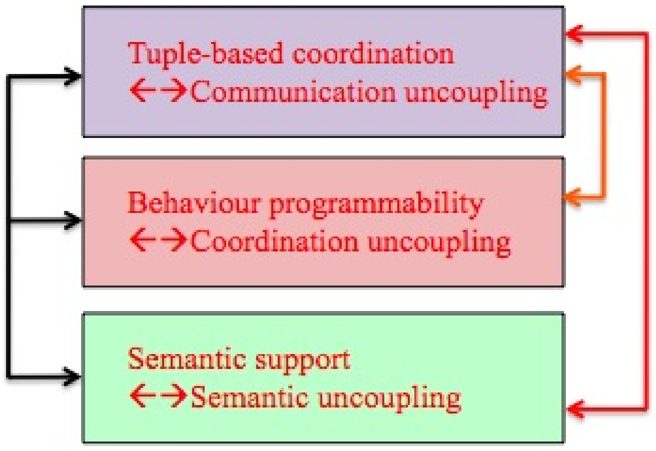

Tuple-based coordination
←→Communication uncoupling
Behaviour programmability
←→Coordination uncoupling
Semantic support
←→Semantic uncoupling

Figure 3.1: Key Elements Composing the New Coordination Abstraction

ogy language [41, 2] to specify a taxonomical knowledge would allow to face: *(i)* heterogeneous problems concerning coordinables that refer to information that is semantically equivalent but has a different syntactic representation, *(ii)* the problem of the communication processes between the software system and users that is complicated by the various ways in which the information is represented, and *(iii)* system dynamic evolution that could be faced by dynamically acquiring, extending and adapting the domain ontology.

Moreover, in order to face the needs of coordination in advanced scenarios like pervasive computing ones [69], our new coordination abstraction should be provided as a *runtime first-class abstraction* [74]. Runtime first-class abstractions are abstractions: *(i)* explicitly accounted in the meta-model of the system-engineering paradigm, *(ii)* "kept alive" through the whole software engineering process – from the analysis to the deployment phase –, *(iii)* enabling both inspection and modification of their current state at runtime, so as to allow for runtime system monitoring and evolution. Runtime first-class abstractions enable software engineers to perform *online engineering*—the capability of supporting system design, development and evolution while the system is running.

Among the several models enhancing tuple spaces with behaviour programmability – as discussed in Section 2.2 – *tuple centres* [67] seems to provide the best level of coordination uncoupling—in particular, in their TuCSoN implementation [70]. A tuple centre is a tuple space enhanced with a *behaviour specification*, defining a tuple centre behaviour in response to communication events in terms of a *reaction specification language*. While LighTS, LIME and TOTA tuple spaces only react when tuples stored in a space match some patterns, a behaviour specification can associate any event possibly occurring in the tuple centre to a set of computational activities called *reactions*. Each reaction can in principle access and modify the current tuple centre state – like adding and remove tuples – and access all the information related to the triggering event—thus allowing for coordination policies to be associated to any sort of event property. Unlike T Spaces, then, tuple centres do not rely on the definition of new primitives to add new coordinative behaviours, so they do not require any prior knowledge shared among coordinated components. Also, the tuple centre model provides a general-purpose coordination media that can be used to engineer the whole interaction space of a software system [61], whereas models like *LGL* and EgoSpaces just focus on the local view of a coordinated component. Finally, the MARS model [14] is the most similar to the tuple centre model: however, MARS tuple spaces are not conceived as runtime first-class abstractions. Furthermore, no specific language for reactions is defined in MARS, which just relies on Java—whereas tuple centres are provided with a general-purpose, logic-based language – ReSpecT – specifically designed for handling any sort of event and tuple space behaviour [66]

As far as semantic support in tuple spaces is concerned, semantic tuple space computing includes the most relevant approaches—as shown in Section 6.3. Semantic tuple space computing was introduced to cope with openness in the context of Semantic Web and Semantic Web Services [64] adopting tuple-based coordination for communication uncoupling, and extending it with semantically-enriched tuple spaces, in order to exploit tuple spaces as repositories of semantic information. For instance, TSC and *Semantic Web Spaces* provide tuple spaces storing *RDF* triples in form of tuples describing respectively Web Services and Web resources. There, tuple spaces are exploited to coordinate Web Services and Web resources with Web slients. Analogous considerations apply to CSpaces and sTuples (see Section 6.3). However, the drawback of semantic tuple space computing is that it focuses on quite a specific context: the Semantic Web (with Web Services). Thus, the adopted solutions are context-dependent and strictly coupled with technologies like RDF and XML, so they are not well suited to work as general-purpose solutions for diverse scenarios like knowledge-intensive and pervasive computing systems.

Accordingly, in this work we aim at providing a *general-purpose semantic tuple space model* with no assumptions about the application context, which would preserve its conceptual coherence independently of the use of either semantic tuples or standard ones. In particular, as shown in the following, *(i)* we start from tuple centres as our coordination model for ensuring coordination uncoupling, then *(ii)* we semantically enrich the model in order to provide a general-purpose and customisable semantic coordination media for semantic uncoupling.

3.2 Semantic Tuple Centre Model

From an information-oriented viewpoint, a tuple centre has a simple and natural interpretation as a knowledge repository structured as a set of tuples, whose behaviour is programmable through a set of reaction specifications. Tuples may be seen as representing objects of the application domain, whose meaning is described by an ontology—that is, in terms of concepts and of relations among them [3]. Accordingly, the ingredients of semantic tuple centres are:

Domain ontology – an ontology describing domain concepts and relationships attached to a tuple centre, allowing the tuples stored there to be semantically interpreted.

Semantic tuples – a tuple representing an individual that can be semantically interpreted by means of the domain ontology associated to the tuple centre.

Semantic tuple templates – tuple templates used to retrieve semantic tuples, consisting in specifications of sets of domain individuals described by the domain ontology.

Semantic primitives – operations (*in*, *rd* and *out*) representing the language whereby system components can read, consume and write tuples representing knowledge described by means of a domain ontology.

Semantic reactions – sets of computational activities within a tuple centre defined through a reaction specification language.

Semantic matching mechanism – algorithms checking the relationships between individuals and concepts described by tuples and tuple templates in the execution of coordination primitives and reactions.

To formally define the tuple centre model, a family of knowledge representation formalisms called *Description Logics* (DL) are used [3]. As formal logic languages with well defined semantics, DL allow *(i)* ontologies – as well as information using vocabulary defined by ontologies – to be shared and exchanged without disputes as to precise meaning, and *(ii)* automated reasoning techniques over ontologies that can be exploited to face the dynamic evolution of the vocabulary used to describe the knowledge. In particular, the DL known as *SHOIN(D)* represents a very expressive DL that is the theoretical counterpart of *OWL DL*, that is, one of the three species of OWL [42]—the standard ontology description language for Semantic Web and the standard *de facto* in many semantic applications. Being a standard, OWL fits well the openness requirement: this is why we exploit SHOIN(D) to enrich tuple centres with domain ontologies and objects. Accordingly, the following DL components are used in the formal definition of the semantic tuple centre model: *TBox*, *ABox* and the *reasoning service.*

Domain ontology An application domain ontology can be formally defined through a TBox, since the TBox includes the so-called terminological axioms: *concept descriptions*, denoting meaningful sets of individuals of the domain; *role descriptions*, denoting relationships among individuals; and the *taxonomy description* of concepts and roles. In particular, in SHOIN(D) TBox concepts can be of the following kinds [3]: $\top$ is the set of all objects, $\bot$ is the void set, $C \sqcup D$ is union of concepts, $C \sqcap D$ is intersection, $\neg D$ is negation, $\{i_1, \ldots, i_n\}$ are nominals, that is a set of individuals, $\forall R.C$ is the set of objects that are in relation through role R with only objects belonging to concept C, $\exists R.C$ is the set of objects that are in relation through role R with at least one object belonging to concept C; and $\leq nR$ is the set of objects that are in relation through role R with no more than n objects (and similarly for concepts $\geq nR$ and $= nR$). DL formalism also provides the constructs $\sqsubseteq$ and $\equiv$ for expressing respectively the concept or role inclusion and equality, thus describing a taxonomy among concepts and roles [3]. For example, the following SHOIN(D) assertions represent a part of the ontology describing the *car domain* [12]:

```
(1)  Maker ⊑ ⊤
(2)  Car ⊑ (=1 hasMaker)
(3)  (=1 hasMaker) ⊑ Car
(4)  ⊤ ⊑ ∀hasMaker.Maker
(5)  CityCar ⊑ Car
```

The above axioms describe the concept `Maker`, the concept `Car` with the mandatory and functional relation `hasMaker` with the concept `Maker` – i.e., each car has precisely one maker – and the concept `CityCar` as a kind of `Car`.

Semantic tuples Semantic tuples can be formally defined through an ABox, since the ABox defines axioms to assert specific domain objects and their properties: they can be of kind `C(a)`, declaring the individual `a` and the concept `C` it belongs to, and of kind `R(a,b)`, declaring that role `R` relates individual `a` with `b`. In the *car domain* example, the ABox could include the following axioms:

```
(6)  Car(f40)           hasMaker(f40,ferrari)
(7)  CityCar(fiat500)   hasMaker(fiat500,fiat)
```

Accordingly to the TBox axioms `(1-5)`, the above ABox axioms define two individual respectively named `f40` and `f500`. The first belongs to the concept `Car` whereas the second belongs to the concept `CityCar`. Both individuals are in relation with respectively the individual `ferrari` and the individual `fiat` both belonging to the concept `Maker`. In order to represent an individual in form of tuple through the ABox formalism – so that it can be interpreted by means of a TBox-based ontology – a language is required that could express the following information: the name of the individual, the concept to which the individual belongs, and the set of relations in which the individual is involved.

Semantic tuple templates Since semantic templates are specifications of set of domain individuals described by a domain ontology, a semantic template becomes a description of the set of individuals one is interested in expressed in TBox formalism. For instance, referring to the *car domain* example, if we are interested in the set of cars that have *Ford* as a maker, the tuple template should provide a concept description expressed in SHOIN(D) like `Car ⊓ (∃hasMaker.ford)` d. Thus, in order to describe an individual set specification in form of a tuple template, a language is required with the same expressive power of the TBox formalism.

Semantic primitives Semantic primitives represent the language whereby system components can read, consume and write knowledge described by means of a domain ontology. Extending the tuple centre model with semantic techniques also requires the semantic of the *out* primitive to be suitable extended. In particular, since the knowledge stored in the semantic tuple centre must be always consistent with the domain ontology, in face of the writing primitive it is required to check – by exploiting the DL reasoner – the consistency of the semantic tuple to be written in the tuple centre, with the domain ontology—here intended as Abox plus TBox. This means that semantic tuple centres also requires to revisit the behaviour of the basic tuple centre primitives. The first extension concerns all the primitives *out*, *in* and *rd*, and consists in checking whether the roles and concepts associated to the individual and concept descriptions, respectively, exist in the domain ontology. Thus, for example if we exploit the primitive *out* with the semantic tuple `fiat500:'CityCar'( hasMaker :  fiat)`, the existence of concept `CityCar` and role `hasMaker` has to be checked. Moreover, while an *out* in a tuple centre always succeeds, an *out* in a semantic tuple centre may fail. In particular, the *out* could fail for two different reasons. First, since a semantic tuple represents an individual, and the individual is unique in the ontology, an *out* of a tuple first requires to check whether the individual represented by the tuple already exists in the knowledge base—in case, the *out* fails. Then, since the knowledge stored in the semantic tuple centre should always be consistent with the domain ontology, an *out* requires to check the consistency of the semantic tuple with the knowledge base, by exploiting the DL reasoner.

Semantic reactions A reaction specification language [67] *(i)* enables the definition of computations within a tuple centre, called reactions, and *(ii)* makes it possible to associate reactions to events occurring in a tuple centre. adding or reading / removing tuples. Accordingly, a reaction specification describes an event specification E describing the set of events Ev for which reaction R has to be executed. In particular, E has to contain two sorts of information: the primitive (*in*, *rd* or *out*) to be intercepted, and an individual set specification, describing either the possible concept descriptions in an *in*/*rd* primitive, or the possible individuals in an *out* primitive. For example, taking axioms `(1-6)`, E could contain the concept description `Car`. If E refers to a primitive *in* or *rd*, R will be executed if the concept description in the semantic tuple template is a sub-kind of `Car`—as for the concept description `CityCar ⊓ (∃hasMaker.fiat)`. Whereas, in case E refers to the primitive *out*, R will be executed if an individual of the kind `Car` is inserted in the tuple centre—as for the individual `fiat500`. As far as reactions are concerned, besides accessing all the information related to the triggering communication event, they can read, remove and write tuples from/to the tuple centre. Accordingly, like coordinated components, reactions can contain coordination primitives to access and modify the semantic knowledge stored in the tuple centre, through semantic tuples and semantic tuple templates.

Semantic matching mechanism In case of tuple centre primitives, the semantic matching mechanism represents the means by which to identify tuples matching templates. In a semantic tuple centre this means that the matching mechanism has to identify and retrieve the set of individuals specified by a concept description by means of the domain ontology. In case of reactions, the semantic matching mechanism has a twofold means. If E describes a writing event, then the matching mechanisms between E and Ev consists in checking *(i)* if Ev is a writing event and *(ii)* if the individual contained in Ev belongs to the concept described in E. Instead, if E describes a consuming/reading event, then the matching mechanisms between E and Ev consists in checking *(i)* if Ev is a consuming/reading event and *(ii)* if the concept description contained in Ev is a sub-concept of the concept described in E.

In order to implement the matching mechanism above, the reasoner service used by any DL-based system to execute reasoning tasks on ABox and TBox can be used. On an ABox, one can execute *instance checking* – verifying whether a given individual is an instance of a specified concept – and *retrieval*—finding the individuals in the knowledge base that are instances of a given concept. Instance checking can be used in case of reaction to an *out*. Instead, retrieval can be used for semantic matching between tuples and templates. On a TBox, one can check the *subsumption* of two concepts `C` and `D`—which can be used in the case of reactions to *in*/*rd*.

3.3 Abstract Architecture

After having defined the ingredients of a semantic tuple centre, it is now possible to outline its the abstract architecture.
As shown in Figure 3.2, besides the *tuple space* storing tuples, the *reaction specifications* defining the tuple space behaviour and the *reaction engine* managing the tuple space behaviour, the *semantic engine* is added to the tuple centre architecture in order to provide the semantic support. In particular, the *semantic engine* should contain a DL reasoner by which it is possible – through the domain ontology related to the semantic tuple centre – to interpreter and reason about the semantic knowledge stored in the tuple space and to interpreter and enact the semantic reaction specifications. Then, the *semantic engine* interacts with both the *tuple space* and the *reaction engine* performing the following tasks:

individual assertion – insert a new individual `a` in the ABox and checks if the ABox is consistent with the new individual. The ABox can or not coincide with the tuple

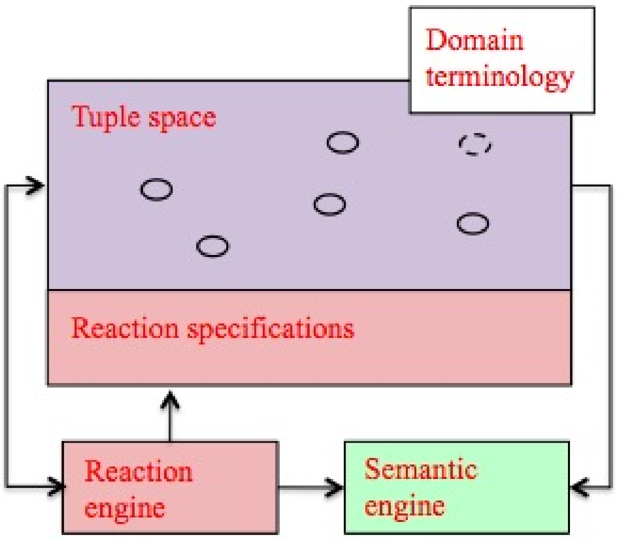


Figure 3.2: Abstract Architecture of a Semantic Tuple Centre

space; it depends if we exploit an internal reasoner engineered specifically for the tuple space or an external DL reasoner with its own ABox representation.

individual deletion – deletes individuals from the ABox.

instance checking – checks if an individual `a` belongs to a concept `C`.

instance retrieval – retrieves all the individuals belonging to a concept `C`.

subsumption checking – checks if a concept `C` subsumes a concept `D`.

Individual assertion and instance retrieval are exploited respectively in face of the primitives *out* and *in/rd* towards the tuple space. In particular, in face of the primitive *in*, besides the instance retrieval, the individual deletion is exploited in order to consume the selected semantic tuple. Then, the instance and subsumption checking are exploited in order to select semantic reactions to be activated (see the *semantic matching mechanism* in Section 3.2).

3.4 Summary

In this chapter we provided the ingredients needed to model a new coordination abstraction based on the following key elements: *tuple-based coordination*, *behaviour programmability* and *semantic support*. The new abstraction overcomes the lacks of support of both *behaviour programmability* and *semantic support* shown by the tuple-space-based models provided in literature. Then, among such different models, the tuple centre model was chosen as starting point for the modelling of the new coordination abstraction. In particular, it was shown the ingredients needed to extend the original tuple centre model in order to support the key elements of the new coordination abstraction and its new abstract architecture supporting such ingredients.

Chapter 4

Semantic Tuple Centres in TuCSoN

The previous chapter provided a new coordination abstraction modelled as an extension of the tuple centre model with the *semantic support*. Starting from the semantic tuple centre model, this chapter aims at describing the implementation of the model based on the TuCSoN coordination infrastructure. TuCSoN is the only infrastructure supporting tuple centres; in particular the infrastructure exploits the ReSpecT tuple centres, that are logic-based programmable tuple spaces. Thus, after having briefly described the TuCSoN infrastructure and the ReSpecT tuple centres, in this chapter it will be described the implementation of semantic tuple centre in TuCSoN by extending the ReSpecT tuple centres towards the model described in the previous chapter. Then, the implementation in TuCSoN will be evaluate on its current performance. Finally, in this chapter it will be outlined the main research directions in order to improve the current implementation of the semantic tuple centres.

4.1 TuCSoN Infrastructure

TuCSoN (Tuple Centres Spread over the Network) [70] is a java-based coordination infrastructure that manages the interaction space of a distributed software system by means of *tuple centres* [67]. From the topology point of view, tuple centres are distributed and hosted in TuCSoN *nodes*, defining the TuCSoN coordination space [18]. In particular, the topological model of TuCSoN classifies nodes as *places* and *gateways*—as shown in Figure 4.1. The former represent the nodes hosting tuple centres used for specific applications/systems need, from supporting coordination activities to hosting information or simply enabling software components communication. The latter provide instead information about a the set of places belonging to a single domain—thus avoiding a single and centralised repository, which is unfeasible in complex and large environments. A *domain* is the set of nodes composed by the gateway and the places for which it provides information. A place can be part of different domains and a gateway can be a place in its turn. The overall picture of the TuCSoN topology is provided in Figure 4.1.

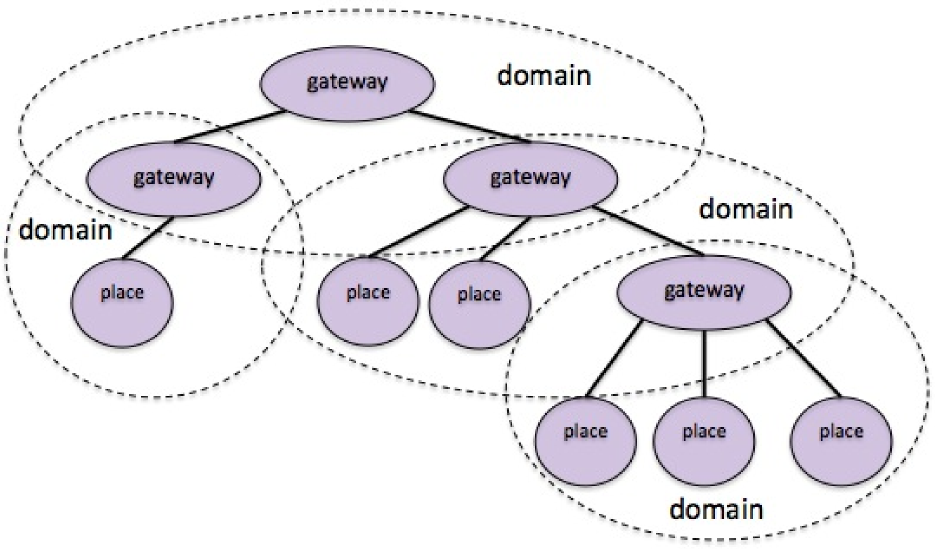


Figure 4.1: Topology of the TuCSoN Coordination Space

Besides topology, the key features of the TuCSoN architecture are: *(i)* a coordination model based on *programmable tuple spaces*, that is tuple centres; *(ii)* the use of the *organisation* and *RBAC* models: the former is exploited to describe the system structures and their relationships, the latter is exploited to model security aspects; and *(iii)* the *online engineering* approach used to support the corrective/adaptive/evolutive maintenance of software systems.

4.1.1 Behaviour in Tuple Centres

The tuple centres behaviour can be determined through a *behaviour specification*, defining how a tuple centre should react to incoming/outgoing coordination events [67]. The behaviour specification can be expressed in terms of a *reaction specification language* that associates any communication event possibly occurring in the tuple centre, to a set of computational activities called *reactions*. Each reaction can access and modify the current tuple centre state by adding or removing tuples and access all the information related to the triggering communication event such as the performing software component, the operation required and the tuple involved. So, differently from tuple spaces, tuple centres represents *general-purpose* and *customisable* coordination media that can be programmed with reactions tailored to the application needs.

TuCSoN exploits ReSpecT tuple centres [67, 66]. ReSpecT adopts a tuple language based on first-order logic, where a tuple is a logic fact, any unitary clause is an admissible tuple template, and unification is the tuple matching mechanism. ReSpecT reactions are defined through logic tuples, too. A *specification tuple* is of kind `reaction(E,G,R)`. It associates a communication event described through `E`, to the reaction `R`. `G` represents a set of conditions that has to be satisfied in order to execute a reaction `R`, if the incoming event matches `E`. A reaction is defined as a transactional sequence of *reaction goals*, which may access properties of the occurred communication event, perform simple term operations, and manipulate tuples in the tuple centre. In order to implement *logic tuple centres* ReSpecT exploits tuProlog [22], a light-weight Prolog system which is java-based.

ReSpecT tuple centres provides two main advantages. Since they are *logic* tuple centres, they make it possible to spread intelligence through the system where needed, for example by exploiting cognitive agents [95]. Also, ReSpecT is Turing-equivalent [21], so any computable coordination law can be encapsulated into the coordination medium. In [66], a complete example of use of reaction specification is discussed.

4.1.2 Organisation & Security

Jennings [45] refers to *organisation* as a tool helping software engineers managing complexity of software system development. Organisation allows the interrelationships between the various components of the system to be defined and managed, by specifying and enacting organisational relationships. In this way, engineers can group basic components into higher-level unit of analysis, and suitably describe the high-level relationship between the units themselves.

TuCSoN exploits an organisational model that is *role-based* [68]. Organisation and coordination are strictly related and interdependent issues. Organisation mainly concerns the structure and the structural relations of a system—i.e. the static issues of the agent interaction space. Coordination mainly concerns the processes inside a system – i.e. the dynamic issues of the agent interaction space –, often related to roles that usually frame agents position in the structure of the system organisation. Moreover, coordination is strictly related to *security*, being both focused on the government of interactions inside a system, however according to two different (dual) viewpoints: normative for security, constructive for coordination [18]. Whereas security focuses on preventing undesired/incorrect system behaviours – which may result in problems like denial of services or unauthorised access to resources – coordination is concerned with enabling desirable/correct system behaviours, typically the meaningful, goal-directed interaction between different system components. Due the relations among coordination, organisation and security, TuCSoN exploits an unique and coherent conceptual framework to manage the complexity of modelling such three dimensions [68].

The TuCSoN conceptual framework is represented by an extended version of the *Role-Based Access Control* (RBAC) model [79] – as shown in Figure 4.2 – called RBAC-MAS [93]. The model interprets an *organisation* as a set of *societies* composed by software components playing certain *roles* according to the *organisation rules*, where each role has an associated set of *policies*. Organisation rules define two types of relationships among roles: *(i) component-role relationship*, through which it is possible to specify whether

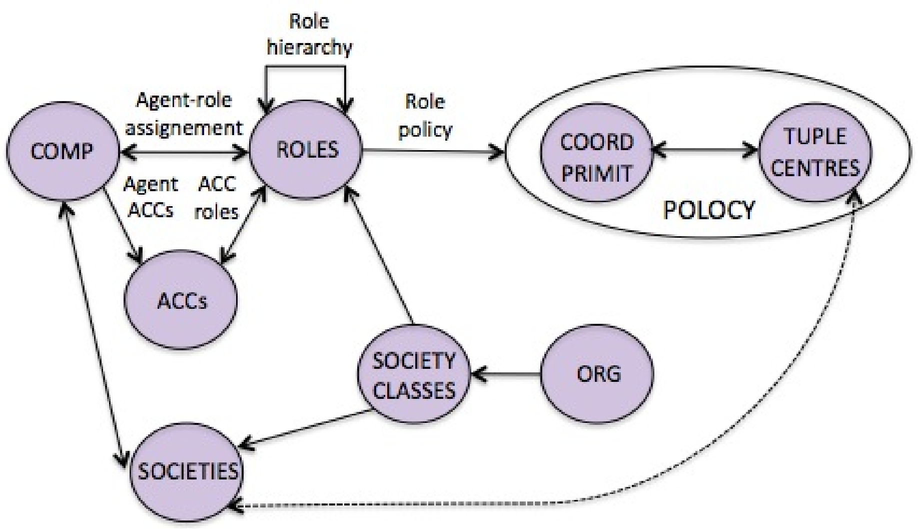


Figure 4.2: Logical Levels in which the Coordination Middleware can be Structured

a specific component is allowed (or forbidden) to assume and then activate a specific role inside the organisation; *(ii) role-role relationship*, through which it is possible to specify structural dependencies among roles, so as to further define constraints on dynamic component role-activation.

A *policy* represents an admissible interaction protocol between the associated role and the rest of the organisation. An *ACC* (Agent Coordination Context [65]) represents an entity contracted by a component, on the basis of its identity, when it enters the organisation. The ACC is then released to components and used from components in order to interact with the resources (here, the tuple centres) belonging to a specific organisation. The interaction is enabled and ruled by the ACC in accordance with the rules and policies defined by the organisation.

From a topology point of view, an organisation is mapped onto a domain (including the linked domains or sub-domains). The description of the structures and rules characterising the organisation are stored and managed dynamically in a specific tuple centre, called `$ORG(OrgID)` – where `OrgID` is the organisation identifier –, hosted in a gateway node of the domain. The `$ORG` tuple centres host then information about societies, roles, components and the related relationships defined for the domain, represented by the gateway and its places.

4.1.3 Online Engineering

The openness of software systems calls for keeping the abstractions *alive* [74]. Alive abstractions are defined in an explicit way in the meta-model of the system-engineering paradigm. Moreover, they are "kept alive" through the whole engineering process of a software system—from the analysis to the corrective/adaptive/evolutive maintenance phase. Such abstractions enable the inspection of their current state at runtime, so as to allow dynamic monitoring of system components that they model, and their creation and modification, so as to allow a dynamic evolution of system components. By exploiting such kind of abstractions, software engineers are enabled to perform *online engineering* [74], that is, the capability of supporting system design, development and test, debugging and evolution while the system is running.

Tuple centres are modelled and built as alive abstractions. Accordingly, TuCSoN allows the runtime maintenance of both coordination laws and organisation structure and rules. In particular, it is possible to maintain and evolve the coordination laws at runtime by inspecting and creating tuple centres, and by modifying their state or behaviour. Then, it is possible to maintain and evolve the organisation model since the organisation structure and rules are reified as knowledge encapsulated in the tuple centre `$ORG`.

By means of its "alive abstractions", TuCSoN allows in principle both humans and (intelligent) software components to maintain and develop a software system. In order to support humans, TuCSoN provides the *Inspector* tool [23] enabling software engineers to first design and then observe and act on system structures and processes at runtime, working upon abstractions adopted and exploited for the design of a system. Besides, TuCSoN also provides intelligent agents with the API needed to create, inspect and modify tuple centres. In particular, since TuCSoN exploits ReSpecT tuple centres – which are *logic* tuple centres – it is possible to exploit agents capable of symbolic reasoning in order to autonomously maintain the structures.

4.2 Designing Ingredients for ReSpecT Semantic Tuple Centres

Domain Ontology Semantic tuple centres are formally defined through SHOIN(D), a very expressive DL representing the theoretical counterpart of OWL DL [42]. Since OWL is the W3C standard ontology description language for the Semantic Web, and the standard de-facto for semantic applications in general, we adopt OWL as the ontology language for the domain ontologies associated to ReSpecT semantic tuple centres in TuCSoN. While for the details of OWL we forward interested readers to [42], we provide an example of a mapping between SHOIN(D) and OWL DL by exploiting the following example provided in Section 3.2 about the *car domain*.

```
(1)  Maker ⊑ ⊤
(2)  Car ⊑ (=1 hasMaker)
(3)  (=1 hasMaker) ⊑ Car
(4)  ⊤ ⊑ ∀hasMaker.Maker
(5)  CityCar ⊑ Car
```

There, SHOIN(D) assertion `Maker` $\sqsubseteq \top$, defining the concept `(1)`, in OWL DL becomes:

```
<owl:Class rdf:ID="Maker"/>
```

where the construct `Class` allows an ontology concept to be defined. The concept `Car` with the mandatory relation `hasMaker` with the concept `Maker` – as with the SHOIN(D) assertions `(2-5)` – in OWL DL becomes:

```
<owl:Class rdf:ID="Car"/>
<owl:ObjectProperty rdf:ID="hasMaker">
     <rdf:type rdf:resource=
            "http://www.w3.org/2002/07/owl#FunctionalProperty"/>
     <rdfs:domain rdf:resource="#Car"/>
     <rdfs:range rdf:resource="#Maker"/>
</owl:ObjectProperty>
```

where the construct `ObjectProperty` represents the relation between concepts, and the construct `type` describes the kind of relation—here, functional [3]. Finally, the SHOIN(D) assertion `(5)` stating that concept `CityCar` is a sub-concept of `Car` would be written in OWL DL using the construct `subClassOf` as follows:

```
<owl:Class rdf:ID="CityCar">
     <rdfs:subClassOf rdf:resource="#Car"/>
</owl:Class>
```

Semantic Tuples In order to describe semantic tuples as domain individuals, a description language is needed to specify, in the form of a logic term, *(i)* the *individual name*, *(ii)* the *concept* which the individual belongs to, and *(iii)* the *set of relations* involving the individual. For simplicity, the only sort of concept description supported here is the *concept name*—so, excluding concept compositions [3]. Accordingly, the language for tuples is defined as follows:

```
Individual ::= iname ‘:’ C
C ::= cname | cname ‘(’ R ‘)’
R ::= rname ‘:’ V | rname ‘in’ ‘(’ Vset ‘)’ | R ‘,’ R
Vset ::= V | V ‘,’ Vset
V ::= iname | string | int | float
```

where `iname`, `cname`, and `rname` represent respectively the individual name, concept name, and role name—while `string`, `int` and `float` represent respectively the datatypes of kind string, int and float. So, for instance, the tuple

```
f550 : ‘Car’ (hasMaker : ferrari, hasMaxSpeed : 285,          (8)
              hasColour in (red, black))
```

would define an individual named `f550` belonging to concept `Car`, involved in three relations: *(i)* `hasMaker` with the individual `ferrari`, *(ii)* `hasMaxSpeed` with the int value `165`, and *(iii)* `hasColour` with strings `red` and `black`. The corresponding logic term using the `:` and `in` operators in the tuProlog logic engine used by ReSpecT would look like

```
‘:’(f550, ‘Car’(‘:’(hasMaker, ferrari), ‘:’(hasMaxSpeed, 285),
                in(hasColour, ‘,’(red, black))))
```

Semantic Templates In order to describe semantic templates as concept descriptions we need a language that allows to express, in the form of a logic term, a tuple template as a description in the SHOIN(D) TBox formalism. Moreover, since a tuple template in ReSpecT can specify arguments as either input or output, the new language has to do the same. The language for tuple template is as follows:

```
C ::= ‘All’ | ‘None’ | cname | C ‘and’ C | C ‘or’ C | ‘not’ C | D |
      ‘{’ [ iname { ‘,’ , iname } ] ‘}’ | C ‘(’ D ‘)’ | ‘(’ C ‘)’ |
      ‘String’ | ‘Int’ | ‘Float’
D ::= F | ‘exists’ F | ‘only’ F | M
F ::= R ‘in’ C | R ‘:’ I | R ‘:’ I | R ‘:’ Msymb N |
      R ‘:’ ‘eq’ string | R
M ::= ‘#’ R Msymb N
R ::= rname | rname ‘/’ vname
Msymb ::= ‘gt’ | ‘lt’ | ‘geq’ | ‘leq’ | ‘eq’
```

where `iname`, `cname`, and `rname` represent respectively the individual name, concept name, and role name. The expression `rname ‘/’ vname`, where `vname` represents a variable name, associates role `rname` with an output argument.

By the grammar above, any SHOIN(D) DL description could be expressed: only the *inverse role* [3] is not supported.
In particular, Figure 4.3 shows the mapping between the main constructs of SHOIN(D) and the constructs introduced by our grammar. Thus, for example, we can define the following tuple template with only input arguments:

```
‘Car’ and (exists hasMaker : ford)
```

in order to obtain an individual of kind `Car` in relation of kind `hasMaker` with the individual `ford`—which corresponds to `Car` ⊓ ∃`hasMaker.ford` in SHOIN (D) syntax. If we are not interested in a particular `Maker`, like `ford`, but would like to know which is the `Maker` associated with the individual selected through the template, we could use an output argument with the expression `rname ‘/’ vname` as follows:

SHOIN(D)	*Language Expression*
$\top$	`All`
$\bot$	`None`
$C \sqcap D$	`C and D`
$C \sqcup D$	`C or D`
$\neg\ C$	`not C`
$\forall\ R.C$	`only R in C`
$\exists\ R.C$	`exists R in C`
$>n\ R,\ \leq n\ R$	`# R lt n, # R leq n`
$>n\ R,\ \geq n\ R$	`# R gt n, # R geq n`
$=n\ R$	`# R eq n`
$\{a_i, \ldots, a_n\}$	{iname$_i$,...,iname$_n$}
concrete domains	`String, Int, Float`
concrete domain operators	`String: eq.  Int, Float: eq, lt, leq, gt, geq.`

Figure 4.3: Language Mapping

```
'Car' and (exists hasMaker / X in 'Maker')                    (9)
```

Thus, besides an individual matching the given concept description, the related `Maker` is retrieved. The corresponding logic term, once the operators defined in the grammar (e.g. `and` and `or`) are added to the tuProlog engine, would look like

```
and('Car', exists(':'(hasMaker, ford)))
```

Semantic Primitives Besides extending the semantic of the coordination primitives, as discussed in Section 3.2, the primitive language should be extended to include in order to *(i)* include both semantic and syntactic primitives, and *(ii)* to obtain an individual as a result from a semantic template. So, for instance, the two primitives below represent "semantic" versions of *out* and *in*:

```
out(semantic fiat500: 'CityCar'( hasMaker : fiat))
in(semantic Result matching ('CityCar' and (exists hasMaker : fiat)))
```

For all primitives (*out*, *in* and *rd*), the keyword `semantic` is used in order to discriminate a semantic primitive from a syntactic one. Also, a variable – called `Result` in the example – is used in *in* and *rd* primitives along with the `matching` keyword in order to unify the variable with the individual resulting from the execution of the primitives *in* and *rd*.

Semantic Reactions From a semantic point of view, in a ReSpecT reaction `reaction(E, (G, R))`, `E` represents a specification of a coordination primitive related to a concept description (*in*/*rd*) or related to a domain individual (*out*). Accordingly, besides the name of the primitive invoked (*out*, *in* or *rd*), `E` should contain a concept description expressed in terms of the semantic template language. For example, considering axioms `(1-7)`, `E` could contain the concept description `Car`, so for instance a reaction to the primitive *in* could look like:

```
reaction( in(semantic ‘Car’), ..., ... )
```

The reaction could be triggered by a primitive *in* containing for example the concept description `CityCar` ⊓ (`∃hasMaker.fiat`), that is

```
in(semantic Result matching (‘CityCar’ and (exists hasMaker : fiat)))
```

Instead, the following reaction:

```
reaction( out(semantic ‘Car’ and (exists hasMaker : fiat)), ..., ...)
```

would be triggered when an individual of kind `Car` in relation `hasMaker` with the individual `fiat` is inserted in the tuple centre, e.g., by the invocation:

```
out(semantic fiat500: ‘CityCar’( hasMaker : fiat))
```

inserting the individual `fiat500` in the semantic tuple centre.

The reaction guard `G` represents a set of conditions to be satisfied in order to execute a reaction `R` if the current event `Ev` matches `E`. Since `G` represents a set of constraints on `Ev` and `E`, ReSpecT is extended so that the guard could contain a concept description in the semantic template language—again denoted by the keyword `semantic`, as in the following reaction:

```
reaction( out(semantic ‘CityCar’), semantic (exists hasMaker : fiat), ...)
```

Finally, also the ReSpecT reaction language needs to be extended, since it can read, remove and write tuples from/to the tuple centre, now including semantic tuples. Thus, `R` can now contain semantic primitives, as in the following example:

```
reaction( out(semantic ‘CityCar’), semantic (exists hasMaker : ford), (
          rd (semantic ford matching ‘Maker’ and (exists hasCars / N in Int)),
          N1 is N + 1,
          out(semantic ford : ‘Maker’(hasCars : N1)) ))
```

There, if an incoming event matches with `out(semantic ‘CityCar’)`, and the guard `(semantic (exists hasMaker : ford))` is satisfied, then the individual `ford` is updated with a new value for the relation `hasCars`.

Fuzzy Matching Mechanism Here we only focus on In order to extend ReSpecT with semantic techniques, a semantic matching mechanism needs to be integrated with logic unification. According to the semantic tuple centre model, when a semantic reading/consuming primitive is performed, the matching mechanism should allow an individual matching the semantic template to be identified and retrieved. Instead, within reactions the semantic matching mechanism should work in two ways. If `E` describes a writing event, then matching `E` and `Ev` consists in checking *(i)* if `Ev` is a writing event, and *(ii)* if the individual contained in `Ev` belongs to the concept described in `E` and in `G`. Whereas, if `E` describes a consuming/reading event, then matching `E` and `Ev` consists in checking *(i)* if `Ev` is a consuming/reading event, and *(ii)* if the concept description contained in `Ev` is a sub-concept of the concept described in `E` and in `G`.

Such a matching mechanism could be easily obtained, in principle, by exploiting the reasoning services provided by any DL reasoner. In particular, we need *(i) instance retrieval*, finding the individuals in the knowledge base that are instances of a given concept, *(ii) instance checking*, verifying whether a given individual is an instance of a specified concept, and *(iii) subsumption checking* between two concepts. Several reasoners are available in the literature—the most popular being RACER [39], Pellet [82], FACT++ [86], KAON2 [59] and HermiT [81]. Pellet represents the most complete reasoner and has competitive performance. In particular, Pellet is Java-based like TuCSoN and ReSpecT, and as well as FACT++, KAON2 and HermiT, is free and open-source. Moreover, like FACT++ and HermiT, it supports SHOIN(D), but unlike them it also support conjunctive queries – an expressive formalism for querying DL knowledge bases – through the SPARQL language [73], a W3C Candidate Recommendation as query language for RDF—so, suitable for querying OWL ontologies.

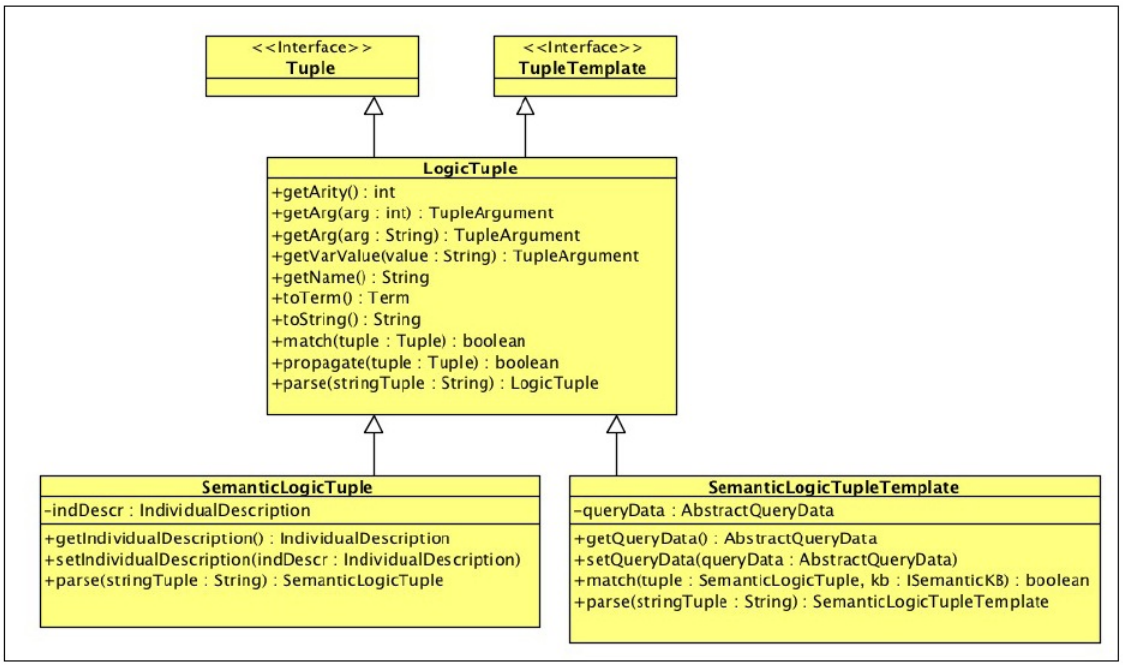
So, we choose Pellet as the reasoner for implementing the semantic matching mechanism in ReSpecT semantic tuple centres. In particular, we exploit the conjunctive queries through SPARQL in order to perform *instance retrieval* when serving *in* and *rd* primitives, *instance checking* to trigger reactions to *out* primitive, and *subsumption checking* to trigger reactions to *in* and *rd* primitives. As a consequence, the realisation of the semantic matching mechanism in ReSpecT requires a generator that builds SPARQL queries starting from semantic templates, event and guard descriptions.

4.3 System Architecture

In order to enable the semantic support in the TuCSoN infrastructure, two main extensions are needed to be apported: *(i)* the support of the new language used to interact with semantic tuple centres and to specify the semantic tuple centre behaviour, as defined in Section 4.2; *(ii)* an extended version of the ReSpecT framework, in order to provide semantic tuple centres as modelled in Section 3.3.

4.3.1 Supporting the Semantic Tuple Centre Language

In TuCSoN and ReSpecT, tuples and templates are both modelled as logic terms through the class `LogicTuple` (see Figure 4.4). From a semantic viewpoint there is a difference



<<Interface>>
Tuple
<<Interface>>
TupleTemplate
LogicTuple
+getArity() : int
+getArg(arg : int) : TupleArgument
+getArg(arg : String) : TupleArgument
+getVarValue(value : String) : TupleArgument
+getName() : String
+toTerm() : Term
+toString() : String
+match(tuple : Tuple) : boolean
+propagate(tuple : Tuple) : boolean
+parse(stringTuple : String) : LogicTuple
SemanticLogicTuple
-indDescr : IndividualDescription
+getIndividualDescription() : IndividualDescription
+setIndividualDescription(indDescr : IndividualDescription)
+parse(stringTuple : String) : SemanticLogicTuple
SemanticLogicTupleTemplate
-queryData : AbstractQueryData
+getQueryData() : AbstractQueryData
+setQueryData(queryData : AbstractQueryData)
+match(tuple : SemanticLogicTuple, kb : ISemanticKB) : boolean
+parse(stringTuple : String) : SemanticLogicTupleTemplate

Figure 4.4: Semantic Logic Tuples and Templates

between tuples and templates (see Section 3.2): a semantic tuple represents an individual, whereas a semantic template represents a concept description. Thus, semantic tuples and templates have to be represented by two different classes; in particular the classes `SemanticLogicTuple` and `SemanticLogicTemplate` are exploited, which are both of kind `LogicTuple` (see Figure 4.4). Then, semantic tuples and templates are expressed through specific languages, as defined in Section 4.2, which are different from the languages exploited for syntactic tuples and templates. Thus, the parsing logic – reified through the method `parse` in both `SemanticLogicTuple` and `SemanticLogicTemplate` – of semantic tuples and templates has to be redefined. Figure 4.5 shows the extended parsing logic of tuples and templates. In particular, since tuples and templates are logic terms (see Section 4.1.1), two different prolog theories are exploited in order to parse semantic tuple and templates, respectively the theory `ASSERTION` and the theory `QUERY`. The complete prolog code of `ASSERTION` and `QUERY` is shown in Appendix A. Besides a prolog representation of semantic tuples and templates, the `parse` method provides an object representation of them: `IndividualDescription` in case of semantic tuples and `AbstractQueryData` in case of semantic tuple templates. Those classes are exploited in order to easily access the information about the individual described in a semantic tuple

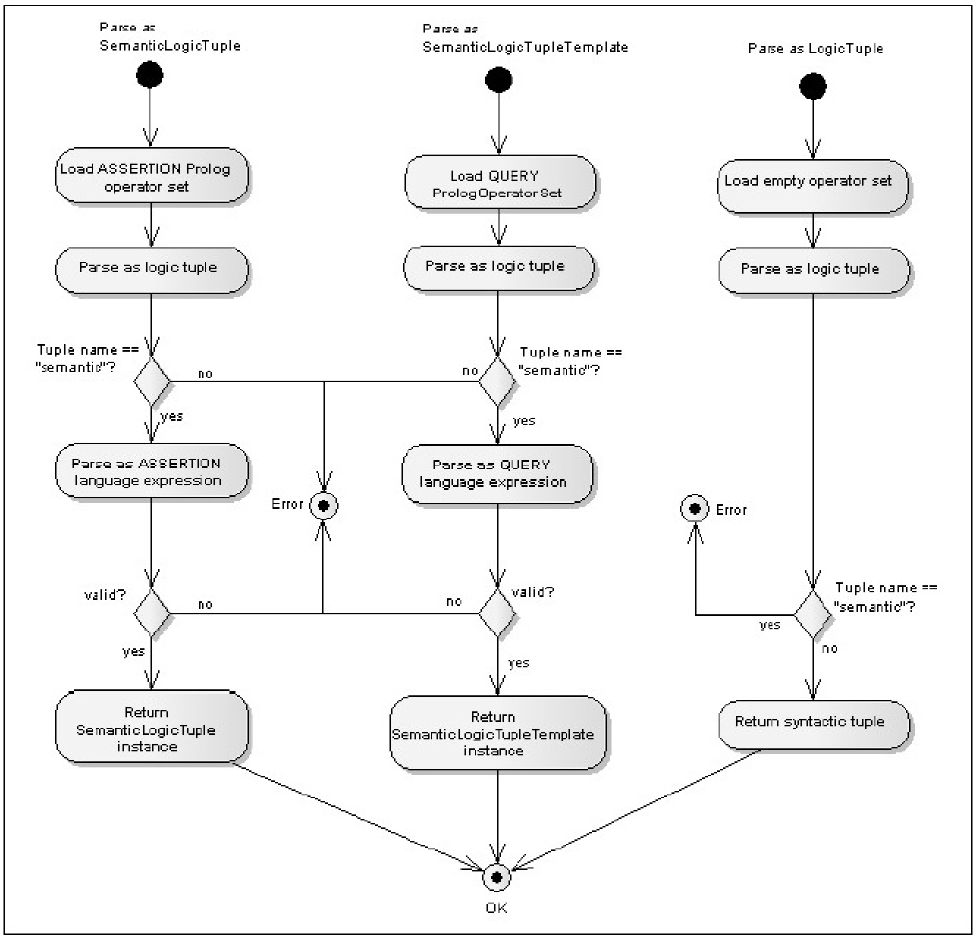

Parse as SemanticLogicTuple
Parse as SemanticLogicTupleTemplate
Parse as LogicTuple
Load ASSERTION Prolog operator set
Load QUERY PrologOperatorSet
Load empty operator set
Parse as logic tuple
Parse as logic tuple
Parse as logic tuple
Tuple name == "semantic"?
no
no
Tuple name == "semantic"?
yes
yes
Parse as ASSERTION language expression
Parse as QUERY language expression
Error
Error
valid?
no
no
valid?
Tuple name == "semantic"?
yes
yes
yes
no
Return SemanticLogicTuple instance
Return SemanticLogicTupleTemplate instance
Return syntactic tuple
OK

Figure 4.5: Parsing Logic

or about the concept described in a semantic tuple template, during the interaction with the DL reasoner. In particular, `IndividualDescription` describes an individual in terms of the individual name, the concept name and role set to which it belongs to (see Figure 4.6). Whereas `AbstractQueryData` describes a concept modelled as a hierarchy of elements of kind `Concept`, as described by the semantic template grammar shown in Section 4.2. The classes of kind `Concept` are shown in Figure 4.7. The hierarchy modelling a concept is particularly exploited in the matching mechanism shown in Section 4.2. In particular, conjunctive queries in SPARQL are exploited to perform *instance retrieval* when serving *in* and *rd* primitives, *instance checking* to trigger reactions to out primitive, and *subsumption checking* to trigger reactions to in and rd primitives. The three operations require to generate a SPARQL query starting from concept descriptions. For example, let is suppose to have the following concept description expressed in a semantic template:

```
'CityCar', (exists hasMaker : 'Maker')
```

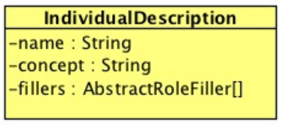


Figure 4.6: Individual Description Related to a Semantic Tuple

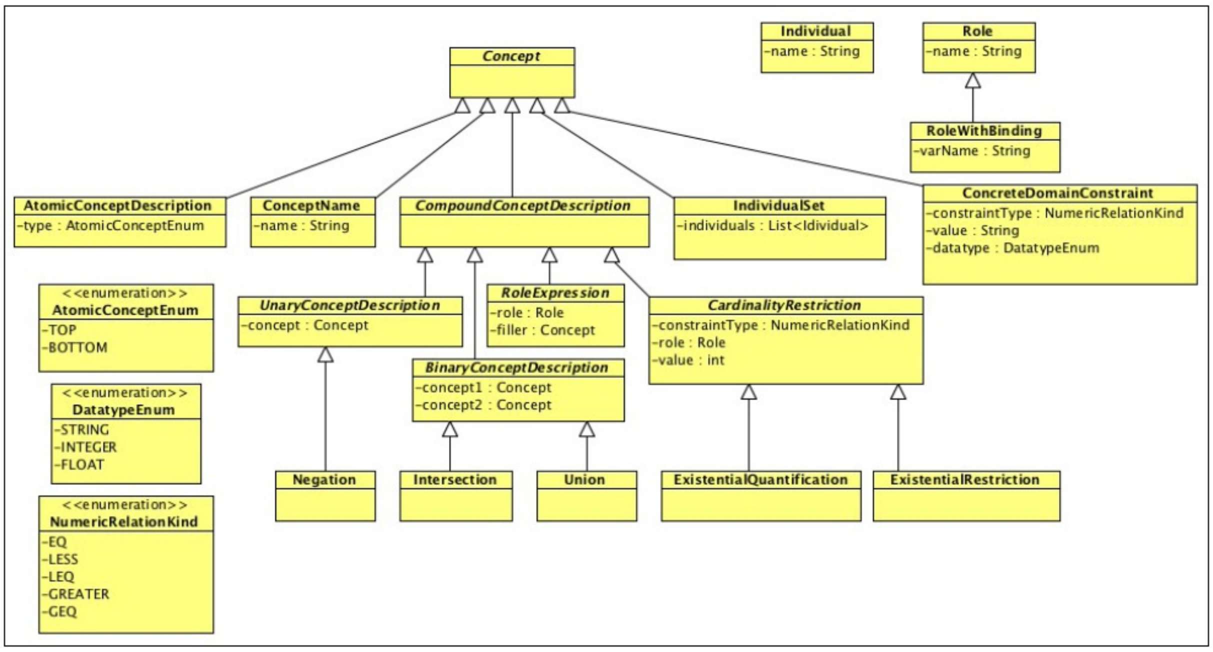


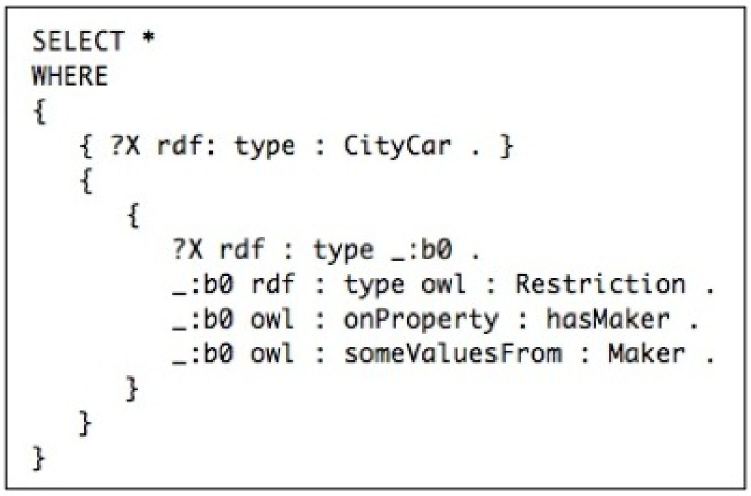
Figure 4.7: Elements Describing the Concept Related to a Semantic Tuple Template

that is equivalent in DL to:

`CityCar ⊓ (∃hasMaker.Maker).`

This concept can be expressed in SPARQL as shown in Figure 4.8: In order to build a SPARQL query starting form a concept description, the pattern *Visitor* [34] is exploited: each object of kind `Concept` (see Figure 4.7) modelling a concept description, can be *visited* through the provided method `accept` (see Figure 4.9). Then, a *visitor* of kind `StringBuildVisitor` (shown in Figure 4.10) implements the query building by *visiting* each object of kind `Concept` describing the concept.

Besides tuples and templates, also the tuple centre primitives and reactions has to be extended as shown in Section 4.2. In this implementation, primitives only retrieve the individual read or written from/to the tuple centre. The retrieving of the other variables expressed in a semantic template is partially supported in the current implementation. The variable values are retrieved through the reasoner, but they are not retrieved through the interface used from system components providing the tuple centre primitives.



```
SELECT *
WHERE
{
   { ?X rdf: type : CityCar . }
   {
      {
         ?X rdf : type _:b0 .
         _:b0 rdf : type owl : Restriction .
         _:b0 owl : onProperty : hasMaker .
         _:b0 owl : someValuesFrom : Maker .
      }
   }
}
```

Figure 4.8: SPARQL Query Equivalent to `‘CityCar’,(exists hasMaker:‘Maker’)`

Whereas, the semantic reactions are not supported in the current implementation. Those extensions are left to the future works.

4.3.2 Extending the ReSpecT Framework

Each TuCSoN node provides tuple centres by the ReSpecT *container*, which represents the life-cycle manager of the tuple centres and provides the API to access and use them. Thus, the first main extension concerns the ReSpecT container, whereby should be possible to create, retrieve and use semantic tuple centres as modelled in Section 3.3. The ReSpecT container is extended with the method `createRespectTC` allowing to create a semantic tuple centre by passing an ontology (see Figure 4.11). Then, the ReSpecT container has the task to provide the API to access the tuple centres through different interfaces called *contexts*. Each context provides a set of operations that are specific for a particular objective. For example, the context *NonBlockingContext* provides the API to perform the tuple centre primitives in a non blocking way (see Figure 4.11). In order to exploit the semantic primitives the original contexts shown in Figure 4.11 are exploited. Naturally, in case of semantic tuples and templates, the right semantic – described in Section 3.3 – is enacted. Whereas, in order to interact with the ontology related to a tuple centre, a new context called *OntologyContext* it is added and provided by the container. The current implementation of the *OntologyContext* only allows to obtain the ontology related to a particular tuple centre. However, it could be extended in order to provide further operations, for instance to modify/adapt at run-time the ontology of a tuple centre.

The ReSpecT container provides tuple centres as objects of kind `RespectTC`. Thus, the second main extension to be performed in ReSpecT is also to provide `RespectTC` ob-

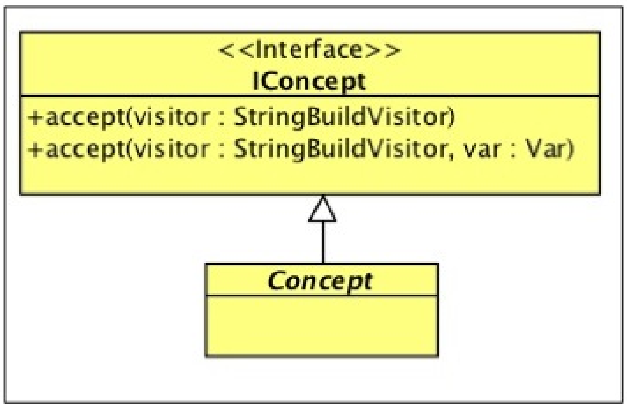

<<Interface>>
IConcept
+accept(visitor : StringBuildVisitor)
+accept(visitor : StringBuildVisitor, var : Var)
Concept

Figure 4.9: Concept as Object to be *Visited*

jects as semantic tuple centres. The extension of the class `RespectTC` is shown in Figure 4.12. As shown in the figure, the main components characterising a ReSpecT semantic tuple centre are the ontology represented by an object of kind `IOntology`, and an object of kind `SemanticKB`, modelling the semantic knowledge base stored in a tuple centre so as to be interpreted from a DL reasoner (see Figure 4.13). `SemanticKB` subscribes the interface `ISemanticKB` and implements it via the Pellet technology. In particular, the interface `ISemanticKB` defines the following operations: *(i) load ontology*, which creates the ontology composed by TBox and ABox and check if it is consistent; *(ii) assert individual*, which inserts a new individual in the ABox and checks if the ABox with the new individual is consistent; *(iii) delete individual*, which deletes an individual from the ABox; *(iv) instance checking*, which checks if an individual `a` belongs to a concept `C`; *(v) instance retrieval* – represented by the operations `readIndividual`, `readIndividualWithExpectedResult`, `removeIndividual` and `removeIndividualWithExpectedResult` –, which retrieves all the individuals belonging to a concept `C`; *(vi) subsumption checking*, which checks if a concept `C` subsumes a concept `D`. In order to implement such operations, `SemanticKB` exploits three other components: `Ontology`, `SparqlQueryGenerator` and `Reasoner`. `Ontology` models a Pellet ontology composed by a TBox – created starting from a OWL file – and an ABox, allowing the execution of *load ontology*, *assert individual* and *delete individual*. `SparqlQueryGenerator` is exploited to create a SPARQL query starting from a concept description for the execution of instance checking, instance retrieval and subsumption checking. Finally, `Reasoner` models the Pellet reasoner executing SPARQL queries, the subsumption checking between two concepts and checking the consistency of both the ABox and the whole ontology. Besides the ReSpecT container, each semantic tuple centre can also access the `SemanticKB` component for implementing semantic primitives and reactions. For instance, when an *in* request is received, the semantic tuple centre performs the instance retrieval to obtain an individual described by the semantic template. Then, the individual is deleted from the semantic KB (delete individual) and returned in

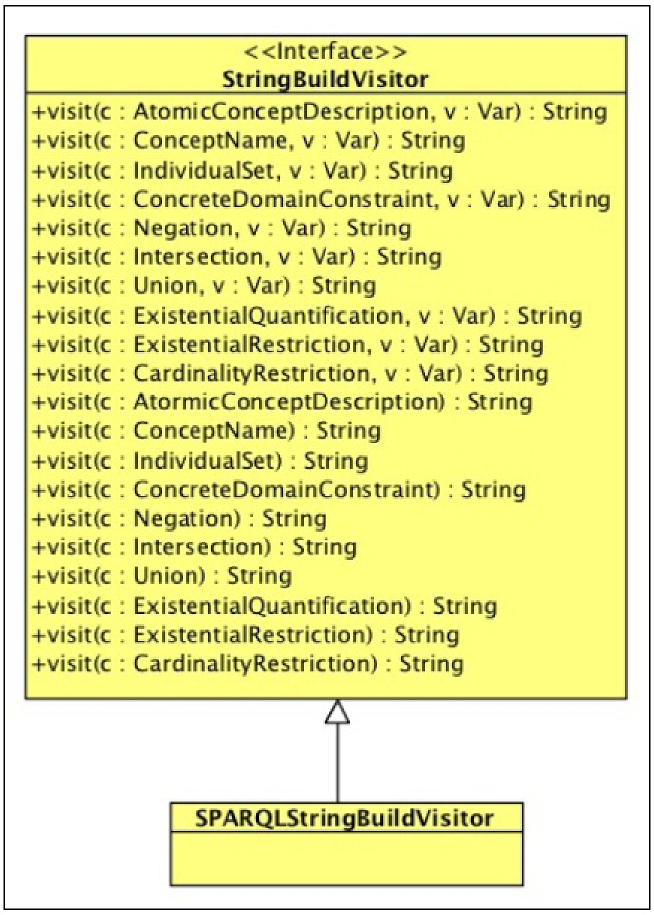
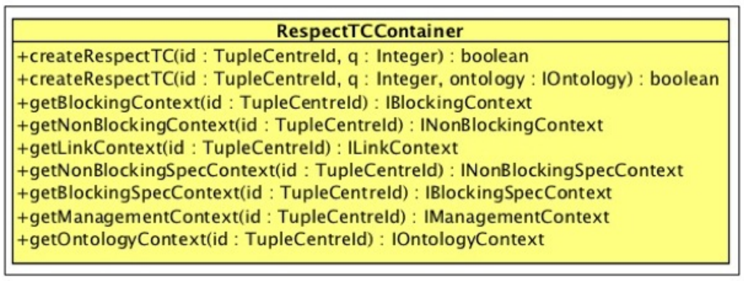

<<Interface>>
StringBuildVisitor
+visit(c : AtomicConceptDescription, v : Var) : String
+visit(c : ConceptName, v : Var) : String
+visit(c : IndividualSet, v : Var) : String
+visit(c : ConcreteDomainConstraint, v : Var) : String
+visit(c : Negation, v : Var) : String
+visit(c : Intersection, v : Var) : String
+visit(c : Union, v : Var) : String
+visit(c : ExistentialQuantification, v : Var) : String
+visit(c : ExistentialRestriction, v : Var) : String
+visit(c : CardinalityRestriction, v : Var) : String
+visit(c : AtormicConceptDescription) : String
+visit(c : ConceptName) : String
+visit(c : IndividualSet) : String
+visit(c : ConcreteDomainConstraint) : String
+visit(c : Negation) : String
+visit(c : Intersection) : String
+visit(c : Union) : String
+visit(c : ExistentialQuantification) : String
+visit(c : ExistentialRestriction) : String
+visit(c : CardinalityRestriction) : String
SPARQLStringBuildVisitor

Figure 4.10: The `StringBuildVisitor` class

response to the request, according to the semantic of *in* [35].

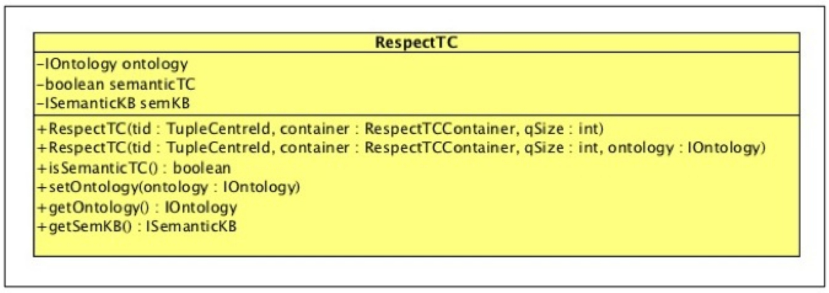
4.4 Evaluation and Discussion

In order to evaluate how much the use of semantic techniques affects the tuple centre behaviour in terms of performance, we tested our implementation of semantic tuple centres in ReSpecT with Pellet against two reference ontologies for DL reasoner benchmarks [13]: the *lumb* and *wine* ontologies. The lumb ontology covers only part of the inference supported by OWL Lite and OWL DL, while the wine ontology is more complex because it supports OWL DL with nominals – that lead to performance decay – and contains much more classes and roles. We performed three kinds of test: *(i)* the time to load an ontology in a semantic tuple centre, *(ii)* the time required to perform a semantic *out* and *(iii)* the time required to perform a semantic *in*—which is more costly than the *rd* primitive. We



```
RespectTCContainer
+createRespectTC(id : TupleCentreId, q : Integer) : boolean
+createRespectTC(id : TupleCentreId, q : Integer, ontology : IOntology) : boolean
+getBlockingContext(id : TupleCentreId) : IBlockingContext
+getNonBlockingContext(id : TupleCentreId) : INonBlockingContext
+getLinkContext(id : TupleCentreId) : ILinkContext
+getNonBlockingSpecContext(id : TupleCentreId) : INonBlockingSpecContext
+getBlockingSpecContext(id : TupleCentreId) : IBlockingSpecContext
+getManagementContext(id : TupleCentreId) : IManagementContext
+getOntologyContext(id : TupleCentreId) : IOntologyContext
```

Figure 4.11: The ReSpecT container extended



```
RespectTC
-IOntology ontology
-boolean semanticTC
-ISemanticKB semKB
+RespectTC(tid : TupleCentreId, container : RespectTCContainer, qSize : int)
+RespectTC(tid : TupleCentreId, container : RespectTCContainer, qSize : int, ontology : IOntology)
+isSemanticTC() : boolean
+setOntology(ontology : IOntology)
+getOntology() : IOntology
+getSemKB() : ISemanticKB
```

Figure 4.12: The `RespectTC` class

omitted the test about the time required to interact with the semantic module when a reaction is evaluated, because this is a simpler case of *(iii)*. Thus, the provided tests cover all the points in which the ReSpecT tuple centre was extended in order to provide the semantic support. The test was executed on an Mac Pro, with two processors Dual-Core Intel Xeon with speed 2.66 GHz and 2 GB of RAM.

Figure 4.14 shows the results of the first test in ms, where the load time is composed by: the time to read the ontology from an OWL file, the time to prepare – initial process and caching – the ontology, and the time to check the ontology consistency. Total time is slightly more than one second—which is not a problem, since this is a startup cost.

Figure 4.15 shows the results of the second test with *out* operations like:

```
out(semantic rossoColleSenesi : ‘Chianti’(locatedIn : italia))
```

As shown in Figure 4.16, the time required to insert a semantic tuple in the tuple centre is composed by *(i)* the check of the individual uniqueness, *(ii)* the creation of the individual in the ABox, *(iii)* the creation of the roles in which the individual is involved, and *(iv)* the consistency check of the ABox with the new individual. The time required to parse the tuple in order to execute *(i)*, *(ii)* and *(iii)* is not considered since it is negligible. It is worth noting that the most of the time is used in exploiting the Pellet reasoner. In fact, the only operation not involving the reasoner is the check of the individual uniqueness.

Finally, Figure 4.17 shows the results of the third test with *in* operations like:

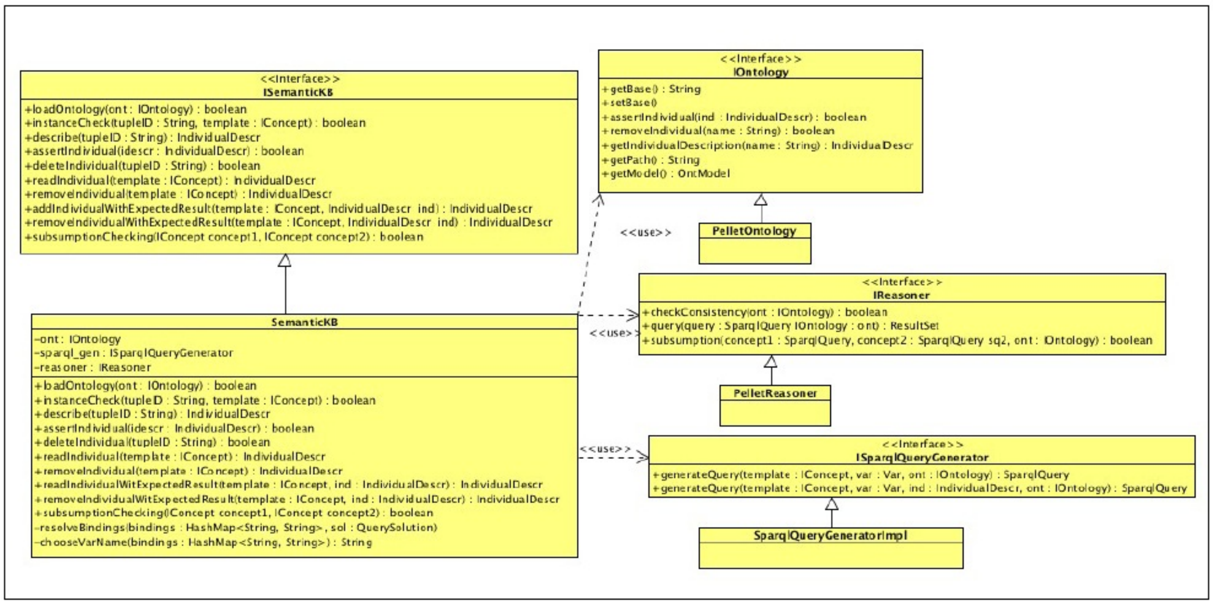

<<Interface>>
ISemanticKB
+loadOntology(ont : IOntology) : boolean
+instanceCheck(tupleID : String, template : IConcept) : boolean
+describe(tupleID : String) : IndividualDescr
+assertIndividual(idescr : IndividualDescr) : boolean
+deleteIndividual(tupleID : String) : boolean
+readIndividual(template : IConcept) : IndividualDescr
+removeIndividual(template : IConcept) : IndividualDescr
+addIndividualWithExpectedResult(template : IConcept, IndividualDescr ind) : IndividualDescr
+removeIndividualWithExpectedResult(template : IConcept, IndividualDescr ind) : IndividualDescr
+subsumptionChecking(IConcept concept1, IConcept concept2) : boolean
SemanticKB
-ont : IOntology
-sparql_gen : ISparqlQueryGenerator
-reasoner : IReasoner
+loadOntology(ont : IOntology) : boolean
+instanceCheck(tupleID : String, template : IConcept) : boolean
+describe(tupleID : String) : IndividualDescr
+assertIndividual(idescr : IndividualDescr) : boolean
+deleteIndividual(tupleID : String) : boolean
+readIndividual(template : IConcept) : IndividualDescr
+removeIndividual(template : IConcept) : IndividualDescr
+readIndividualWitExpectedResult(template : IConcept, ind : IndividualDescr) : IndividualDescr
+removeIndividualWitExpectedResult(template : IConcept, ind : IndividualDescr) : IndividualDescr
+subsumptionChecking(IConcept concept1, IConcept concept2) : boolean
-resolveBindings(bindings : HashMap<String, String>, sol : QuerySolution)
-chooseVarName(bindings : HashMap<String, String>) : String
<<Interface>>
IOntology
+getBase() : String
+setBase()
+assertIndividual(ind : IndividualDescr) : boolean
+removeIndividual(name : String) : boolean
+getIndividualDescription(name : String) : IndividualDescr
+getPath() : String
+getModel() : OntModel
<<use>>
PelletOntology
<<Interface>>
IReasoner
+checkConsistency(ont : IOntology) : boolean
+query(query : SparqlQuery IOntology : ont) : ResultSet
+subsumption(concept1 : SparqlQuery, concept2 : SparqlQuery sq2, ont : IOntology) : boolean
<<use>>
PelletReasoner
<<use>>
<<Interface>>
ISparqlQueryGenerator
+generateQuery(template : IConcept, var : Var, ont : IOntology) : SparqlQuery
+generateQuery(template : IConcept, var : Var, ind : IndividualDescr, ont : IOntology) : SparqlQuery
SparqlQueryGeneratorImpl

Figure 4.13: The **SemanticKB** class and its Components

Ontology	*Read Ontology*	*Prepare*	*Consistency Check*
lumb	1221	2	94
wine	1345	1	103

Figure 4.14: Ontology load time in ms

```
in(semantic Result matching 'Wine' and exists locatedIn : italia)
```

As shown in Figure 4.18, the time required to insert a semantic tuple in the tuple centre is composed by: *(i)* the SPARQL query execution, *(ii)* some ReSpecT computation in order to manage the interaction with the Pellet reasoner, *(iii)* the interaction with the Pellet reasoner in order to obtain the complete description of the individual retrieved through the query. The time required to parse the template in order to build a SPARQL query is not considered since it is negligible. Again, it is worth noting that the most of the time is used in exploiting the Pellet reasoner, in particular in order to execute *(iii)*.

By analysing the test result, we can conclude that the scalability of a ReSpecT semantic tuple centre is quite good in case of the *out* primitive, even if it is not optimal. The results of the test with the *in* primitive are not so good, since a query of average complexity takes more than two seconds with only 150 individuals. However, it should be noted that the performance of the semantic extension of the tuple centre in a negligible way depends on ReSpecT, but it mainly depends on the Pellet reasoner: so, any improvement in the DL reasoner performance would largely improve the performance of our implementation.

N individuals	*Lumb*	*Wine*
10	11	29
20	12	24
30	9	20
40	12	15
50	12	12
100	10	24
150	9	21

Figure 4.15: Semantic *out* in ms

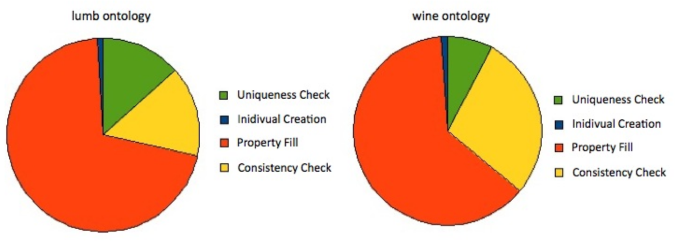


Figure 4.16: Semantic *out* composition

4.5 Summary

In this chapter we discussed the model and implementation of semantic tuple centres—a basic brick of coordination infrastructures for open, distributed and knowledge-intensive systems like Web-based and pervasive computing applications. In particular, this model supports the novel notion of programming the semantic coordination aspects of a complex system, a key notion in the coordination of self-organising systems [69]. The model is currently implemented as an extension of the TuCSoN coordination infrastructure [?], and is based on the well-known Pellet semantic reasoner [82]. Our analysis of performance showed that, while tuple insertion is typically quite fast, tuple retrieval can be an heavy operation especially when many tuples occur in the tuple space, which is basically due to the latency of instance checking by the semantic reasoner.

N individuals	*Lumb*	*Wine*
10	111	312
20	123	328
30	113	231
40	119	309
50	138	346
100	179	1058
150	192	2150

Figure 4.17: Semantic *in* in ms

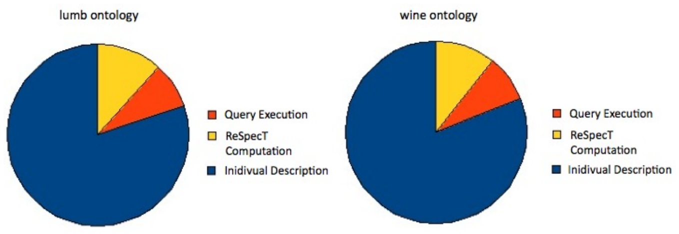


Figure 4.18: Semantic *in* composition

Chapter 5

Coordination in e-Health Systems

This chapter aims at giving an overview of the e-Health – that is, Healthcare supported by software systems – since it provides several application scenarios that are characterised by the requirements *distribution* and *openness*. Among the several e-Health research activities, research on Electronic Health Record (EHR) is particularly intensive. The main challenge in the EHR domain is to ensure interoperability among EHR fragments – medical information that are stored in a digital format over different healthcare institutions – belonging to an environment that is *distributed* and *open* and where the *security support* represents a fundamental requirement to protect the patient privacy. Several efforts have been made in the EHR domain in order to cope with such requirements, but the approaches provided in literature seems to be not enough powerful to fully rich the EHR domain challenge. Along this line, in this chapter it will be shown how it is possible to extend the solutions proposed in literature by tacking the semantic version of TuCSoN as inspiration model, in order to augment their effectiveness in building EHR services, in particular as far as interoperability is concerned. Thus, in this chapter first it will be provided a brief survey of the existing approaches supporting interoperability among EHR fragments, of their benefits and drawbacks. Then, it will be shown how to exploit the key-features of the TuCSoN architecture supporting semantic tuple centres in order to extend the solutions proposed in literature. The chapter will conclude providing the future research directions of this work.

5.1 EHR Systems Interoperability

5.1.1 Towards Electronic Health Records

The healthcare domain has been evolving quickly over the past decades. Nevertheless, advances are somewhat limited in several domain [6] and obstacles are still to be overcome [24, 78]. Most of the administrative processes have adopted an e-Health solution so as to become computerised. However, in some hospitals and for the large majority of

General Practitioners (GPs), medical data is still acquired and exchanged on paper. The totally paperless hospital has yet a way to go [80]. Besides digital storage, the computerised acquisition of medical data also makes data accessible for computerised decision support [89] and has the potential to reduce the large number of adverse events, particularly in hospitals [19, 49]. Alongside local advantages, communication of health data is another important factor in computerised data acquisition to overcome limits of paper-based information exchange, which is often slow [88] and error prone [49]. e-Health information exchange strategies and solutions have been developed on a local regional or cross-institutional [52] and national [71, 28] level and some already have concrete implementation in research projects [88].

Among the several e-Health research activities concerning the health information exchange, research on *Electronic Health Record* (EHR) is particularly intensive [49, 90]. A Patient EHR-document refers to the medical record of a patient stored in a digital format. The information stored in an EHR might include patient information such as demographics, medical history, medication, allergy list, lab results or radiology. Medical data belonging to an EHR are called *fragments*, and can be distributed over different EHR systems. The introduction of EHR offers several benefits [54]:

Better patient safety — Storing and transferring patient information electronically allows reducing clinical errors caused for example by illegible handwriting, documents or images, as well as it allows clinicians to communicate more quickly and accurately and to identify relevant information more easily.

Lower cost of health services — EHR technology can reduce administrative work to manage medical data since it can increase medical-data search efficiency and reduce medical-data duplication and waste.

Better audit and research — Behind improving medical assistance of patients, EHR technologies are also useful for other purposes. In particular, electronic databases of health information can be exploited for healthcare audit and research.

In order to keep the EHR benefits, EHR systems should ensure interoperability among EHR fragments. *Interoperability* – that is, the ability for two or more heterogeneous systems to communicate together – is of paramount importance in health information communication [26]. In case of EHR systems, interoperability should satify the following conditions [48]:

Distribution — EHR fragments should be easy to share even if the information is widespread across multiple EHR systems.

Openness — EHR supporting servers at different caregivers could be heterogeneous and change dynamically.

Security — It is necessary to support security mechanisms in order to avoid failures that can cause injury to the patient and violations to privacy.

Accordingly, interoperability among EHR systems call for specialised middleware able to deal with distribution, openness and security requirements in a coherent and transparent way. In the next section, we discuss standards and solutions from the literature, which propose design principles for middleware of such a sort.

5.1.2 Existing Approachers: a Survey

In order to cope with distribution, openness and security, the first approach to the issue of interoperability is the definition of standards for EHR-fragment format and communication. The two most representative are [48]:

- *Health Level Seven* (HL7): a set of open standards for the exchange, management and integration of EHR fragments [25]. In particular, HL7 provides *Clinical Document Architecture* (CDA) – a standard for the representation and machine processing of clinical documents – and *Messaging* standard—a standard covering EHR-fragment messaging aspects.

- *Digital Imaging and Communications in Medicine* (DICOM): a standard for handling and transmitting information in medical imaging. It includes a file format definition and a network communication protocol [44].

Standards such as HL7 and DICOM are not enough to achieve interoperable health systems. In fact, the result is that EHR systems use different set of format and communication standards, often incompatible, incomplete or involving overlapping scopes, thus breaking the interoperability requirement [40]. As a response to these problems – and as a complementary step towards the requirements of interoperability among EHR fragments – the following standards and initiatives were proposed:

- *openEHR* [27] and *CEN EN 13606* [32]: standards aiming at facing interoperability among EHR fragments. In particular they propose semantic approaches based on *Archetype Definition Language* (ADL) [8] – a formal language for expressing application-domain concepts – in order to describe semantically EHR fragments. By exploiting such kind of semantic techniques it may be possible to support interoperability among EHR fragments with different syntactic structures depending on the adopted standard.

- *Integrating the Healthcare Enterprise* (IHE) [43]: a non-profit initiative founded in 1998 led by professionals of the e-Health industry. The initiative goal is not to develop standards as such, but to select and recommend an appropriate usage of existing standards (e.g. HL7 and DICOM) in order to improve the sharing of information among EHR systems.

In this context, openEHR, CEN EN 13606, and IHE emphasise two important requirements. The first is to describe semantically EHR fragments in order to face heterogeneity and dynamism of fragment formats. The second is to provide a coordination middleware able to coordinate EHR systems and actors interacting with such systems, hiding distributed-fragment management and security issues from entities to be coordinated. In particular, XDS, ATNA and XUA are central profiles for building a coordination middleware that connects EHR systems.

A further important contribute in this context is represented by *Triple Space Computing* (TSC) [16], which provides a different solution based on the *Linda tuple space* model [36]. Through the tuple space model, coordination among system entities occurs by exchanging information in form of tuples, in a common shared space called tuple space. System entities to be coordinated communicate with one another through *out*, *rd* and *in* primitives to respectively put, read and consume associatively tuples to/from the shared space. In particular, TSC shows the following interesting features:

- It provides a general coordination model to manage all kinds of interactions among system entities.
- Tuple space model is based on *generative communication* [36]: tuples generated by a tuple producer have an independent existence in the tuple space leading to *time*, *space* and *name uncoupling.* Uncoupling is a requirement to satisfy in order to cope with openness. Entities belonging to an open environment can be heterogeneous and can be added, removed or modified at runtime. For this reason, an entity cannot make a-priori assumptions about other system entities.
- It is based on *semantic tuple-space computing* [64]: it adopts tuple spaces enriched semantically thus allowing an exchange of data (tuples) semantically described by means of an ontology.
- Like IHE, it provides a Web-service interface to tuple spaces which promotes interoperability.

TSC puts together the advantages of using openEHR, CEN EN 13606 and IHE. Moreover, it improves the IHE approach by proposing a more general coordination model suitable for open scenarios, and not only specialised on the storage and retrieval of EHR fragments. In particular, through TSC it is possible to exploit a unique coordination model – the tuple space model – to manage all the system interactions. This is useful in case it is needed to extend e-Health systems with coordination functionalities concerning different kinds of interactions. For example, interactions with patients, with scientific research systems or with systems providing consumers with access to medical developments and research.

However, TSC exhibits some limits, too. The first one derives from the Linda tuple-space model: the tuple space behaviour is set once and for all by the model and cannot be tailored to the specific application needs [67]. Thus, any coordination law not directly supported by the model has typically to be charged upon coordinated components, thus obstructing the way to open systems. A further feature that should be supported by an EHR coordination-middleware – and that it is not covered by either IHE or TSC – is the ability to change its configuration at runtime in order to cope with application dynamism. In fact, during the lifetime of an application, requirements could be changed, added or removed. For example, new nodes could be added/removed to/from the network, or, coordination algorithms could be changed in order to improve the efficiency and effectiveness of the overall application. If the middleware does not allow for runtime changes, it might be necessary to shut it down in order to update its configuration—which is definitely undesirable, especially in application scenarios like e-Health that require continuous service availability.

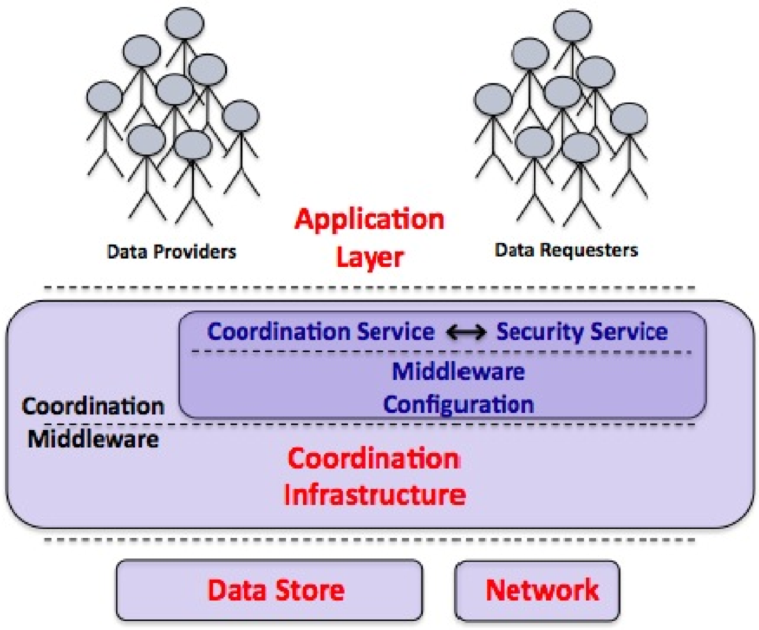


Figure 5.1: Logical Levels for a Coordination Middleware

As far as middleware for the health domain is concerned, the current state of the art can be summarised as follows:

- IHE profiles, in particular XDS profile, do not provide a middleware model expressive enough to manage interactions among EHR actors. In particular, XDS provides a coordination middleware model not based on semantic techniques, and focused on coordinating meta-data in order to store and retrieve EHR fragments. As a consequence, it cannot be used for e-health applications going beyond the mere fragment coordination.

- TSC provides a solution that overcomes part of the XDS profile drawbacks. In particular, it exploits the Linda tuple-space model enriched with semantic techniques that exhibits features particularly useful for the realisation of an EHR middleware. On the other hand, TSC has some limits due to the fixed behaviour of its coordination abstractions, and to its inability to cope with middleware evolution over time.

Accordingly, an EHR middleware should be a coordination middleware supporting interactions among heterogeneous EHR-fragment providers and requesters—as in Figure 5.1. The middleware should be developed upon a coordination infrastructure giving the support for *distribution* of heterogeneous EHR-fragment-stores and providers/requesters of EHR fragments in a transparent way. Then, the infrastructure should provide the API required to build a *coordination* and a *security service*. The coordination service has the task to enable and rule interactions among system actors. It should exploit a general-purpose coordination model based on the tuple-space model and on semantic techniques, in order to cope with the openness requirement. In turn, the security service should be able to guarantee privacy of patient EHR-fragments, taking into account the federated nature of the healthcare system. Finally, the coordination infrastructure should be based on *engineering approaches* making it possible to build a coordination middleware whose configuration is adaptable at runtime, in order to maintain a continuously-available EHR-service in front of dynamic changes of the application requirements.

5.2 Exploiting Semantic TuCSoN in e-Health

In the following, we show how the TuCSoN approach can be adopted in order to extend the solutions proposed by TSC and IHE (see Section 5.1.2): the overall goal is to increase the effectiveness of TSC and IHE approaches in coordinating EHR fragments. In particular, we discuss how such approaches can be integrated and extended with the key features of TuCSoN architecture, presented in Section 4.1. When dealing with IHE, we refer in particular to the following recommendations [43]: *Cross-Enterprise Document Sharing* (XDS) – i.e. profile describing an infrastructure for storing and registering medical documents –, *Audit Trial and Node Authentication* (ATNA) – i.e. profile describing security procedures – and *Cross-Enterprise User Assertion Profile* (XUA)—i.e. profile describing means to communicate claims about the identity of an authenticated principal (user, application, system,...) in operations that cross healthcare-enterprise boundaries.

TuCSoN topology and XDS Affinity Domains The e-Health environment is federated, that is, each healthcare enterprise belongs to a domain with other healthcare enterprises, using a common set of policies ruling interactions with and within a domain, and sharing common clinical documents. XDS calls each domain *Affinity Domain*. According to Section 4.1, the hierarchical topology of TuCSoN fits well with the sort of topology required by the EHR scenario. In particular, an Affinity Domain could be mapped in a TuCSoN domain whose gateway maintains the information about the policies and the structures associated to the domain itself. Then, each healthcare enterprise belongs to an Affinity Domain can be mapped in a TuCSoN place.

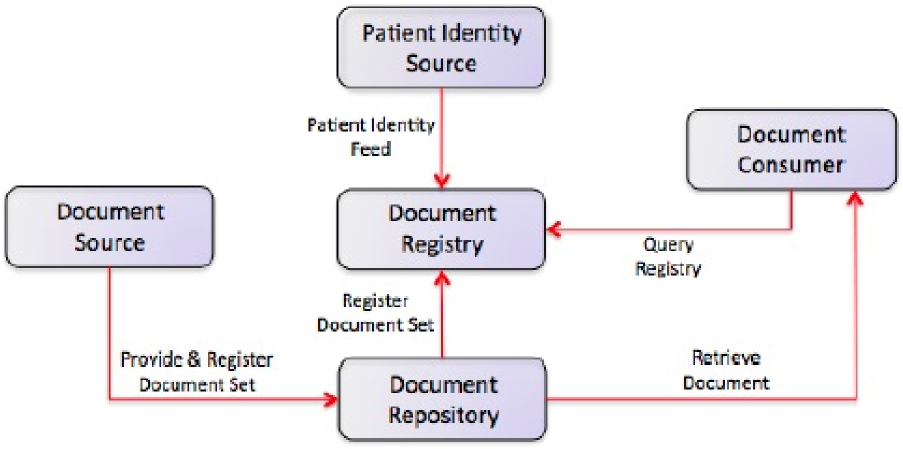


Figure 5.2: Actors for the IHE XDS Profile

TuCSoN semantic tuple centres as fragment coordination media XDS provides a model to store and retrieve EHR fragments. Figure 5.2 shows the actor model defined by XDS. In particular the model is composed by:

Document Source — A healthcare point of service where clinical data is collected.

Document Consumer — A service application where care is given and information is requested.

Document Registry — A system storing *ebXML descriptions* of the clinical fragments to rapidly find them back.

Document Repository — A system that stores documents and forwards the metadata to the document registry.

Patient Identity Source — A system that manage patients and identifiers for an Affinity Domain.

The XDS actor model has two main drawbacks. Document Registry is exploited to store and search metadata describing EHR fragments whereby it is possible to retrieve the related document from the Document Repository. In particular, XDS suggests to realise the registry through the *ebXML Registry* standards.

However, the main limit of an ebXML Registry is that it describes metadata in XML, and retrieves metadata in face of a query written in XML and SQL format. This kind of knowledge representation and retrieval lacks the expressive power provided by semantic approaches exploiting ontologies. In fact, unlike an ontology, an XML schema does not allow the description of complex taxonomies among concepts like those exploiting subsumption relationships. Also, XML tools does not perform powerful reasoning over metadata like semantic reasoning, which is able instead to infer new knowledge that is not declared in an explicit way. Thus, ontology-based approaches are more suitable for engineering knowledge in open context where the knowledge structure can evolve and where software components only have a partial awareness about the overall knowledge.

Another limit of the ebXML Registry is that it promotes a pre-defined behaviour only able to store and retrieve metadata. As a consequence, in order to extend the behaviour of the registry, a layer should be upon it that would enrich the operational semantic behind its interface in order to implement the new desired behaviour. This, of course, would definitely augment the complexity of the system. In order to cope with complexity, instead, it would be desirable to be able to define new behaviours directly in the registry, customising the registry with the policies associated to a particular healthcare domain. For example, a policy would allow the registry to be distributed over different nodes belonging to the domain, instead of having an unique registry per domain, as suggested by XDS. By exploiting behaviour programmability of the coordination media, it would be possible to coordinate a set of domain registries collaborating with one another in order to search and distribute metadata within the domain, in a smart way.

This is why the tuple centre model looks like a good candidate to build a Document Registry. On one hand, a semantic tuple centre supports the semantic representation of the stored knowledge – like TSC –, but – unlike TSC – it also provides a tuple/template language that is independent from the technology exploited to implement the semantic support. Thus, each domain can choose to exploit a particular semantic technology guaranteeing interoperability with other domains. On the other hand, since the behaviour of a tuple centre is programmable, it is possible to tailor the registry to specific application needs. Moreover, by exploiting logic tuple centres like ReSpecT tuple centres, it is possible to promote cognitive processes by exploiting rational agents.

Exploiting TuCSoN RBAC model As shown in Section 4.1, TuCSoN provides the organisation abstraction to describe the structures and rules composing a system. In particular, an organisation in an Affinity Domain could be mapped in the TuCSoN `$ORG` tuple centre managing the domain structures, like Document Registries and the domain places where e-Health enterprises are hosted, and defining the set of roles that can interact with the organisation along with a set of related policies to rule such interactions.

In the context of Affinity Domains, a role represents a class of identities that can interact with EHR fragments, whereas policies represent the admissible interactions for a specific role. Accordingly, the RBAC-MAS model [93] can be suitable integrated with security recommendations defined in ATNA and XUA. Such recommendations in particular require: *(i)* an authentication service able to authenticate users, *(ii)* access control policies, *(iii)* a secure communication between system actors, and *(iv)* a security service supporting cross-authentication among EHR domains. In order to satisfy such requirements, TuCSoN can be integrated with two technologies suggested by IHE: *Kerberos* authentication service, and Web Services as interface to access to TuCSoN organisations, so as to promote the interoperability requirement.

Thus, Web Services can be used to access to TuCSoN organisation in secure way by exploiting *WS-Security*, that is, a secure communication protocol developed by the *OASIS-Open group*. In particular, *WS-Security* includes both *WS-SecureConversation* – which can be exploited to ensure secure conversations among system actors –, and *WS-Trust*—which can be exploited to support cross-authentication among EHR domains. Through *WS-Trust* it is possible to establish trust relations among domains that are exploitable to accept requests coming from different domains without having to authenticate users again. Finally, by integrating the authentication service Kerberos with TuCSoN, user identities can be associated to roles and policies in the `$ORG` tuple centre, and be authenticated.

Online engineering for continuos e-Health system interoperability As discussed in Section 4.1, TuCSoN exploits *alive* abstractions to model coordination, organisation and security, thus promoting their online engineering. By exploiting semantic tuple centres to model Document Registries and organisation to model the structures composing an Affinity Domain, TuCSoN makes it possible to support the runtime corrective/adaptive/evolutive maintenance of an e-Health fragment system—that is, with no need to stop the system. This is particularly useful whenever application requirements are expected to change substantially over time. For instance, it may happen that new places hosting e-Health enterprises need to be added by reconfiguring Affinity Domains dynamically, or that roles and policies have to be added/removed/modified to cope with dynamic organisation changes. This would require to change the behaviour of a Document Registry to face the new application requirements. Through online engineering as supported by the TuCSoN architecture, the system could be evolved in a consistent way at runtime, maintaining a continuos interoperability among EHR systems. We think this is a crucial aspect to be considered in the engineering of e-Health applications, where a continuos service availability is indeed fundamental—and this is why we promote the integration of IHE recommendations within the TuCSoN architecture.

5.3 Summary

In this chapter we shown an e-Health application scenario represented the EHR fragment coordination. The application scenario is interesting because it concerns a *distributed* and *open* environment in which: *(i)* EHR fragments could be distributed among different healthcare systems, *(ii)* healthcare systems can be heterogeneous and change dynamically, and *(iii)* security mechanisms play a fundamental role to ensure patient privacy and safety. The chapter shown the main shortcomings of the several efforts provided by the literature trying to cope with such requirements—like HL7, DICOM, CEN EN 13606, openEHR, IHE and TSC. First of all, they provide special-purpose models of coordination, which increase the complexity of building an EHR coordination middleware. This limits system interoperability by making it difficult to integrate independent e-Health systems. Moreover, they do not support any form of online engineering. As a consequence, the coordination middleware cannot be updated at runtime in order to cope with new application requirements without stopping the system. Along this line, in this chapter it is proposed a coordination model and technology that could integrate the solutions and standards proposed in literature while addressing the aforementioned issues. In particular, it is proposed semantic TuCSoN as a reference architecture for coordination in the e-Health scenario since it provides a general-purpose model of coordination accounting for distribution and security issues in the engineering of EHR systems, and promotes online engineering for continuos service availability of e-Health applications.

Chapter 6

Fuzziness in Semantic Tuple Centres

[... under vagueness/fuzziness theory fall all those approaches in which statements (for example, "the tomato is ripe") are true to some degree, which is taken from a truth space. That is, an interpretation maps a statement to a truth degree, since we are unable to establish whether a statement is completely true or false due to the involvement of vague concepts, such as "ripe", which only have an imprecise definition. For example, we cannot exactly say whether a tomato is ripe or not, but rather can only say that the tomato is ripe to some degree... Vague statements are truth-functional, i.e., the degree of truth of a statement can be calculated from the degrees of truth of its constituents...] [53].

Starting from the semantic tuple centre model shown in Chapter 4, this chapter aims at describing an extension of the model towards fuzziness, in order to support coordination based on information exchange, which can be vague. Thus, after having discussed fuzzy Description Logics, in particular fuzzy SHOIN(D), we list and describe the elements required to extend the semantic tuple centre model in order to support fuzziness, and detail a possible line of extension in semantic TuCSoN. Then, we provide some examples showing how fuzzy semantic tuple centres could be fruitfully exploited. Finally, it will be outlined the main research directions in order to improve the current implementation of the fuzzy semantic tuple centres.

6.1 Towards Fuzziness

In the last decades, research on Artificial Intelligence has paid a lot of attention on the representation of vague/fuzzy knowledge, in order to extend existing knowledge representation systems that are not suitable to cope with the imperfect nature of real world information [99, 51, 83]. In particular, as stated by Zadeh in [99], *[... More often than not, the classes of objects encountered in the real physical world do not have a precisely defined criteria of membership ...]* and *[... imprecisely defined "classes" – for example, the "class of beautiful women" or the "class of tall men" – play an important role in human* *thinking, particularly in the domains of pattern recognition, communication of information, and abstraction ...].* As Description Logics [2] are limited to deal with crisp and well defined concepts, individuals and queries, starting from the first work done by Yen in [98] they were subject of several extensions toward fuzziness [53, 84, 10]. In particular, drawing from the most incisive works in literature [53, 10, 84] it is possible here to show a fuzzy generalisation of SHOIN(D). From the syntactic point of view, it is possible to describe *fuzzy SHOIN(D)* in terms of *fuzzy datatype theories*, *fuzzy modifiers* and then *fuzzy knowledge bases* and *fuzzy axioms.*

Fuzzy datatypes. Fuzzy SHOIN(D) makes it possible to reason with datatypes, such as strings and integers, using the so-called *concrete domains* [2]. As showed by Straccia in [84], fuzzy concrete domains and thus datatypes are based on fuzzy sets. For example, the datatype predicate $\leq_{18}$ is a unary crisp predicate over the natural numbers denoting the set of integers smaller or equal to `18`. Functions for specifying fuzzy-set-membership degrees, are used. For example we can define the datatype predicate `High` as `High(x) = rs(x; 80,250)`, where `rs` is the *right-shoulder function* that is shown in Figure 6.1.

Fuzzy modifiers. Fuzzy SHOIN(D) also supports fuzzy modifiers like `very`, `more_or_less`, and `slightly`, applied to fuzzy sets so as to change their membership function. As shown in [84], a fuzzy modifier `m` represents a function f_m`:[0,1]` $\rightarrow$ `[0,1]`. For example, we may define $f_{very}(x) = x^2$ and the datatype predicate `High` and use `very(High)` to define the semantic of the fuzzy concept *very high.*

Fuzzy knowledge bases and fuzzy axioms. In the following we let variable α range in interval `[0,1]`, denoting so-called *degrees.* As defined in [53], a fuzzy knowledge base `K=(R;T;A)` consists of a *fuzzy RBox* `R`, a *fuzzy TBox* `T` and a *fuzzy ABox* `A`. Omitting the fuzzy RBox that is not of interest for this work, a fuzzy TBox `T` is a finite set of fuzzy concept equality and inclusion axioms. Concerning the formalism for the concept definition in the TBox [2], besides fuzzy datatypes and modifiers, Bobillo et al. [10] introduced *fuzzy nominals* $\{\alpha_1/\mathtt{o}_1, \ldots, \alpha_m/\mathtt{o}_m\}$, where $\{\alpha_i\}$ represents the degree by which the individual $\{\mathtt{o}_i\}$ belongs to the concept represented by the nominals. For example, we can define the nominals {`1/germany, 1/austria, 0.67/switzerland`} to represent the concept of country where German is a widely-spoken language. Then, new interesting

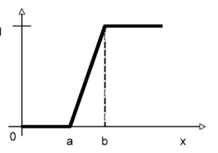


Figure 6.1: Right-shoulder function `rs(x; a,b)`

concept constructs were introduced by Bobillo and Straccia in [12]: *weighted concepts* (α C), *weighted sum concepts* ($\alpha_1 C_1$ + ... + $\alpha_N C_N$), and *threshold concepts* (C[$\geq \alpha$]) and (C[$\leq \alpha$]). A *weighted concept* represents a concept such that for any individual a, the degree of it being an instance of (α C) is given by α times the degree of being an instance of C. Then, a *weighted sum concept* represents a concept which is the weighted sum of the concepts C_i, i.e. ($\alpha_1 * C_1$ + ... + $\alpha_N * C_N$), where it is assumed α_1 + ... + α_N=1. Finally, *threshold concepts* are such that, for example, the degree by which a is instance of ([$\geq \alpha$] C) is 0 if a is an instance of C to a degree less than $\geq \alpha$, otherwise the degree is C(a). The case of ([$\leq \alpha$] C) is dual. Then, a fuzzy TBox is a finite set of fuzzy concept inclusion axioms $\tau \geq \alpha$, $\tau \leq \alpha$, $\tau > \alpha$, and $\tau < \alpha$, where τ is a concept inclusion axiom in SHOIN(D). For example, SportCar $\sqsubseteq$ Car $\geq$ 1 is a fuzzy axiom stating SportCar is a Car with degree ≥ 1—namely, with degree $= 1$

A fuzzy ABox A consists of a finite set of equality and inequality axioms a = b and a $\neq$ b, respectively, and of fuzzy concept and fuzzy role membership axioms of the form $\tau \geq \alpha$, $\tau \leq \alpha$, $\tau > \alpha$ and $\tau < \alpha$, where τ is a concept or role membership axiom in SHOIN(D). For example, SportsCar(audi_tt) $\geq$ 0.92 is a fuzzy axiom stating the individual audi_tt is a SportCar with degree ≥ 0.92.

From a semantic point of view, concepts and roles are interpreted as fuzzy subsets of an interpretation domain. Therefore, axioms in fuzzy SHOIN(D), rather than being satisfied or unsatisfied in an interpretation, are associated with a degree of truth in [0,1]. Accordingly, *fuzzy interpretation functions* $.^I$ are introduced in [53, 10, 84], associating a degree in [0,1] to each fuzzy concept construct. For instance, the concept constructs $\sqcap$ and $\sqcup$ have the following fuzzy interpretation functions in [53]:

$$(C_1 \sqcap C_2)^I(x) = {C_1}^I(x) \otimes {C_2}^I(x)$$
$$(C_1 \sqcup C_2)^I(x) = {C_1}^I(x) \oplus {C_2}^I(x)$$

where the operators $\otimes$ and $\oplus$ are interpreted according to one of the following logics: *Lukasiewicz Logic*, *Gödel Logic*, *Product Logic* and *Zadeh Logic*. For example, *Gödel Logic* defines $\alpha \oplus_G \beta$ such as max{α,β} whereas *Lukasiewicz Logic* defines $\alpha \oplus_L \beta$ such as min{α+β,1} — see [53] for more details.

6.2 Fuzzyfying Semantic Tuple Centres

In the previous section it was observed that real-world information are often imprecise and vague. For example, in applications involving sensors, reading measurements usually comes with degrees of evidence; in applications like multimedia processing, object recognition might come with degrees of truth [53]. Moreover, in open contexts it may not have the knowledge required to formulate precise queries [4]. For instance, a user may request to find a cinema that is close to her, without bothering about a real distance range. Indeed, people commonly describe an object property using words like *close*, *far* or *cheap*.

Although several works discuss how to represent vague/fuzzy knowledge (see Section 6.1), we found only the work of Balzarotti et al. [4] exploiting fuzziness to describe knowledge in tuple spaces—there, a tuple space framework called LighTS is presented, where fuzzy templates can be built in terms of fuzzy fields. For example, the template `A(Temperature is Hot, Distance is Far)` can be defined in order to obtain a tuple `A` in which the field `Temperature` and `Distance` are considered respectively *hot* and *far* by following a defined membership function associated to the fuzzy concepts `Hot` and `Far`. However, LighTS does not support semantic matching, that how it is shown in Chapter 2, it is required in the engineering of distributed and open systems. Thus, in this work we aim at extending the tuple centre model in order to support *fuzzy* and *semantic* knowledge description with the objective to support coordination in open contexts where the knowledge about the application domain is often not complete and precise. In particular, following our definition of semantic tuple centres described in Chapter 3, the ingredients required to extend semantic tuple centres toward fuzziness are: *fuzzy ontology*, *fuzzy tuples*, *fuzzy templates*, *fuzzy primitives*, *fuzzy reactions* and *fuzzy semantic matching.*

Fuzzy Ontology. The domain ontology associated to a semantic tuple centre is formally described through SHOIN(D) (see Section 3.2). According to Section 6.1, we can formally describe a fuzzy domain ontology associated to a tuple centre through fuzzy SHOIN(D) in the form of a fuzzy TBox. Unlike crisp SHOIN(D), fuzzy SHOIN(D) allows us to use *fuzzy datatypes*, *fuzzy modifiers*, *fuzzy nominals*, *weighted concepts*, *weighted sum concepts* and *treshold concepts* for the concept definition, and *fuzzy concept inclusions* for the taxonomy definition. Hence, we can for instance define the following assertions:

(1) $\texttt{High} \equiv \texttt{rs(80,250)}$
(2) $\texttt{SportsCar} \equiv \texttt{Car} \sqcap \exists\texttt{hasSpeed.very(High)}$

in order to describe *(i)* the fuzzy datatype `High` by exploiting the right-shoulder function, and *(ii)* the concept `SportCar` as a `Car` characterised by a very high speed. In order to define the concept *very high*, the modifier `very(High)` is exploited.

Fuzzy Tuples. Semantic tuples are formally described as SHOIN(D) individuals belonging to an ABox (see Section 3.2). Fuzzy SHOIN(D) makes it possible to represent a fuzzy ABox by associating a degree of truth to each concept and role membership axiom (see Section 6.1). For example, the following assertions can be defined

(3) $\texttt{SportsCar(audi\_tt)} \geq \texttt{0.92}$
(4) $\texttt{hasMaker(audi\_tt, audi)} \geq \texttt{1}$

in order to describe *(i)* the individual `audi_tt` belonging to the concept `SportCar` with degree `0.92`, and *(ii)* the role `hasMaker` in which the individual `audi_tt` is involved with the individual `audi` with degree `1`. Hence, in order to describe a fuzzy semantic tuple, a language is needed that, besides describing the name, the concept and the set of roles associated to an individual as for crisp semantic tuples (see Section 3.2), could also describe the degree of truth related to concept and role memberships.

Fuzzy Templates. Semantic tuple templates are formally described as SHOIN(D) concept descriptions by exploiting the TBox formalism (see Section 3.2). Fuzzy SHOIN(D) allows fuzzy concepts to be represented using the fuzzy TBox formalism (see Section 6.1). As a consequence, in order to define fuzzy templates, a language needs to be defined that support such a formalism. Hence, if we are interested in a semantic tuple describing individuals belonging to the concept `SportCar` with degree $\geq$ `0.8`, the fuzzy template should provide a concept description like `[`$\geq$ `0.8] SportCar` (see Section 6.1).

Fuzzy Primitives. Semantic tuple centre primitives – *in*, *rd* and *out* – represent the coordination language whereby system components can read, consume and write knowledge described by means of a domain ontology [63]. Extending the semantic tuple centre model with fuzziness also requires the semantic of the primitives *in* and *rd* to be suitably extended. The first extension consists, in face of a fuzzy template, to return a fuzzy tuple along with the degree by which it satisfies the template, since this could be different from `1`. Then, one has to decide which individual to retrieve, possibly based on such a degree. A first alternative is to retrieve the first individual found without bothering about the degree. A less efficient but more effective alternative is to retrieve the individual with highest degree. However, in this case the original model of tuple-based coordination where the retrieved tuple is chosen in a non-deterministic way would somehow be violated. Another alternative is to choose the individual on the basis of a probability distribution depending on the degree. Among the different alternatives, there is not a best solution: the best approach depends on the requirements of the application scenario in which the fuzzy semantic tuple centres have to be used. Hence supporting all those alternatives seems the more appropriate choice.

Fuzzy Reactions. A semantic reaction specification, as defined in Section 3.2, consists in an event E describing the set of events Ev for which reaction R has to be executed. In particular, E contains two sorts of information: the primitive (*in*, *rd* or *out*) to be intercepted and an individual set specication, describing either the possible concept descriptions in an *in*/*rd* primitive, or the possible individuals in an *out* primitive. As individual set specification, it is possible to exploit the language of fuzzy TBox which is also exploited for fuzzy semantic templates. Thus, for example E could contain the concept description `[`$\geq$ `0.8] Car`. If E refers to a primitive *in* or *rd*, R will be executed if the concept description in the semantic tuple template is a subkind of `[`$\geq$ `0.8] Car`—as for the concept description `CityCar` $\sqcap$ `(`$\exists$`hasMaker.fiat)`. Whereas, in case E refers to the primitive *out*, R will be executed if an individual of the kind `[`$\geq$ `0.8] Car` is inserted in the tuple centre—as for the individual `audi_tt` defined in axioms `(3-4)`. As far as reactions are concerned, besides accessing all the information related to the triggering communication event, they can read, remove and write tuples from/to the tuple centre. Accordingly, like coordinated components, reactions can contain coordination primitives to access and modify the semantic knowledge stored in the tuple centre, through fuzzy tuples and fuzzy tuple templates.

Fuzzy Matching Mechanism. As seen in Section 3.2, the semantic tuple matching mechanism requires to execute the *instance checking* – verifying whether a given individual is an instance of a specified concept –, the *instance retrieval* – finding the individuals in the knowledge – and the *subsumption checking* of two concepts `C` and `D`—which can be used in the case of reactions to *in*/*rd*. Accordingly, there are two ways to support the fuzzy matching mechanism. The first is to exploit an existing fuzzy SHOIN(D) reasoner. To the best of our knowledge, the only reasoner supporting fuzzy SHOIN(D) is DeLorean [11]. The advantage in using DeLorean is that it represents fuzzy SHOIN(D) using crisp SHOIN(D), thus reducing the reasoning within fuzzy SHOIN(D) to reasoning within crisp SHOIN(D). As a consequence, it would be possible to translate fuzzy SHOIN(D) into a crisp ontology language like OWL DL and to use currently available SHOIN(D) reasoners like Pellet. However, the implementation is not yet available, and the solution has still performance problems. An alternative to this solution is to simplify the problem by omitting fuzzy domain ontology and to develop a module – that we call *(de)fuzzificator* – in charge of bridging between the tuple centre interface and a crisp SHOIN(D) reasoner. In particular, the task of the (de)fuzzificator is to transform a fuzzy individual description in a crisp one when the corresponding tuple is inserted in the tuple centre. The crisp individual is stored in the crisp SHOIN(D) reasoner whereas the corresponding fuzzy individual is stored in the tuple centre. Then, in face of a reading or consuming operation, the task of the (de)fuzzificator is to interpret the fuzzy semantic tuple template as a crisp semantic tuple template, to query the crisp reasoner through the *instance retrieval*, and finally to retrieve an individual with the degree with which it matches the template. In order to calculate such a degree, the fuzzy individual version stored in the tuple centre needs to be used. Whereas, concerning reactions, in case of a reaction to an *out* it is possible to exploit the *instance checking* function of the crisp reasoner in order to know if the individual belongs to the particular concept. If so, again in order to obtain the degree to which the individual belongs to the concept, the fuzzy individual version stored in the tuple centre needs to be used. Then, in case of a reaction to an *in* or *rd*, it is possible the exploit the *subsumption checking* function of the crisp reasoner in order to know if the concept `C` defined in E, subsumes the concept `D` defined in the primitive. Moreover, it is also needed to check the conditions in `[...]` associated to `C` and `D`. In particular, it is needed to check if the conditions in `[...]` do not invalidate the subsumption between `C` and `D`.

6.3 Fuzzyfying TuCSoN

In the following we show how semantic tuple centres provided by the coordination infrastructure TuCSoN could be extended with fuzziness.

Fuzzy Domain Ontology. As sketched in Section 6.2, an ontology language following fuzzy-SHOIN(D) approach is needed to associate a fuzzy domain ontology to a semantic tuple centre. In [63], OWL DL was chosen as the ontology language, since it is a W3C standard. In literature, fuzzy extensions to OWL are proposed, however, the drawback of those solutions is that they are not standard—and one cannot reuse existing reasoners, for they do not deal with fuzziness. Thus, in the following we assume that the domain ontology is only crisp and that the ontology language is OWL DL.

Fuzzy Tuples. As described in Section 6.2, a fuzzy tuple has to describe an individual in terms of the individual name, and the concept and role membership with associated degree of truth. Accordingly, the language for semantic tuples defined for TuCSoN and sketched in Section 4.2 becomes as follows:

```
Individual ::= iname ‘:’ C
C ::= cname [→ α] | cname ‘(’ R [→ α] ‘)’
R ::= rname ‘:’ V | rname ‘in’ ‘(’ Vset ‘)’ | R ‘,’ R
Vset ::= V [→ α] | V [→ α] ‘,’ Vset
V ::= iname | number | string
```

The extensions to the language are highlighted in grey. In particular, by considering the generic membership axiom τ and the degree α, the tuple language is extended with the possibility of constraining the degree of τ to be α. For the sake of simplicity, the relation between τ and α can only be of kind $\geq$. Moreover, as described by the language grammar, the definition of the degree is not required. If a degree is not defined, it defaults to 1. By supposing we already defined the car domain ontology, with such a grammar we could obtain the following tuples:

```
ca:‘CityCar’(hasMaker:ford, hasMaxSpeed:130,
             hasColour in (red → 0.7 , black → 0.3 ))

audiTT:‘SportCar’ → 0.8 (hasMaker:audi, hasMaxSpeed : 260,
                         hasColour : black)
```

The first tuple asserts that ca is a CityCar with degree 1, is hasMaker-related to ford with degree 1, is hasMaxSpeed-related to 130 with degree 1, and is hasColour-related to red and black, respectively with degree 0.7 and 0.3. Then, the second tuple asserts that audiTT is a SportCar with degree 0.8, is hasMaker-related to audi with degree 1, is hasMaxSpeed-related to 20 with degree 1, and is hasColour-related to black with degree 1.

Fuzzy Templates. According to Section 6.2, a fuzzy semantic template is a fuzzy specification of a set of domain individuals. As such, a fuzzy template is a description of a concept in the fuzzy TBox formalism. In order to describe a semantic template as a *fuzzy concept* as described in Section 6.2, support for *fuzzy datatypes* `dt`, *fuzzy modifiers* `m(C)`, *fuzzy nominals* $\{\alpha_1/\mathtt{o}_1,\ldots,\alpha_m/\mathtt{o}_m\}$, *weighted concepts* $(\alpha\ \mathtt{C})$, *weighted sum concepts* $(\alpha_1 C_1 + \ldots + \alpha_N C_N)$, and *threshold concepts* $(\mathtt{C}[\geq \alpha])$ and $(\mathtt{C}[\leq \alpha])$ is required. In particular, for *fuzzy datatypes* and *fuzzy modifiers* we adopt the functions defined by Straccia et al. in [53]: `crisp(a,b)`, `L(a,b)`, `R(a,b)`, `triangular(a,b,c)` and `trapezoidal(a,b,c,d)`. Accordingly, the language for semantic templates defined for TuCSoN and sketched in Section 3.2, becomes as follows (extensions highlighted in gray):

```
C ::= ‘$ALL’ | ‘$NONE’ | cname | C ‘,’ C |
      C ‘;’ C | ‘$not’ C | D |
      ‘{’ [ iname [→ α] { ‘,’ , iname [→ α] } ] ‘}’ |
      C ‘(’ D ‘)’ | ‘(’ C ‘)’ |
      CW | C‘[’≥ α‘]’ | C‘[’≤ α‘]’ |
      DT | M ‘(’ C ‘)’

CW ::= C→ α |CW + CW

DT ::= crisp‘(’N‘,’N‘)’ | l‘(’N‘,’N‘)’ |
      r‘(’N‘,’N‘)’ | M

M ::= triangular‘(’N‘,’N‘,’N‘)’ |
     trapezoidal‘(’N‘,’N‘,’N‘,’N‘)’

D ::= F | ‘$exists’ F |‘$all’ F | M

F ::= R ‘in’ C | R ‘:’ I | R ‘:’ Msymb N |
     R ‘:=’ string | R

M ::= ‘#’ R Msymb N

R ::= rname | rname ‘/’ vname

Msymb ::= ‘>’ | ‘<’ | ‘≥ ’ | ‘≤ ’ | ‘=’
```

Referring to the car domain, with such a grammar we could obtain e.g. the following fuzzy semantic templates:

```
‘Car’, ((hasMaxSpeed:280→ 0.6) +
          (hasMaker:ferrari→ 0.4))
```

```
[≥ 0.8] 'SportCar'
```

```
'Car', ($exists hasSpeed in linear(0.8)(r(80,250)) )
```

The first tuple defines a concept describing a set of individuals represented of concept `Car` and possibly `hasMaxSpeed` equal to `180` and `hasMaker` equal to `ferrari`. The second concept describes a set of individuals represented by `SportCar` with degree equal to or greater than 0.8. Then, the third concept describes a set of individuals represented by `Car` with *speed very high*, where the concept *high* is represented by `r(80,250)` whereas the concept *very(x)* is represented by the modifier `linear(0.8)(x)`.

Fuzzy Primitives. According to Section 6.2, the primitive *out* for inserting a semantic tuple in a tuple centre remains unchanged. Thus, taking the syntax of *out* primitive shown in [63], we can write:

```
out(semantic audiTT : 'SportCar'→ 0.8(hasMaker : audi,
                                  hasMaxSpeed : 260,
                                  hasColour : black))
```

in order to insert a fuzzy tuple in a tuple centre. For the primitives *in* and *rd*, the degree of matching has to be retrieved together with the selected fuzzy tuple. Thus, the syntax of *in* and *rd* primitives [63] is extended to make it possible to write:

```
in(semantic (X degree Y matching ([≥ 0.8] 'SportCar')))
```

The above operation unifies `X` with the fuzzy tuple matching the template `[≥0.8] 'SportCar'`, and `Y` with the matching degree. As far as the *in* and *rd* primitives are concerned, it has to be decided which tuple to retrieve on the basis of the matching degree with the template. Since TuCSoN provides general purpose coordination media, along with coordination contexts to provide coordinable with different coordination models, the best solution is to provide all the three alternatives proposed in Section 6.2. In particular, we define the following primitives:

```
in(semantic (X degree Y matching ([≥ 0.8]'SportCar')))
in_max(semantic (X degree Y matching ([≥ 0.8]'SportCar')))
in_prob(semantic (X degree Y matching ([≥ 0.8]'SportCar')))
```

retrieving respectively the first tuple matching with the template with a degree $>$ `0`, the tuple with the best degree and a tuple chosen on the basis of a probability distribution depending on the matching degree. In particular, the probability of retrieving a tuple is its degree divided by the sum of the degrees of all the matching tuples.

Moreover, thanks to the programmability of the tuple centres exploited in TuCSoN, it is also possible to implement more complex operations by exploiting the specification language ReSpecT. For example, a set of interesting operations could be devised as follows:

- if the operations *in* and *rd* retrieve the tuple with the highest resulting degree, one could define the operations *in_min* and *rd_min* retrieving the tuple with the lowest resulting degree;
- moreover, one could define the operations *in_all* and *rd_all* retrieving the entire set of tuple with the resulting degree, sorted in ascending way. Also, one could allow the specification of the max cardinality of the resulting set in order to avoid obtaining a too huge set of tuples as the result;
- dually, one could define the operations *in_all_min* and *rd_all_min* retrieving the set of tuples with the resulting degree, sorted in descending way.

Fuzzy Reactions. As described in Section 4.2, from a semantic point of view, in a ReSpecT reaction `reaction(E, (G, R))`, `E` represents a specification of a coordination primitive related to a concept description (*in*/*rd*) or related to a domain individual (*out*). Accordingly, besides the name of the primitive invoked (*out*, *in* or *rd*), `E` should contain a concept description expressed in terms of the semantic template language. For example, `E` could contain the concept description `[≥ 0.8] Car`, so for instance a reaction to the primitive *in* could look like:

```
  reaction( in(semantic ([≥ 0.8] 'Car')), ..., ... )
```

The reaction could be triggered by a primitive *in* containing for example the concept description `CityCar ⊓ (∃hasMaker.fiat)`, that is

```
  in(semantic (X degree Y matching ('CityCar' and (exists hasMaker : fiat))))
```

since `1`, implicitly defined for the concept `CityCar ⊓ (∃hasMaker.fiat)`, is greater than `0.8` defined for `Car`. Instead, the following reaction:

```
  reaction( out(semantic ([≥ 0.8] 'Car')), ..., ... )
```

would be triggered when an individual of kind `Car` with degree greater or equal to `0.8`, is inserted in the tuple centre, e.g., by the invocation:

```
out(semantic audiTT : 'SportCar'→ 0.8(hasMaker : audi,
                          hasMaxSpeed : 260,
                          hasColour : black))
```

inserting the individual `audiTT` in the semantic tuple centre.

The reaction guard `G` represents a set of conditions to be satisfied in order to execute a reaction `R` if the current event `Ev` matches `E`. Since `G` represents a set of constraints on `Ev` and `E`, ReSpecT is extended so that the guard could contain a concept description in the fuzzy template language, as in the following reaction:

```
reaction( out(semantic([≥ 0.8] ‘Car’)),semantic(exists hasMaker : audi), ... )
```

Finally, also the ReSpecT reaction language needs to be extended, since it can read, remove and write tuples from/to the tuple centre, now including semantic tuples. Thus, `R` can now contain fuzzy primitives, as in the following example:

```
reaction( out(semantic([≥ 0.8] ‘Car’)),semantic(exists hasMaker : audi), (
          rd (semantic (audi degree Y matching ‘Maker’
                        and (exists hasCars / N in Int))),
          N1 is N + 1,
          out(semantic audi : ‘Maker’(hasCars : N1)) ))
```

There, if an incoming event matches with `out(semantic [≥ 0.8]‘Car’)`, and the guard `(semantic (exists hasMaker : audi))` is satisfied, then the individual `audi` is updated with a new value for the relation `hasCars`.

Fuzzy Matching Mechanism. Here, we only focus on fuzzy primitives, omitting the fuzzy matching mechanism exploited in reactions. In case a fuzzy SHOIN(D) reasoner is not available, according to Section 6.2, in order to support matching between fuzzy semantic tuples and templates, a *(de)fuzzificator* needs to be added, bridge the tuple centre interface and the adopted DL reasoner. In particular, when a new tuple is inserted in the tuple centre, the (de)fuzzificator should separate the degrees associated to the tuple in order to obtain a non-fuzzy individual-assertion description in the DL language of the specific DL reasoner adopted. The degrees remains stored only in the tuple inserted in the tuple centre. Then, in order to understand what happens when a tuple template is provided to a tuple centre, it is important to highlight that:

- A template is a composition of concept descriptions obtained by exploiting the operators defined by the grammar language sketched in Section 4.2.

- Each operator has its own priority. So, for example, in order to interpreter the template `‘Car’, (exists hasMaker : ford)` we can represent it through the tree shown in Figure 6.2;

- Concerning the template grammar extended here, after `not`, the operators with the highest priority are →, `[ ≤ ... ]` and `[ ≥ ... ]`, then the operator `+` used for the weighted sum concept `(`$\alpha_1 C_1$ `+ ... +` $\alpha_N C_N$`)` and the set `crisp`, `l`, `r`, `triangular` and `trapezoidal`.

- The semantic of the SHOIN(D) operators ($\sqcap$, $\sqcup$, $\neg$, $\forall$, $\exists$, $\leq$, and $\geq$) changes: they became fuzzy operators. In particular, we adopt the fuzzy operators defined by Straccia et al. in [12].

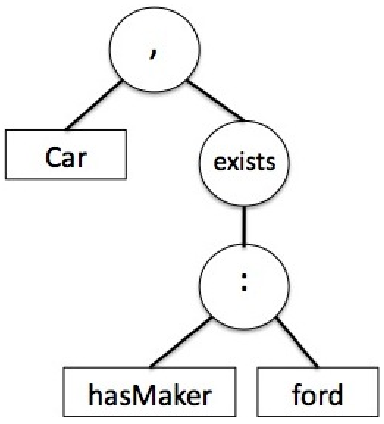


Figure 6.2: Syntax Tree Representing a Semantic Template

Said that, in face of a fuzzy tuple template, the template syntax tree has to be visited from bottom to top (e.g. in post-order). In leafs, the (de)fuzzificator should:

- query the DL reasoner in order to obtain the set of individuals belonging to the concept in current node;
- recover from the tuple centre the degrees related to each obtained individual;
- return the set of individuals, each along the degree resulting from the application of the fuzzy operators.

In other nodes, which correspond to fuzzy operators, the set of individuals has to be properly combined, and degrees are to be re-computed. For example, taking the fuzzy tuple template `'SportCar'→0.8; (hasMaker:audi)→0.2` and the related tree shown in Figure 6.3, the (de)defuzzificator:

- queries the DL reasoner for individuals belonging to the concept `SportCar`;
- calculates for each of them its degree as the product of 0.8 by the degree as specified in the tuple centre;
- queries the DL reasoner for individuals belonging to the concept `hasMaker:audi` by querying the DL reasoner;
- calculates for each of them its degree as the product of 0.2 by the degree as specified in the tuple centre;

- executes the fuzzy operator `;` (the counterpart of SHOIN(D) operator $\sqcup$) over the two above sets of individuals. By using e.g. the implementation based on *Lukasiewicz t-conorm*, we join the two sets and update degrees of each tuple by formula `min`$\{\alpha+\beta$`,1`$\}$), where α and β are the two originatind degrees—0 if the tuple was not in the set.

Accordingly, supposing to have the following tuples:

```
audiTT : 'SportCar'→ 0.8(hasMaker : audi, hasMaxSpeed : 260,
                          hasColour : black)

f380 : 'SportCar'→ 0.9(hasMaker : ferrari, hasMaxSpeed : 320,
                        hasColour : red)
```

Then matching the fuzzy tuple template `'SportCar'→0.8; (hasMaker:audi)→0.2` returns tuple `audiTT` with degree `0.84` and `f380` with degree `0.72`.

6.4 An Application Scenario

In this section we aim at providing an example of how fuzzy semantic tuple centres could be fruitfully used. So, we exploit fuzzyfied TuCSoN within an application environment where tuple centres provide a suitable coordination media, and show how adding semantic and fuzziness to tuple centres results in a more powerful coordination model.

A *Virtual Enterprise* (VE) [76] is a temporary aggregation of autonomous and possibly heterogeneous enterprises, meant to provide the flexibility and adaptability to frequent changes that characterise the openness of business scenarios. As shown in [75], TuCSoN infrastructure addresses in a good way the requirements of the VE application scenario represented by *(i)* the integration of selected heterogeneous resources provided by single participants and *(ii)* the integration and coordination of distributed business activities.

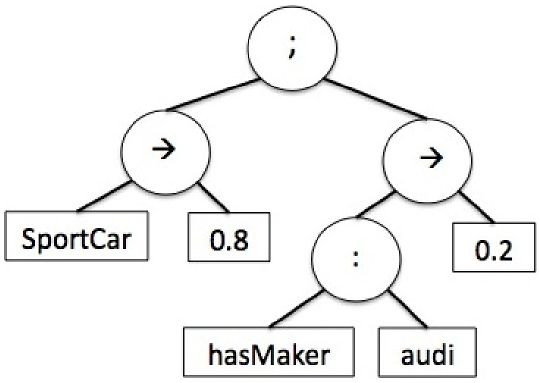


Figure 6.3: Syntax Tree Representing a Fuzzy Semantic-Template

However, the original model of TuCSoN tuple centres is based on a syntactic matching. As a consequence, it does not fit very well requirements like openness – heterogeneity and dynamism – of the information exchanged among different enterprises. In particular, the information exchanged by each participant could have a different syntactic structure even if it is semantically equivalent. Then, enterprises could be not able to have *a priori* knowledge about all possible information belonging to the application scenario. Tuple centres enhanced with both semantics and fuzziness allow in principle to overcome such limitations.

To give an evidence of such an assertion, let us take an example of VE application scenarios: a *virtual bookshop* [75]. The virtual bookshop is a VE aggregating several companies of different sorts to sell books through the Internet. There, four basic roles could be identified: the *bookseller* (providing the books), the *carrier* (delivering books from sellers to customers), the *interbank service* (executing payment transactions) and the *Internet service provider* (the Web portal of the customer). For the sake of simplicity, here we only consider the problem of knowledge representation about the *book domain*, that is, the knowledge mainly exchanged among *booksellers* and *Internet service providers* in order to coordinate them with one another.

In this context, the first problem to be faced concerns the different syntactic structure of information. For example, when a Web portal receives a request for a book of genre *fantasy*, whereas sellers only have books of genre *classic fantasy* and *contemporary fantasy*, a syntactic approach could not match the genres. Moreover, a communication based on concepts rather than on "instances" of concepts is more flexible: often VE participants do not mean to refer to particular instances but rather to their (partial) descriptions. Thus, for instance, a request from a Web portal may not identify exactly the title of a specific book – maybe because the requesting user does not know the precise title –, but might concern books of genre *classic fantasy* authored by a specific writer. As shown in [63], such limitations could be overcome by exploiting semantic tuple centres where knowledge is represented in terms of a domain ontology. Thus, supposing information about books is stored in TuCSoN semantic tuple centres associated to an opportune *book ontology*, it would be possible to exploit the primitive *in* in this way: `in(semantic (X matching ('Fantasy')))`. Through such a primitive, it would be possible to obtain fantasy books, including classic and contemporary ones.

A second problem to be faced is the management of vague/imprecise information. For example, in the book domain there could be vague/imprecise information like *books for kids* or *books for adults.* Then, books could belong to a category with a given degree—for example a book could be *fantasy* with degree 0.7. Finally, participants of VE application may be not interested only in knowledge perfectly matching their requests. For example, in the *virtual bookshop* context, a user could submit her preferences through a Web portal aiming at obtaining a ranked list of books matching with her preferences. In order to face all those requirements, fuzziness in TuCSoN tuple centres is clearly of help. First, it is possible to define fuzzy concepts like `linear(0.8)(l(80,250))` (see Section 6.3) to represent the concept *kids* and the concept `($exists forAge in linear(0.8)(r(80,250)))` to describe the set of books for kids. Then, those concepts could be exploited as a part of a tuple template in order to obtain book information through the primitive *in* or *rd*. Moreover, in order to describe books belonging to categories with a degree one could exploit tuples like `book1:'Fantasy'`$\rightarrow$ `0.7` describing a book of kind fantasy with degree 0.7. Finally, in order to obtain a ranked list of books matching a Web provider request one could exploit the primitive *in_all* or *rd_all* described in Section 6.3.

6.5 Summary

In this chapter we extended the model of semantic tuple centres with fuzziness as a fundamental aspect of coordination to face vagueness and imprecision characterising real-world information. After discussing fuzzy SHOIN(D), we presented the elements required to support fuzziness in semantic tuple centres. Then, we shown how to apply the new model in the semantic TuCSoN. However, in this chapter we only provided a model of semantic tuple centres supporting fuzziness. We did not provide a real implementation of this kind of tuple centres and an evaluation of it. Thus, among the future direction of this work, we aim at experiment the model of TuCSoN semantic tuple centre extended with fuzziness in order to evaluate the pros and cons of our approach. Then, we aim at experiment the implementation with a specific case study as shown in Section 6.4.

Chapter 7

Service Coordination in Pervasive Systems

The Information and Communication Technology (ICT) landscape – yet notably changed by the advent of ubiquitous wireless connectivity – will further re-shape due to the increasing deployment of computing technologies like pervasive services and social networks. Addressing this scenario calls for finding infrastructures promoting a concept of pervasive "eternality", namely, changes in topology, device technology and continuous injection of new services have to be dynamically tolerated as much as possible, and incorporated with no significant re-engineering costs at the middleware level [101, 94]. As far as the coordination of such services is concerned, it will increasingly be required to tackle self-organisation (supporting situatedness, adaptivity and long-term accommodation of diversity) as an inherent system property rather than a peculiar aspect of the individual coordinated components. As typical in self-organising computational mechanisms, a promising direction is to take inspiration from natural systems (e.g. physical, chemical, biological, social [94]), where self-organisation is intrinsic to the basic "rules of the game".

Focussing on chemical natural systems, this chapter first shows the concept of chemical tuple spaces [91] – tuple spaces programmed with coordination rules resembling chemical reactions – as suitable coordination media for situated and adaptive pervasive computing [77, 17, 46, 56]. Then, it is shown how a distributed architecture for chemical tuple spaces [91] can be implemented in TuCSoN providing semantic ReSpecT tuple centres.

7.1 Requirements for Self-Organising Pervasive Service Systems

In order to better explain the motivation behind the model presented in this chapter, we rely on a case study, which we believe well represents a large class of pervasive computing applications in the near future.

We consider a pervasive display infrastructure, used to surround our environments with digital displays, from those in our wearable devices and domestic hardware, to wide wall-mounted screens that already pervade urban and working environments [30]. However, instead of considering displays as static information servers as usual nowadays (i.e. showing information in a manually configured manner), we envision a truly general, open, and adaptable information service infrastructure. As a reference domain, we consider an airport terminal filled with wide screens mounted in the terminal area, i.e in the shops, in the corridors and in the gates, down to tiny screens installed in each seat of the gate areas or directly on passengers' PDAs.

Situatedness We first notice that information should be generally displayed based on the current state of the surrounding physical and social environment. For instance, by exploiting information coming from surrounding temperature sensors and passenger profiles/preferences, an advertiser could decide to have ice tea commercials – instead of liquor ones – displayed on a warm day and in a location populated by teenagers.

Thus, a general requirement for pervasive services is *situatedness*. Namely, pervasive services deal with spatially and socially situated users' activities, hence, they should be able to interact with the surrounding physical and social world, accordingly adapting their behaviour. As a consequence, the infrastructure itself should act based on spatial concepts and data.

Adaptivity Secondly, and complementary to the above, the display infrastructure, and the services within it, should be able to automatically adapt to changes and contingencies in an automatic way. For instance, when a great deal of new information to be possibly displayed emerges, the displayed information should overall spontaneously re-distribute and re-shape across the set of existing local displays, possibly discharging obsolete visualisation services.

Accordingly, another requirement is *adaptivity*. Pervasive services and infrastructures should inherently exhibit self-adaptation and self-management properties, so as to survive contingencies without any human intervention and at limited management costs.

Diversity Finally, the display infrastructure should be not only intrinsically open to any kind of visualisation services that may be added to the system, but also able to allow users – other than display owners – to upload information to displays so as to enrich the information offer or adapt it to their own needs. For instance, a passenger could watch private content uploaded from her/his PDA to a wider screen close to her/his seat, and may be willing also to share it with people nearby. Put simply, users should act as "prosumers"—i.e. as both consumers and producers of devices, data, and services.

Another general requirement is hence *diversity*. Namely, the infrastructure should tolerate open models of service production and usage without limiting the number and classes of services potentially provided, but rather taking advantage of the injection of new services by exploiting them to improve and integrate existing services whenever possible.

7.2 Chemical-inspired Tuple Spaces for Pervasive Services

In many proposals for pervasive computing environments and middleware infrastructures, situatedness and adaptiveness are promoted by the adoption of shared virtual spaces for services and component interaction [77, 17, 46, 56, 91]. In these approaches, tuple spaces are disseminated in the pervasive environment, one in each network location, and reify the local situation in terms of tuples (structured information items). Depending on the specific proposal, such tuples can represent the occurrence of people nearby, the availability of devices, the state of pervasive services, knowledge, contextual information, signals spread in the network, and so on. Relying on such shared virtual spaces has a main implication: system coordination can be achieved by a rather simple set of rules for managing tuples, acting locally to each space and providing for the "laws" by which such tuples evolve, diffuse, and possibly combine so as to support adaptivity [56].

Among the various tuple space models proposed in literature, we adopt the chemical tuple space model [91], in which tuples – containing information about the "individuals" to be coordinated (services, devices, data) – evolve in a stochastic and spatial way through coordination laws resembling chemical reactions. We observe that this model can properly tackle the requirements sought for adaptive pervasive services.

Concerning *situatedness*, the current situation in a system locality is represented by the tuples existing in the tuple space. Some of them can act as catalysts for specific chemical reactions, thus making system evolution intrinsically context-dependent. Moreover, mechanisms resembling chemical diffusion can be designed to make a node influencing neighbouring ones.

Concerning *adaptivity*, it is known from biology that some complex chemical systems are auto-catalytic (i.e. they produce their own catalyst), providing positive-negative feedbacks that induce self-organisation [15] and lead to the spontaneous toleration of environment perturbations. Such systems can be modelled by simple idealised chemical reactions, e.g. prey-predator systems, Brussellator, and Oregonator [38]. Regarded as a set of coordination laws for pervasive services, this kind of chemical reactions has the potential to intrinsically support adaptivity.

Finally, considering *diversity*, we note that chemical reactions follow a simple pattern: they have some reactants (typically 1 or 2) which combine, resulting in a set of products, through a propensity (or rate) dictating the likelihood for this combination to actually happen. In natural chemistry, this generates a plethora of specific chemical reactions that take into account the diversity of chemical species, and the possibility of creating complex molecular structures. In our framework, general reactions will be designed that can be instantiated for the specific and unforeseen services that will be injected in the system over time—using semantic matching to fill the gap.

Though key requirements seem to be supported in principle, designing the proper set of chemical reactions to regulate system behaviour is crucial. Without excluding the appropriateness of other solutions, in this chapter we mostly rely on chemical reactions resembling laws of population dynamics as, e.g. the prey-predator system [38, 9]. This kind of idealised chemical reactions has been successfully used to model auto-catalytic systems manifesting self-organisation properties: but moreover, they can also nicely fit the "ecological" metaphor that is often envisioned for pervasive computing systems [7, 87, 1, 94, 101]—namely, seeing pervasive services as spatially situated entities living in an ecosystem of other services and devices.

7.3 The Coordination Model of Chemical Tuple Spaces

In this section, first we informally introduce the coordination model of chemical tuples spaces (Section 7.3.1), then describe some example applications in the context of competitive pervasive services (Section 7.3.2).

7.3.1 Coordination Model

The chemical tuple space model is an extension of standard Linda settings with multiple tuple spaces [35]. A Linda tuple space is simply described as a repository of tuples (structured data chunks like records) for the coordination of external "agents", providing primitives used respectively to insert, read, and remove a tuple. Tuples are retrieved by specifying a tuple template—a tuple with wildcards in place of some of its arguments. The proposed model enhances this basic schema with the following ingredients.

Tuple Concentration and Chemical reactions

We attach an integer value called "concentration" to each tuple, measuring the pertinence/activity value of the tuple in the given tuple space: the higher such a concentration, the more likely and frequently the tuple will be retrieved and selected by the coordination laws to influence system behaviour. Tuple concentration is dynamic, as typically the pertinence of system activities is. In particular, tuple concentration "spontaneously" evolves similarly to what happens in chemical behaviour, namely, a tuple with concentration N is handled pretty much in the same way as if it were a chemical substance made of N molecules of the same species. This is achieved by coordination rules in the form of chemical reactions—the only difference with respect to standard chemical reactions is that they now specify tuple templates instead of molecules. For example, a reaction "$X + Y \xrightarrow{0.1} X + X$" would mean that tuples x and y matching X and Y are to be selected, get combined, and as a result one concentration item of y turns into x—concentration of y decreases by one, concentration of x increases by one. According to [38], this transition is modelled as a Poisson event with average rate (i.e. frequency) $0.1 \times \#x \times \#y$ ($\#x$ is the concentration of x). In the general case, such a rate is obtained by multiplying reaction rate by a coefficient depending scontributeolely on the concentration of reactants in the solution: in particular, the contribution of each reactant in the product is $\binom{m}{n}$, where m is the concentration of the reactant in the solution, and n is the number of such reactants existing in the chemical reaction (typically, $n = 1$ or $n = 2$). This model makes a tuple space running as a sort of exact chemical simulator, picking reactions probabilistically: external agents observing the evolution of tuples would perceive something equivalent to the corresponding natural/artificial chemical system described by those reactions.

Semantic Matching

It is easy to observe that standard syntactic matching for tuple spaces can hardly deal with the openness requirement of pervasive services, in the same way as syntactic matchmaking has been criticised for Web services [72]. This is because we want to express general reactions that apply to specific tuples independently of their syntactic structure, which cannot be clearly foreseen at design time. Accordingly, *semantic matching* can be considered as a proper matching criterion for our coordination infrastructure [72, 5, 12].

It should be noted that matching details are orthogonal to our model, since the application at hand may require a specific implementation of them—e.g. it strongly depends on the description of application domain. As far as the model is concerned, we only assume that matching is fuzzy [12], i.e. matching a tuple with a template returns a "vagueness" value between 0 and 1, called *match degree.* Vagueness affects the actual application rate of chemical reactions: given a chemical reaction with rate r, and assume reactants match some tuples with degree 0.5, then the reaction can be applied to those tuples with an actual rate of $0.5{*}r$, implying a lower match likelyhood—since match is not perfect. Namely, the role of semantic matching in our framework is to allow for coding general chemical laws that can uniformly apply to specific cases—appropriateness influences probability of selection.

Tuple Transfer

We add a mechanism by which a (unit of concentration of a) tuple can be allowed to move towards the tuple space of a neighbouring node, thus generating a *computational field*, namely, a data structure distributed through the whole network of tuple spaces. Accordingly, we introduce the notion of "firing" tuple (denoted $t^{\leadsto}$), which is a tuple (produced by a reaction) scheduled for being sent to a neighbouring tuple space—any will be selected non-deterministically. For instance, the simple reaction "$X \xrightarrow{0.1} X^{\leadsto}$" is used to transfer items of concentration of any tuple matching X out from the current tuple space.

7.3.2 Examples

We now discuss some examples of chemical reactions enacting general coordination patterns of interest for pervasive service systems. We proceed incrementally, first providing basic laws for service matching and competition, which will be then extended towards a distributed setting.

Local competition

We initially consider a scenario in which a single tuple space mediates the interactions between pervasive services and their users in an open and dynamic system. In the pervasive display infrastructure, this example is meant to model the basic case where, given the node where a display is installed, visualisation services are to be selected based on the profile of users nearby the display. We aim at enacting the following behaviour: *(i)* services that do not attract users fade until eventually disappearing from the system, *(ii)* successful services attract new users more and more, and accordingly, *(iii)* overlapping services compete one another for survival, so that some/most of them eventually come to extinction.

An example protocol for service providers can be as follows. A tuple `service` is first inserted in the space to model publication, specifying service identifier and (semantic) description of the service content. Dually, a client inserts a specific request as a tuple `request`—insertion is the (possibly implicit) act of publishing user preferences. The tuple space is charged with the role of matching a request with a reply, creating a tuple `toserve(service,request)`, combining a request and a reply semantically matching. Such tuples are read by the service provider, which collects information about the request, serves it, and eventually produces a result emitted in the space with a tuple `reply`, which will be retrieved by the client. The abstract rules we use to enact the described behaviour are as follows:

$$\begin{array}{lrcl}
\texttt{(USE)} & \texttt{SERV} + \texttt{REQ} & u & \texttt{SERV} + \texttt{SERV} + \texttt{toserve(SERV,REQ)} \\
\texttt{(DECAY)} & \texttt{SERV} & d & 0
\end{array}$$

On the left side (reactants), `SERV` is a template meant to match any service tuple, `REQ` a template matching any request tuple; on the right side (products), `toserve(SERV,REQ)` will create a tuple having in the two arguments the service and request tuples selected, while `0` means there will be no product. Rule (USE) has a twofold role: *(i)* it first selects a service and a request, it semantically matches them and accordingly creates a `toserve` tuple, and dynamically removes the request; and *(ii)* it increases service concentration, so as to provide a positive feedback—resembling the prey-predator system described by Lotka-Volterra equations [9, 38]. We refer to *use rate* of a couple service/request as u

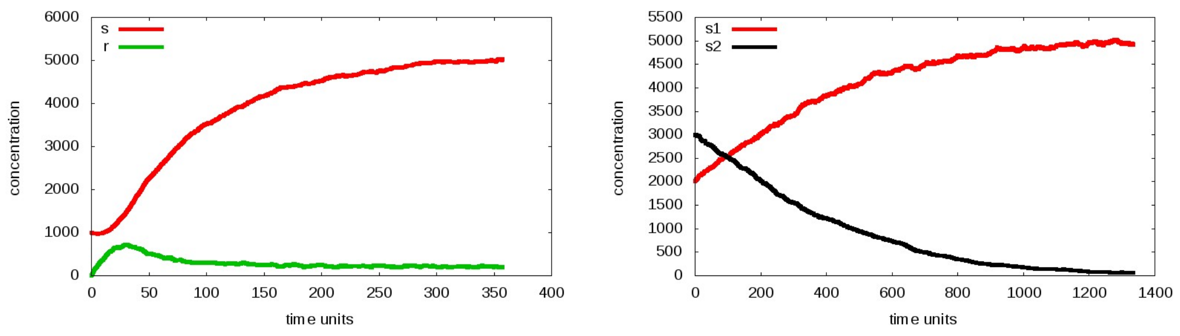


Figure 7.1: Service s exploited by matching requests r (left); and competition between services $s1$ and $s2$ (right).

multiplied by the match degree of those reactants when applying (USE) law, as described in the previous section: as a result, it can be noted that the higher the match degree, the more likely a service and a request are combined. On the other hand, rule (DECAY) makes any concentration item of the service tuple disappear at rate d, contrasting the positive feedback of (USE): here, the overall *decay rate* of a service is d multiplied by the match degree—with no match, we would have no decay at all.

In Figure 7.1 (left) we consider a scenario in which requests `r` are injected at average rate 50, and a matching service `s` exists in the system with initial concentration $1,000$: we additionally have decay rate 0.01 and use rate 0.05. We can observe that after an initial growth, the number of requests which are not served stabilises to few hundreds, while the concentration of `s` grows to about $5,000$. The behaviour of service concentration can be understood in terms of the positive-negative feedback loop of rules (USE) and (DECAY). It can be shown that service concentration increases while stabilising to about p/d, where p is the rate of injection of requests (pumping rate) and d is the service decay rate (decay rate d).

We now consider a similar scenario but with two services $s1$ and $s2$ initially having concentration $2,000$ and $3,000$ respectively, and matching the same requests, though with different use rate: 0.04 for $s1$, and 0.06 for $s2$. This models the situation in which two different services exist to handle requests, one leading to a better match. The result is that $s1$ and $s2$ engage a competition: this is lost by $s1$ which starts fading until completely vanishing (i.e. being disposed) even though it has an initially higher concentration, as shown in Figure 7.1 (right). In fact, the sum of the concentration of $s1$ and $s2$ still stabilises to $5,000$, but the contribution of $s1$ and $s2$ changes depending on the number of requests they can serve. Hence, matching degree is key when more services are concerned and the shape and dynamics of user requests is unknown, as it is responsible of the rate at which a service is selected each time, and ultimately, of the evolution of service concentration, i.e. of its competition/survival/extinction dynamics.

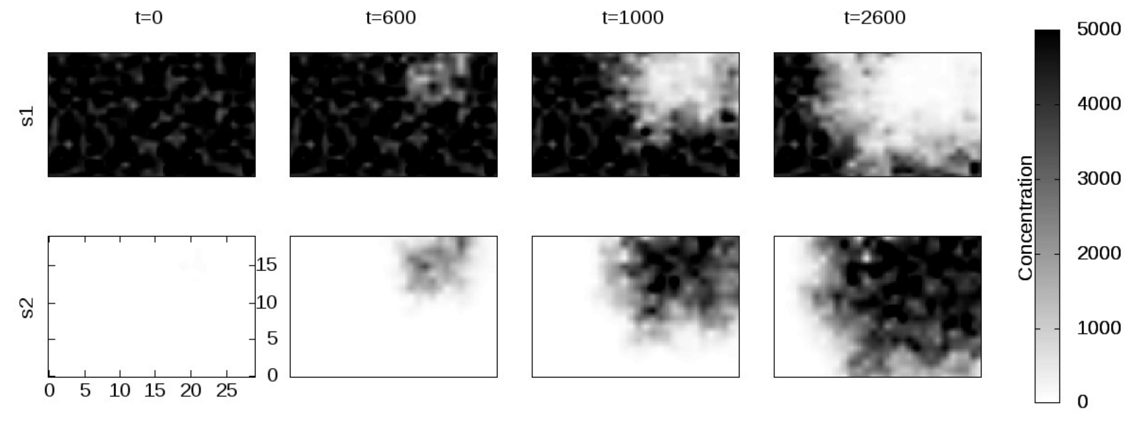


Figure 7.2: Spatial competition: after an initial pointwise injection, service `s2` (down) globally overcomes `s1` (top).

Spatial competition

This example can be extended to a network of tuple spaces, so as to emphasise the spatial and context-dependent character of competing services. Suppose each space is programmed with (USE,DECAY) reactions plus a simple diffusion law for service tuples:

$$\texttt{(DIFFUSE)} \qquad \texttt{SERV}\ m\ \texttt{SERV}^{\rightsquigarrow}$$

The resulting system can be used to coordinate a pervasive service scenario in which a service is injected into a node of the network (e.g. the node where service is more urgently needed, or where the producer resides), and accordingly starts diffusing around on a step-by-step basis until possibly covering the whole network—hence becoming a global service. This situation is typical in the pervasive display infrastructure, since a frequent policy for visualisation services would be to show them on any display of the network—although more specific policies might be enacted to make certain services only locally diffuse.

In this system, we can observe the dynamics by which the injection of a new and improved service may eventually result in a complete replacement of previous versions—spatially speaking, the region where the new service is active is expected to enlarge until covering the whole system, while the old service diminishes. In the context of visualisation services, for instance, this would amount to the situation where an existing advertisement service is established, but a new one targeted to the same users is injected that happens to have greater use rate, namely, it is more appropriate for the average profile of users: we might expect this new service to overcome the old one, which accordingly extinguishes.

For the sake of explanation, we start from an abstract case, with a reference “random grid” of 30 × 20 nodes (i.e., locations). We call a random grid a lattice-like network in which locations are placed as nodes of a square grid (each non-boundary location has in its proximity 8 nodes, 4 in the horizontal/vertical direction and 4 in the diagonal direction), but connection between a location and a neighbouring one is randomly set (with probability 50%). This choice is motivated by the fact that very often computing devices are placed more or less uniformly over the "space" formed by the buildings, corridors, or rooms of the pervasive computing systems of interest, though randomness is useful to tackle heterogeneity of the environment at hand, failures, and so on.

In the case we consider, in every node requests for using a service are supposed to arrive at a fixed rate for simplicity, and a service called `s1` is the only available to match the requests (we use the following parameters: use rate $u = 0.01$, decay rate $d = 0.01$, request injection rate $p = 50$, moving rate $m = 0.01$). In particular, in every node, the system stabilises approximately to a concentration of $5,000$ `s1` (p/d as usual), in spite of diffusion.

Another service `s2` is at some point developed that can serve the same requests, now with use rate 0.1 instead of 0.05, namely, it is a service developed to more effectively serve requests—it matches requests twice as much as `s1` does. This service is injected into a randomly chosen node of the network, with an initial very low concentration (10 in our experiment). Figure 7.2 shows in each column a different snapshot (from left to right), reporting concentration of `s1` on top and `s2` on bottom: we can observe that `s2` starts diffusing where it is injected, until completely overcoming service `s1` after about $3,000$ time units.

7.4 Case study of long-term competition

To better ground the discussion, and emphasise the adaptive and diversity-accommodation character of our model, we consider some application case for the airport display infrastructure, showing how the proposed reactions would work in a more concrete setting.

7.4.1 Local competition

We first analyse the behaviour of a single display, located near a gate where passengers wait for the departure of their flight. As soon as a passenger gets nearby, her/his preferences are sensed, and become tuples representing requests for a visualisation service—such a sensing might be due to either the passenger's PDA or the passenger's data which are stored in an RFID (or alike) placed on the boarding pass. Visualisation services are continuously injected in the system (in the long-term, they could be many): they are meant to tackle passengers' preferences (e.g. sport, food, tourism, cars) and accordingly compete with one another, since the display is meant to probabilistically select the best service/passenger match.

Figure 7.3 shows all the parameters of the considered simulation scenario, in which 100 visualisation services are injected during a year. Parameters `r_req` and `t_stay` are

Parameter	**Value**	**Description**
`r_req`	$141/137\ min^{-1}$	preference injection rate, as passengers per flight over interval between two flights
`t_stay`	$50\ min$	passenger time nearby the display
`r_ads`	$100\ year^{-1}$	advertisement service injection rate
`t_show`	$30\ sec$	showing time for an advertisement service
`c_ad`	$1,000$	maximum expected concentration of an advertisement service
`r_match`	$1,000\ sec^{-1}$	match rate

Figure 7.3: Airport scenario: competition of services in one display. Simulation parameters.

inferred from Heathrow statistics in 2008[1]. Match rate is the rate at which a single match can be performed: note this is negligible with respect to the time between arrival of two passengers' preferences (namely, about 1 minute due to `r_req`). For the simulation, we used decay rate $d = 1/30,000\ sec^{-1} = 1/(\texttt{c\_ad} \times \texttt{t\_show})$, since the final service concentration `c_ad` has been shown to be p/d, and the pumping rate for services p is 1/`t_show` (service concentration increases by 1 each 30 seconds).

This simulation scenario is relative to a class of advertisement services (e.g., concerning cars), which can match 5 different "marketing targets" (e.g. sport cars, luxury cars, city cars, vans, crossovers). Each passenger is associated to a single marketing target, while each advertisement can cover many marketing targets (e.g. a Ferrari is both a sport and luxury car). Accordingly, each time an advertisement service is created, we randomly draw its match degree with respect to the 5 different marketing targets (a number in between 0 and 1 each), though we keep the sum of such degrees less than the "overlap factor" 1.5 (OF)—to avoid the unrealistic case in which some advertisement perfectly fits all marketing targets (which could happen if $OF = 5$).

Figure 7.4 shows a simulation over a whole year. We note that: *(i)* only few services are actually active at a given time (i.e., they have non-negligible concentration), for the others get extinguished throughout system evolution, and *(ii)* some new service can overcome an existing and established one, causing its extinction (e.g. $s51$ enters the system at day 290, and makes $s89$ extinguishing at day 350)—results of a larger number of simulations show that the average number of active services in the system is about 3.25. At the end of the year, only the following three services are active (reported with their matching degrees):

```
s91[0.52,0.11,0.84,0,0], s20[0.62,0.84,0,0,0], s51[0,0,0,0.75,0.70]
```

Namely, $s91$ is mainly tackling the fourth marketing target (and a good deal of the first), $s20$ is mainly tackling the second marketing target (and a good deal of the first

[1] `http://www.caa.co.uk/`.

as well), while $s51$ mainly tackles the fourth and fifth targets. Of course, in practice such match factors will be computed from semantic matching module, which ranks the extent to which a newly introduced advertisement deals with the 5 marketing targets. By increasing the number of marketing targets and their overlap factor we can deal with more involved situations; for instance, analogous simulations with 20 marketing targets and $OF = 3$ give an average number of 7 visualisation services for the specific class considered. In general, it is predictable that the services that best tackle one ore more marketing targets will survive, while most of the others will end up extinguishing without unnecessarily overloading the system, and with no human intervention. This sort of "ecological" behaviour is typical of today socially situated domains like social networks, and will be likely to play a key role in future pervasive computing systems [1, 87, 101].

7.4.2 Spatial competition

We now analyse a more concrete example, extending the airport scenario studied in previous section to show the spatial character of our framework. Instead of a single display we now consider an airport terminal with 5 gates in a row, and 25 displays near each gate disseminated in the corridor and gate areas. This is modelled as a 25×5 random grid (gates are at coordinates $(3, 3)$, $(8, 3)$, $(13, 3)$, $(18, 3)$, $(23, 3)$).

Advertisement services are now injected from a random node of the network (taken from 8 nodes in the perimeter of the grid considered as entry point nodes), using same dynamics of previous case. Such services diffuse using (DIFFUSE) reaction.

In a scenario in which passenger preferences are uniformly distributed in space and time, we would expect a behaviour similar to that of Figure 7.2, where winning services diffuse in the whole network reaching an uniform value. But in a real-life situation preferences are not uniform but context-dependent, and this influences the actual region in which certain services can actually win competition. As an example, we consider a class of services containing news about specific locations in the world, and each passenger's profile – e.g. automatically extracted from the RFID in the boarding pass – is associated to the preference for just one continent, namely, the one where she/he is flying to. As in the previous case, the overlap factor is 1.5 (in that some news service might span more continents). Now assume each gate hosts flights towards a given continent: this means that each gate is a context where passengers will more likely be interested in news on the corresponding continent. This is obtained by making the injection rate of preferences dependent on the distance from the gate: the higher the distance, the smaller the rate (and still r_req in the node of the gate, as in previous section).

A simulation result is shown in Figure 7.5, which emphasises again the adaptive character of our framework, now also taking into account spatial aspects. Only 4 services are active at the end of the simulation, which are those actually charted. At day 60, $s51$ already established at 2^{nd} gate (from left). At day 167, $s51$ is also establishing on 1^{st} gate, while $s38$ established on gate 4^{th} and 5^{th}. At day 274, $s9$ is appearing on 3^{rd} gate and $s82$ is taking over 1^{st} gate winning competition against $s51$. At the end of simulation, both $s9$ and $s82$ completely established.

Note that in this model, `service` tuples act as a reification of the spatial service state as enacted by the coordination infrastructure: the resulting system features situatedness (success of a service in a location depends on requests and existing services there), adaptivity (the best service actually wins, and unused services fade and get garbage-collected), and accommodation of diversity (the arrival of new services is not foreseen at design time, but automatically managed). In spite the discussed set of chemical reactions appears suitable for the application at hand and its requirements, we believe that different contexts can call for different reactions, without harming the general validity of the proposed model/architecture.

7.5 An architecture based on the TuCSoN infrastructure

In this section we show that an infrastructure for the chemical tuple space model does not have to be necessarily built from scratch, but can be implemented on top of an existing tuple space middleware, such as TuCSoN. In particular, as discussed in [92], TuCSoN supports features that are key for implementing self-organising systems, some of which are here recapped that are useful for implementing the chemical tuple space model:

Topology and Locality Tuple centres can be created locally to a specific node, and the gateway tuple centre can be programmed to keep track of which tuple centres reside in the neighbourhood—accessible either by agents or by tuple centres in current node.

On-line character and Time TuCSoN supports the so called "on-line coordination services", executed by reactions that are fired in the background of normal agent interactions, through timed reactions—reactions whose event `E` is of kind `time(T)`. When the tuple centre time (expressed as Java milliseconds) reaches `T`, the corresponding reaction is fired. Moreover, a reaction goal can be of the kind `out_s (reaction(time(T),G,R))`, which inserts tuple `reaction(time(T),G,R)` in the space, thus triggering a new reaction. As a simple example, the following reactions are used to insert a tuple `tick` in the space each second:

```
reaction( time(T), endo,   ( out(tick) )).
reaction( out(tick), endo, ( currentTime(T), NewT is T+1000,
                    out_s(reaction( time(NewT), endo, out(tick))) )).
```

Probability Probability is a key feature of self-organisation, which is necessary to abstractly deal with the unpredictability of contingencies in pervasive computing. In

TuCSoN this is supported by drawing random numbers and using them to drive the reaction firing process, that is, making tuple transformation be intrinsically probabilistic. For instance, the following reaction inserts either tuple `head` or `tail` in the space (with 50% probability):

```
reaction( out(draw), from-agent, (X > 0.5, out(head)) ; out(tail)  )).
```

In the following, it will be described how the two basic additional ingredients of the proposed model can be supported on top of TuCSoN: semantic matching (Section 7.5.1) and chemical engine (Section 7.5.2).

7.5.1 Semantic Matching

As shown in Chapter 4, in order to enable semantic support in TuCSoN, a tuple centre has to be related to an ontology, to which semantic tuples refer to. In order to encapsulate an ontology, tuple centres exploit the aforementioned Pellet reasoner [82]—an open-source DL reasoner based on OWL and written in Java likewise TuCSoN. In particular, Pellet can load an OWL TBox and an ABox, and provides the *Jena-API* in order to add and remove individuals by its own ABox. Hence, each semantic tuple is carried not only in the tuple space likewise syntactic tuples, but also in the ontology ABox, in order to support reasoning. The reasoner is internally called each time we are checking for a semantic match: the semantic template is converted into a SPARQL query (the language used by Jena-API) and the results obtained by the reasoner (the name of individuals) are used to retrieve the actual tuples in the tuple space. This behaviour is embedded in the tuple centre, such that each time a semantic template is specified into a retrieval operation, *any* semantic tuple can actually be returned—namely, the standard behaviour is still non-deterministic. Additionally, in order to support probabilistic retrieval, such as in the case of our chemical model, it is possible to exploit fuzzy matching as shown in Section 6.3. In particular, this is achieved via predicate the fuzzy primitive *in* like `in(semantic (X degree Y matching ('CityCar' and (exists hasMaker : fiat))))`, taking the semantic template as an input, providing as an output a matching semantic tuple and the corresponding match degree (a number in between 0 and 1).

7.5.2 Chemical Reactions

We now describe how a TuCSoN tuple centre can be specialised to act as a chemical-like system where semantic tuples play the role of reactants, which combine and transform over time as occurring in chemistry.

Coding reactants and laws

Tuples modelling reactant individuals are kept in the tuple space in the form `reactant(X,N)`, where `X` is a semantic tuple representing the reactant and `N` is a natural number denoting concentration. Laws modelling chemical reactions are expressed by tuples of the form `law(InputList,Rate,OutputList)`, where `InputList` denotes the list of the reacting individuals, and `Rate` is a float value specifying the constant rate of the reaction. As a reference example, consider the chemical laws resembling (USE,DECAY) rules in previous section: $S + R \xrightarrow{10.0} S + S$ and $S \xrightarrow{10} 0$, where S and R represent semantic templates for services and requests. Such laws can be expressed in TuCSoN by tuples `law([S,R],10,[S,S])` and `law([S],10,[])`. On the other hand, tuples `reactant(sa,1000)`, `reactant(sb,1000)` and `reactant(r,1000)` represent reactants for two semantic tuples (`sa` and `sb`) matching `S`, and one (`ra`) matching `R`. The set of enabled laws at a given time is conceptually obtained by instantiating the semantic templates in reactions with all the available semantic tuples. In the above case they would be `law([sa,r],r1a,[sa,sa])`, `law([sb,r],r1b,[sb,sb])`, `law([sa],r2a,[])` and `law([sb],r2b,[])`. The rate of each enabled reaction (usually referred to as *global rate*) is obtained as the product of chemical rate and match degree as described in Section 7.3. For instance, rate `r1a` can be calculated as $10.0 \times \#\mathtt{sa} \times \#\mathtt{r} \times \mu(\mathtt{S}+\mathtt{R}, \mathtt{sa}+\mathtt{r})$, where μ is the function returning the match factor between the list of semantic reactants and the list of actual tuples.

As an additional kind of chemical law, it is also possible to specify a transfer of molecules towards other tuple centres by a law of the kind `law([X],10,[firing(X)])`.

ReSpecT engine

The actual chemical engine is defined in terms of ReSpecT reactions, which can be classified according to the provided functionality. As such, there are reactions for *(i)* managing chemical laws and reactants, i.e. ruling the dynamic insertion/removal of reactants and laws, *(ii)* controlling engine start and stop, *(iii)* choosing the next chemical law to be executed, and *(iv)* executing chemical laws. For the sake of conciseness we only describe part (iii), which is the one that focusses on Gillespie's algorithm for chemical simulations [38].

This computes the choice of the next chemical law to be executed, based on the following ReSpecT reaction, triggered by operation `out(engine_trigger)` which starts the engine.

```
reaction( out(engine_trigger), endo, (
          in(engine_trigger),
          chooseLaw(law(IL,_,OL),Rtot),
          rand_float(Tau), Dt is round((log(1.0/Tau)/Rtot)*1000),
          event_time(Time), Time2 is Time + Dt,
          out_s(reaction( time(Time2), endo, out(engine_trigger) )),
          out(execution(law(IL,_,OL),Time))
)).
```

First of all, a new law is chosen by the `chooseLaw` predicate, which returns `Rtot`, the global rate of all the enabled chemical laws, and a term `law(IL,_,OL)`—`IL` and `OL` are bound respectively to the list of reactants and products in the chosen law, after templates are instantiated to tuples as described above. Then, according to Gillespie algorithm, time interval `Dt` – denoting the overall duration of the chemical reaction – is stochastically calculated (in milliseconds) as $log(1/Tau)/Rtot$, where Tau is a value randomly chosen between 0 and 1 [38]. A new timed reaction is accordingly added to the ReSpecT specification and will be scheduled for execution `Dt` milliseconds later with respect to `Time`, which is the time at which `out(engine_trigger)` occurred: the corresponding reaction execution will result in a new `out(engine_trigger)` operation that keeps the chemical engine running. Finally, a new tuple `execution(law(IL,_,OL),Time)` is inserted so that the set of reactions devoted to chemical-law execution can be activated.

The actual implementation of the Gillespie's algorithm regarding the choice of the chemical law to be executed is embedded in the `chooseLaw` predicate, whose implementation is as follows:

```
chooseLaw(Law,Rtot):-
          rd(laws(LL)),
          semanticMatchAll(LL,NL,Rtot), not(Rtot==0),
          sortLaws(NL,SL),
          rand_float(Tau),
          chooseGillespie(SL,Rtot,Tau,Law).
```

After retrieving the list `LL` of the chemical laws defined for the tuple centre, `semanticMatchAll` returns the list `NL` of enabled chemical laws and the corresponding overall rate `Rtot`, computed as the sum of the global rate of every enabled law. To this end, predicate `semanticMatchAll` relies on predicate `retrieve(+SemanticTemplateList, -SemanticTupleList,-MatchFactor)` already described (properly extended to deal with lists of semantic tuples and templates).

The chemical law to be executed is actually chosen via the `chooseGillespie` predicate if `Rtot` > 0, i.e. if there are enabled chemical laws. This choice is driven by a probabilistic process: given n chemical laws and their global rates $r_1, ..., r_n$, the probability for law i to be chosen is defined as r_i/R, where $R = \sum_i r_i$. Consequently, law selection is simply driven by drawing a random number between 0 and 1 and choosing a law according to the probability distribution of the enabled laws.

7.6 Summary

In this chapter we described research and development challenges in the implementation of the chemical tuple space model in semantic TuCSoN. These are routed in two basic dimensions, which are mostly – but not entirely – orthogonal. On the one hand, the basic tuple centre model is to be extended to handle semantic matching, which we support by the following ingredients: *(i)* an OWL ontology (a set of definitions of concepts) stored into a tuple centre which grounds semantic matching; *(ii)* tuples (other than syntactic as usual) can be semantic, describing an individual of the application domain (along with the concept it belongs to and the individuals it is linked to through roles); and *(iii)* a matching function implemented so as to check whether a tuple is the instance of a concept, returning the corresponding match factor. On the other hand, the coordination specification for the tuple centre should act as a sort of "online chemical simulator", evolving the concentration of tuples over time using the same stochastic model of chemistry [38], so as to reuse existing natural and artificial chemical systems (like prey-predator equations); at each step of the process: *(i)* the reaction rates of all the chemical laws are computed, *(ii)* one is probabilistically selected and then executed, *(iii)* the next step of the process is triggered after an exponentially distributed time interval, according to the Markov property. The path towards a fully featured and working infrastructure has been paved, but further research and development is required to tune several aspects:

Match degree Implementing fuzzy matching techniques as shown in Section 6.3.

Performance The problem of performance was not considered yet, but will be subject of our future investigation. Possible bottlenecks include the chemical model and its implementation as a ReSpecT program, but also semantic retrieval, which is seemingly slower than standard syntactic one. We still observe that in many scenarios of pervasive computing – like those considered in Section 7.4 – this is not a key issue.

Chemical language Developing a suitable language for semantic chemical laws is a rather challenging issue. The design described in this paper supports limited forms of service interactions that will likely be extended in the future. For instance, a general law $X+Y \rightarrow Z$ is meant to combine two individuals into a new one, hence the chemical language should be able to express into Z *how* the semantic templates X and Y should combine—aggregation, contextualisation, and other related patterns of self-organising pervasive systems are to be handled at this level.

Application cases The model, and correspondingly the implementation of the infrastructure, are necessarily to be tuned after evaluation of selected use cases can be performed. Accordingly, the current version of the infrastructure is meant to be a prototype over which initial sperimentation can be performed. A main application scenario we will considered for actual implementation is a general purpose pervasive display infrastructure.

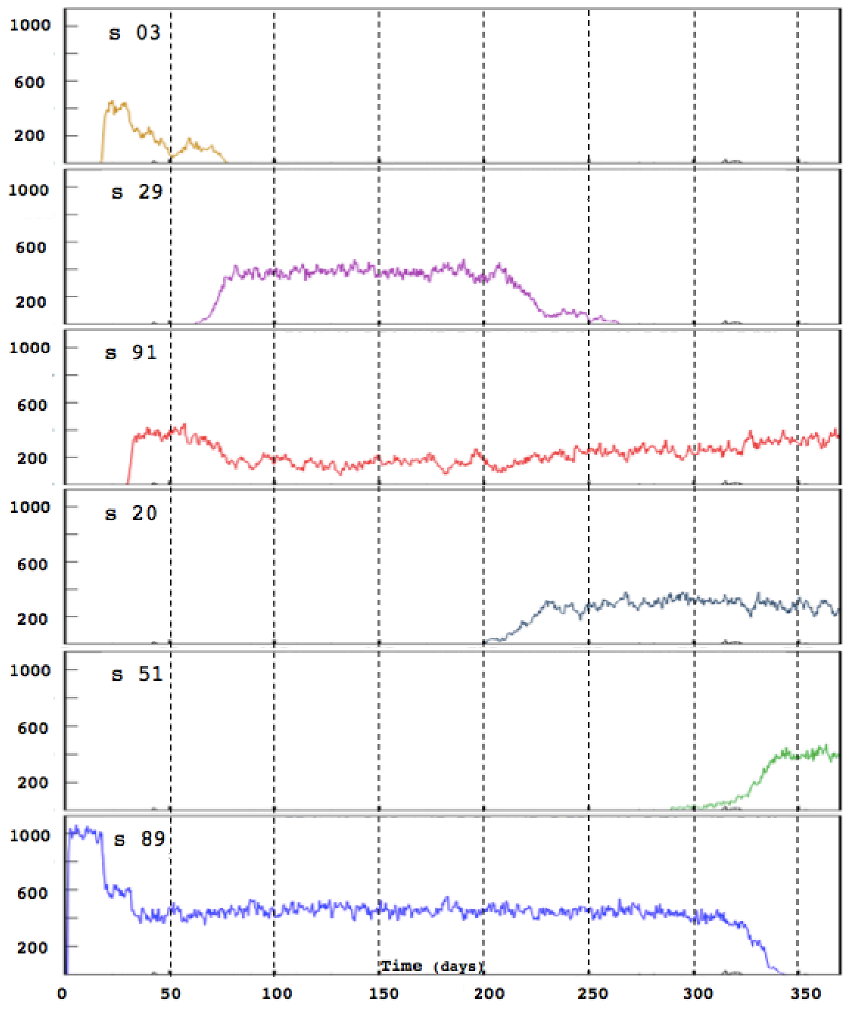


Figure 7.4: Airport scenario: competition of services in one display. Charting concentration of the 6 services active throughout the simulation.

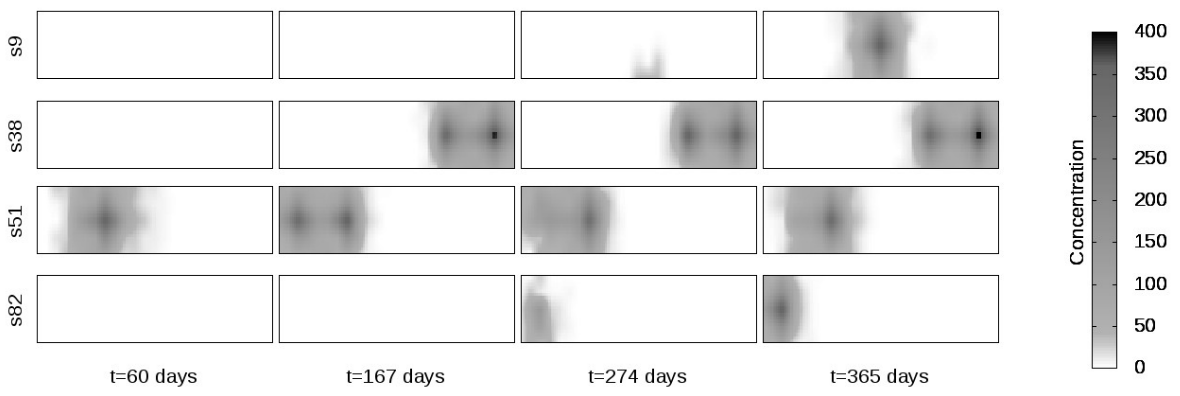


Figure 7.5: Airport scenario: competition of services in the terminal. Showing spatial concentration of the 4 surviving services, in 4 snapshots.

Chapter 8

Conclusion

This chapter summarises the work presented in the thesis, highlighting the corresponding contributions and then tracing feasible future works.

8.1 Summary of the Contributions

The work of the thesis aimed to model and implement semantic tuple centres—a basic brick of coordination infrastructures for open, distributed and knowledge-intensive systems like Web-based and pervasive computing applications. In particular, this model supports the novel notion of programming the semantic coordination aspects of a complex system, a key notion in the coordination of self-organising systems [69]. The thesis started with a detailed analysis of related works concerning tuple-space-based models. Then, the thesis provided a deeper description of the semantic tuple centres providing its abstract architecture. It was described an implementation of the model based on the TuCSoN coordination infrastructure and it was evaluated on its current performance showing its applicability in real application contexts. In particular, our analysis of performance showed that, while tuple insertion is typically quite fast, tuple retrieval can be an heavy operation especially when many tuples occur in the tuple space, which is basically due to the latency of instance checking by the semantic reasoner. Additionally, it was shown the important role of the fuzziness in describing the knowledge stored in tuple centres in open context and an extension of the semantic tuple centre model and of the implementation of the new model in TuCSoN. Finally, it was shown two main important application scenarios in which semantic tuple centres seems to represent a suitable coordination media. The first application scenarios is in the context of e-Health, that is, Healthcare supported by software systems. In particular we will focus on the interoperability of Electronic Health Record (EHR) fragments – medical information that are stored in a digital format over different healthcare institutions – belonging to an environment that is *distributed* and *open* and where the *security support* represents a fundamental requirement to protect the patient privacy. We will show how it is possible to extend the solutions proposed in literature by tacking the semantic version of TuCSoN as inspiration model, in order to augment their effectiveness in building EHR services, in particular as far as interoperability is concerned.

The second application scenario is in the context of the pervasive services. Addressing this scenario calls for finding infrastructures promoting a concept of pervasive "eternality", namely, changes in topology, device technology and continuous injection of new services have to be dynamically tolerated as much as possible, and incorporated with no significant re-engineering costs at the middleware level [101, 94]. As far as the coordination of such services is concerned, it will increasingly be required to tackle self-organisation (supporting situatedness, adaptivity and long-term accommodation of diversity) as an inherent system property rather than a peculiar aspect of the individual coordinated components. As typical in self-organising computational mechanisms, a promising direction is to take inspiration from natural systems (e.g. physical, chemical, biological, social [94]), where self-organisation is intrinsic to the basic "rules of the game". Focussing on chemical natural systems, we will first shows the concept of chemical tuple spaces [91] – tuple spaces programmed with coordination rules resembling chemical reactions – as suitable coordination media for situated and adaptive pervasive computing [77, 17, 46, 56]. Then, we will show how a distributed architecture for chemical tuple spaces [91] can be implemented in TuCSoN providing semantic ReSpecT tuple centres.

8.2 Future Directions

Starting from the work done so far, a first direction for future investigations is hence on the optimisation side, evaluating other semantic approaches (or reasoners) that can trade-off speed for expressiveness. From the model viewpoint, a basic extension we are evaluating concerns the introduction of fuzzyness, relying on approaches like fuzzyDL [12], which would allow us to rank the degree by which a tuple matches a given template—a feature that is particularly useful in open knowledge-intensive systems. Finally, the model is particularly suitable for complex application scenarios like self-organising pervasive computing domains – like e.g. pervasive and wearable displays [31] – where the coordination rules regulating component interactions are required to semantically apply to proper annotations (i.e. tuples) capturing the occurrence and state of components in the shared space formed by pervasive devices [62].

Bibliography

[1] Gul Agha. Computing in pervasive cyberspace. *Commun. ACM*, 51(1):68–70, 2008.

[2] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[3] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 3rd edition, 2010.

[4] Davide Balzarotti, Paolo Costa, and Gian Pietro Picco. The LighTS tuple space framework and its customization for context-aware applications. *Web intelligence and Agent Systems*, 5(2):215–231, 2007.

[5] Ayomi Bandara, Terry R. Payne, David De Roure, Nicholas Gibbins, and Tim Lewis. A pragmatic approach for the semantic description and matching of pervasive resources. In *Advances in Grid and Pervasive Computing*, volume 5036 of *LNCS*, pages 434–446. Springer, 2008.

[6] G.O. Barnett and H.J. Sukenik. Hospital Information Systems. In J.F. Dickson and J.H.U. Brown, editors, *Future Goals of Engineering in Biology and Medicine*. Academic Press, 1969.

[7] Alistair P. Barros and Marlon Dumas. The rise of web service ecosystems. *IT Professional*, 8(5):31–37, 2006.

[8] T. Beale. 2001.

[9] Alan A. Berryman. The origins and evolution of predator-prey theory. *Ecology*, 73(5):1530–1535, October 1992.

[10] Fernando Bobillo, Miguel Delgado, and Juan Gómez-Romero. A crisp representation for fuzzy SHOIN with fuzzy nominals and general concept inclusions. In Paulo Cesar G. da Costa, Claudia d'Amato, Nicola Fanizzi, Kathryn B. Laskey, Kenneth J. Laskey, Thomas Lukasiewicz, Matthias Nickles, and Michael Pool, editors, *Uncertainty Reasoning for the Semantic Web I*, volume 5327 of *LNCS*, pages 174–178. Springer, 2008.

[11] Fernando Bobillo, Miguel Delgado, and Juan Gomez-Romero. Delorean: A reasoner for fuzzy OWL 1.1. In Fernando Bobillo, Paulo Cesar G. da Costa, Claudia d'Amato, Nicola Fanizzi, Kathryn B. Laskey, Kenneth J. Laskey, Thomas Lukasiewicz, Trevor P. Martin, Matthias Nickles, Michael Pool, and Pavel Smrz, editors, *4th International Workshop on Uncertainty Reasoning for the Semantic Web (URSW 2008)*, volume 423 of *CEUR Workshop Proceedings*, Karlsruhe, Germany, October 2008. Sun SITE Central Europe, RWTH Aachen University.

[12] Fernando Bobillo and Umberto Straccia. FuzzyDL: An expressive fuzzy description logic reasoner. In *2008 International Conference on Fuzzy Systems (FUZZ-IEEE 2008)*, pages 923–930. IEEE CS, 2008.

[13] Jrgen Bock, Peter Haase, Qiu Ji, and Raphael Volz. Benchmarking OWL reasoners. In Frank van Harmelen, Andreas Herzig, Pascal Hitzler, Zuoquan Lin, Ruzica Piskac, and Guilin Qi, editors, *Workshop on Advancing Reasoning on the Web: Scalability and Commonsense (ARea2008)*, volume 350 of *CEUR Workshop Proceedings*, ESWC 2008, Tenerife, Spain, 2 June 2008.

[14] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, July/August 2000.

[15] Scott Camazine, Jean-Louis Deneubourg, Nigel R. Franks, James Sneyd, Guy Theraulaz, and Eric Bonabeau. *Self-Organization in Biological Systems.* Princeton Studies in Complexity. Princeton University Press, Princeton, NJ, USA, 2001.

[16] D. Cerizza, E. Della Valle, D. Foxvog, R. Krummenacher, and M. Murth. Towards European Patient Summaries based on Triple Space Computing. In *ECEH 2006*, 2006.

[17] Patricia Dockhorn Costa, Giancarlo Guizzardi, João Paulo A. Almeida, Luís Ferreira Pires, and Marten van Sinderen. Situations in conceptual modeling of context. In *Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006), 16-20 October 2006, Hong Kong, China, Workshops*, page 6. IEEE Computer Society, 2006.

[18] Marco Cremonini, Andrea Omicini, and Franco Zambonelli. Multi-agent systems on the Internet: Extending the scope of coordination towards security and topology. In Francisco J. Garijo and Magnus Boman, editors, *Multi-Agent Systems Engineering*, volume 1647 of *LNAI*, pages 77–88. Springer, 1999.

[19] J. Denny, D. Giuse, and J. Jirjis. The Vanderbilt Experience with Electronic Health Records. *Seminars in Colon and Rectal Surgery*, 12:59–68, 2005.

[20] Enrico Denti, Antonio Natali, and Andrea Omicini. Programmable coordination media. In David Garlan and Daniel Le Métayer, editors, *Coordination Languages and Models*, volume 1282 of *LNCS*, pages 274–288. Springer-Verlag, 1997. 2nd International Conference (COORDINATION'97), Berlin, Germany, 1–3 September 1997. Proceedings.

[21] Enrico Denti, Antonio Natali, and Andrea Omicini. On the expressive power of a language for programming coordination media. In *1998 ACM Symposium on Applied Computing (SAC'98)*, pages 169–177, Atlanta, GA, USA, 27 February – 1 March 1998. ACM. Special Track on Coordination Models, Languages and Applications.

[22] Enrico Denti, Andrea Omicini, and Alessandro Ricci. tuProlog: A light-weight Prolog for Internet applications and infrastructures. In I.V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *LNCS*, pages 184–198. Springer, 2001. 3rd International Symposium (PADL 2001), Las Vegas, NV, USA, 11–12 March 2001. Proceedings.

[23] Enrico Denti, Andrea Omicini, and Alessandro Ricci. Coordination tools for MAS development and deployment. *Applied Artificial Intelligence*, 16(9/10):721–752, October/December 2002.

[24] R.S. Dick and E.B. Steens. *The Computer-Based Patient Record: An Essential Technology for Health Care.* Institute of Medicine, National Academic Press, 1991.

[25] R.H. Dolin, L. Alschuler, S. Boyer, and C. Beebe. HL7 Clinical Document Architecture, Release 2.0. 2004.

[26] Y. Ducq, D. Chen, and B. Vallespir. Interoperability in enterprise modelling: Requirements and roadmap. *Advanced Engineering Informatics*, 18:193–203, 2004.

[27] Ltd. eHealth Media. OpenEHR Foundation launches international standard.

[28] B. S. Elgera, J. Iavindrasana, L. Lo Iacono, H. Müller, N. Roduit, Paul Summers, and J. Wright. Strategies for health data exchange for secondary, cross-institutional clinical research. *Computer Methods and Programs in Biomedicine*, 99:230–251, 2010.

[29] Dieter Fensel. Triple-space computing: Semantic web services based on persistent publication of information. In Finn Arve Aagesen, Chutiporn Anutariya, and Vilas Wuwongse, editors, *Intelligence in Communication Systems*, volume 3283 of *LNCS*, pages 43–53, 2004. IFIP International Conference (INTELLCOMM 2004), Bangkok, Thailand, 23–26 November 2004. Proceedings.

[30] Alois Ferscha, Andreas Riener, Manfred Hechinger, and Heinrich Schmitzberger. Building pervasive display landscapes with stick-on interfaces. In *CHI Workshop on Information Visualization and Interaction Techniques*, April 2006.

[31] Alois Ferscha and Simon Vogl. Wearable displays – for everyone! *IEEE Pervasive Computing*, 9(1):7–10, January/March 2010.

[32] European Committee for Standardization. Health informatics Electronic health record communication Part 1: Reference model Draft European Standard for CEN Enquiry prEN 13606-1. 2004.

[33] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice: Principles, Patterns and Practices.* The Jini Technology Series. Addison-Wesley Longman, June 1999.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: abstraction and reuse of object-oriented design*, pages 701–717. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[35] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[36] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[37] GigaSpaces. Home page.

[38] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[39] Volker Haarslev and Ralf Möller. RACER system description. In Rajeev Goré, Alexander Leitsch, and Tobias Nipkow, editors, *1st International Joint Conference on Automated Reasoning (IJCAR '01)*, volume 2083 of *LNCS*, pages 701–705. Springer, Siena, Italy, 18–23 June 2001.

[40] W. Hasselbring and S. Pedersen. Metamodelling of Domain-Specific Standards for Semantic Interoperability. In J.F. Dickson and J.H.U. Brown, editors, *WM 2005*. Academic Press, 2005.

[41] Ian Horrocks, Peter F. Patel-Schneider, and Frank Van Harmelen. From shiq and rdf to owl: The making of a web ontology language. *Journal of Web Semantics*, 1, 2003.

[42] Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: The making of a Web ontology language. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):7–26, December 2003.

[43] IHE International. Integration Profiles: IHE IT Infrastructure Technical Framework. 2009.

[44] M.D. James and D. Thomas. Thomas The DICOM image formatting standard: What it means for echocardiographers. *Journal of the American Society of Echocardiography*, 8:319–327, 1995.

[45] Nicholas R. Jennings. An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41, 2001.

[46] Christine Julien and Gruia-Catalin Roman. Egospaces: Facilitating rapid development of context-aware mobile applications. *IEEE Trans. Software Eng.*, 32(5):281–298, 2006.

[47] Christine Julien and Gruia-Catalin Roman. EgoSpaces: Facilitating rapid development of context-aware mobile applications. *IEEE Transactions on Software Engineering*, 32(5):281–298, May 2006.

[48] D. Kalra. Electronic health record standards. *IMIA Yearbook of Medical Informatics*, pages 150–161, 2006.

[49] L.T. Khon, J.M. Corrigan, and M.S Donaldson. *To Err is Human: building a safer health system.* National Academy Press, 2000.

[50] Deepali Khushraj, Ora Lassila, and Timothy W. Finin. sTuples: Semantic tuple spaces. In Timothy W. Finin, Chiara Ghidini, Tom La Porta, and Chiara Petrioli, editors, *1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04)*, pages 268–277, Boston, MA, USA, 22–26 August 2004.

[51] G.J. Klir and B. Yuan. *Fuzzy sets and fuzzy logic: theory and applications.* Prentice-Hall, Inc., 1994.

[52] C. Lovis, S. Spahni, C. Cassoni, and A. Geissbuhler. Comprehensive management of the access to the electronic patient record: towards trans-institutional network. *International Journal of Medical Informatics*, 76:466–470, 2006.

[53] Thomas Lukasiewicz and Umberto Straccia. Managing uncertainty and vagueness in Description Logics for the Semantic Web. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(4):291–308, October 2008.

[54] K. Malloch. The electronic health record: An essential tool for advancing patient safety. *Nursing Outlook*, 55:150–161, 2007.

[55] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications with the TOTA middleware. In *Pervasive Computing and Communications*, pages 263–273, 2004. 2nd IEEE Annual Conference (PerCom 2004), Orlando, FL, USA, 14–17 March 2004. Proceedings.

[56] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: the TOTA approach. *ACM Trans. Software Engineering and Methodology*, 18(4), 2009.

[57] Francisco Martín-Recuerda. Towards Cspaces: A new perspective for the Semantic Web. In Max Bramer and Vagan Terziyan, editors, *Industrial Applications of Semantic Web*, volume 188, pages 113–139. Springer, 2005. 1st IFIP WG12.5 Working Conference on Industrial Applications of Semantic Web, 25–27 August 2005, Jyväskylä, Finland. Proceedings.

[58] Naftaly H. Minsky and Jerrold Leichter. Law-Governed Linda as a coordination model. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-based Models and Languages for Concurrent Systems*, volume 924 of *LNCS*, pages 125–146. Springer, 1995.

[59] Boris Motik and Ulrike Sattler. A comparison of reasoning techniques for querying large Description Logic ABoxes. In Andrei Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 4246 of *LNCS*, pages 227–241. Springer, 2006. 13th International Conference (LPAR 2006), Phnom Penh, Cambodia, 13-17 November, 2006. Proceedings.

[60] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A coordination model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology*, 15(3):279–328, July 2006.

[61] Elena Nardini, Andrea Omicini, and Mirko Viroli. General-purpose coordination abstractions for managing interaction in MAS. In *The WI-IAT 2009 Workshops Proceedings*, pages 501–506, Milano, Italy, 15–18 September 2009. IEEE Computer Society. 2nd Workshop on Logics for Intelligent Agents and Multi-Agent Systems (WLIAMAS 2009), 2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT'09).

[62] 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.

[63] Elena Nardini, Mirko Viroli, and Emanuele Panzavolta. Coordination in open and dynamic environments with TuCSoN semantic tuple centres. 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 2037–2044, Sierre, Switzerland, 22–26 March 2010. ACM.

[64] Lyndon J. B. Nixon, Elena Simperl, Reto Krummenacher, and Francisco Martín-Recuerda. Tuplespace-based computing for the Semantic Web: A survey of the state-of-the-art. *The Knowledge Engineering Review*, 23(2):181–212, 2008.

[65] Andrea Omicini. Towards a notion of agent coordination context. In Dan C. Marinescu and Craig Lee, editors, *Process Coordination and Ubiquitous Computing*, chapter 12, pages 187–200. CRC Press, Boca Raton, FL, USA, October 2002.

[66] Andrea Omicini. Formal ReSpecT in the A&A perspective. *Electronic Notes in Theoretical Computer Sciences*, 175(2):97–117, June 2007. 5th International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'06), CONCUR'06, Bonn, Germany, 31 August 2006. Post-proceedings.

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

[68] Andrea Omicini and Alessandro Ricci. MAS organisation within a coordination infrastructure: Experiments in TuCSoN. In Andrea Omicini, Paolo Petta, and Jeremy Pitt, editors, *Engineering Societies in the Agents World IV*, volume 3071 of *LNAI*, pages 200–217. Springer-Verlag, June 2004. 4th International Workshop (ESAW 2003), London, UK, 29–31 October 2003. Revised Selected and Invited Papers.

[69] Andrea Omicini and Mirko Viroli. Coordination models and languages: From parallel computing to self-organisation. *The Knowledge Engineering Review*, 26(1):53–59, March 2011. Special issue for the 25th Years of the Knowledge Engineering Review.

[70] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999. Special issue: Coordination Mechanisms for Web Agents.

[71] J. M. Overhage, L. Evans, and J. Marchibroda. *Journal of the American Medical Informatics Association*, 12:107–112, 2005.

[72] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Semantic matching of web services capabilities. In *International Semantic Web Conference*, volume 2342 of *LNCS*, pages 333–347. Springer, 2002.

[73] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):16:1–16:45, September 2009.

[74] Alessandro Ricci and Andrea Omicini. Supporting coordination in open computational systems with TuCSoN. In *IEEE 12th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2003)*, pages 365–370, 1st International Workshop "Theory and Practice of Open Computational Systems" (TAPOCS 2003), Linz, Austria, 9–11 June 2003. IEEE CS. Proceedings.

[75] Alessandro Ricci, Andrea Omicini, and Enrico Denti. Virtual enterprises and workflow management as agent coordination issues. *International Journal of Cooperative Information Systems*, 11(3/4):355–379, September/December 2002.

[76] Ana Paula Rocha and Eugenio Oliveira. An electronic market architecture for the formation of virtual enterprises. In *IFIP TC5 WG5.3 / PRODNET Working Conference on Infrastructures for Virtual Enterprises: Networking Industrial Enterprises*, volume 153 of *IFIP Conference Proceedings*, pages 421–432. Kluwer, B.V., 1999.

[77] Manuel Román, Christopher K. Hess, Renato Cerqueira, Anand Ranganathan, Roy H. Campbell, and Klara Nahrstedt. Gaia: a middleware platform for active spaces. *Mobile Computing and Communications Review*, 6(4):65–67, 2002.

[78] C. Safran, D.Z. Sands, and D.M. Rind. Online medical records: a decade of experience. *Methods Inf Med*, 38:308–312, 2000.

[79] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *Computer*, 29(2):38–47, Feb 1996.

[80] D.Z. Sands, D.M. Rind, C. Vieira, and C. Safran. Can a large institution go paperless? In *MEDINFO*, 1998.

[81] Rob Shearer, Boris Motik, and Ian Horrocks. HermiT: A highly-efficient OWL reasoner. In Catherine Dolbear, Alan Ruttenberg, and Uli Sattler, editors, *5th International Workshop "OWL: Experiences and Directions" (OWLED 2008)*, volume 432 of *CEUR Workshop Proceedings*, ISWC 2008, Karlsruhe, Germany, 26–27 October 2008.

[82] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, June 2007.

[83] Umberto Straccia. Reasoning within fuzzy description logics. *Journal of Artificial Intelligence Research*, 14(1):137–166, January 2001.

[84] Umberto Straccia. Description logics with fuzzy concrete domains. In Fahiem Bachus and Tommi Jaakkola, editors, *21st Conference on Uncertainty in Artificial Intelligence (UAI-05)*, pages 559–567, Edinburgh, Scotland, 2005. AUAI Press.

[85] Robert Tolksdorf, Lyndon J. B. Nixon, and Elena Simperl. Towards a tuplespace-based middleware for the Semantic Web. *Web Intelligence and Agent Systems*, 6(3):235–251, 2008.

[86] Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: System description. In Ulrich Furbach and Natarajan Shankar, editors, *3rd International Joint Conference on Automated Reasoning (IJCAR '06)*, volume 4130 of *LNCS*, pages 292–297. Springer, Seattle, WA, USA, 17–20 august edition, 2006.

[87] Mihaela Ulieru and Steve Grobbelaar. Engineering industrial ecosystems in a networked world. In *5th IEEE International Conference on Industrial Informatics*, pages 1–7. IEEE Press, June 2007.

[88] W.J. van der Kam, P. W. Moormanb, and M.J. Koppejan-Mulder. Effects of electronic communication in general practice. *International Journal of Medical Informatics*, 60:59–70, 2000.

[89] J. Van der Lei, M.A. Musen, E. van der Does, A. Main in't Veld, and J.H. van Bemmel. Review of physician decision making using data from computer-stored medical records. *The Lancet*, 338:1504–1508, 1991.

[90] U. Varshney. *Pervasive Healthcare Computing.* Springer, 2009.

[91] 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-Verlag, June 2009. 11th International Conference (COORDINATION 2009), Lisbon, Portugal, June 2009. Proceedings.

[92] Mirko Viroli, Matteo Casadei, and Andrea Omicini. A framework for modelling and implementing self-organising coordination. In *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, volume III, pages 1353–1360. ACM, 8–12 March 2009.

[93] Mirko Viroli, Andrea Omicini, and Alessandro Ricci. Infrastructure for RBAC-MAS: An approach based on Agent Coordination Contexts. *Applied Artificial Intelligence*, 21(4–5):443–467, April 2007. Special Issue: State of Applications in AI Research from AI*IA 2005.

[94] Mirko Viroli and Franco Zambonelli. A biochemical approach to adaptive service ecosystems. *Information Sciences*, 180(10):1876–1892, 2010.

[95] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, June 1995.

[96] Peter Wyckoff, Stephen W. McLaughry, Tobin J. Lehman, and Daniel A. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[97] XMLSpaces. Home page.

[98] John Yen. Generalizing term subsumption languages to fuzzy logic. In *12th International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 472–477, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[99] Lotfy A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.

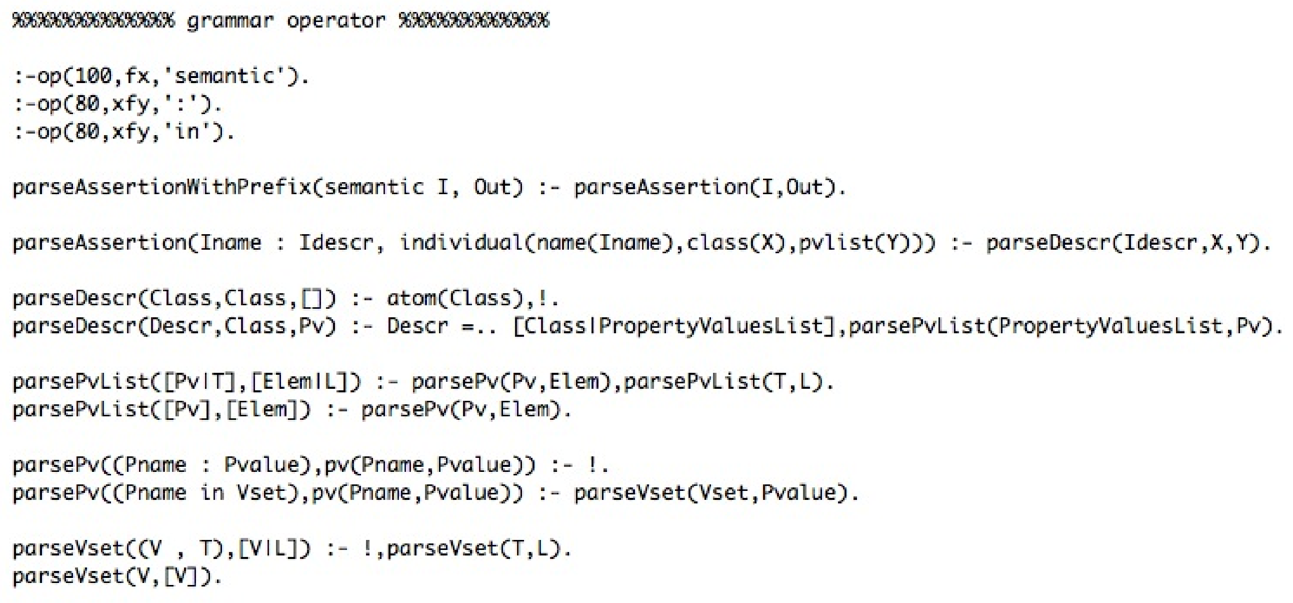
[100] Franco Zambonelli and H. Van Dyke Parunak. Towards a paradigm change in computer science and software engineering: A synthesis. *The Knowledge Engineering Review*, 18(4):329–342, December 2003.

[101] Franco Zambonelli and Mirko Viroli. Architecture and metaphors for eternally adaptive service ecosystems. In *IDC'08*, volume 162/2008 of *Studies in Computational Intelligence*, pages 23–32. Springer Berlin / Heidelberg, September 2008.

Appendix A

Prolog Theories for the Semantic TuCSoN Implementation

A.1 The ASSERTION Prolog Theory

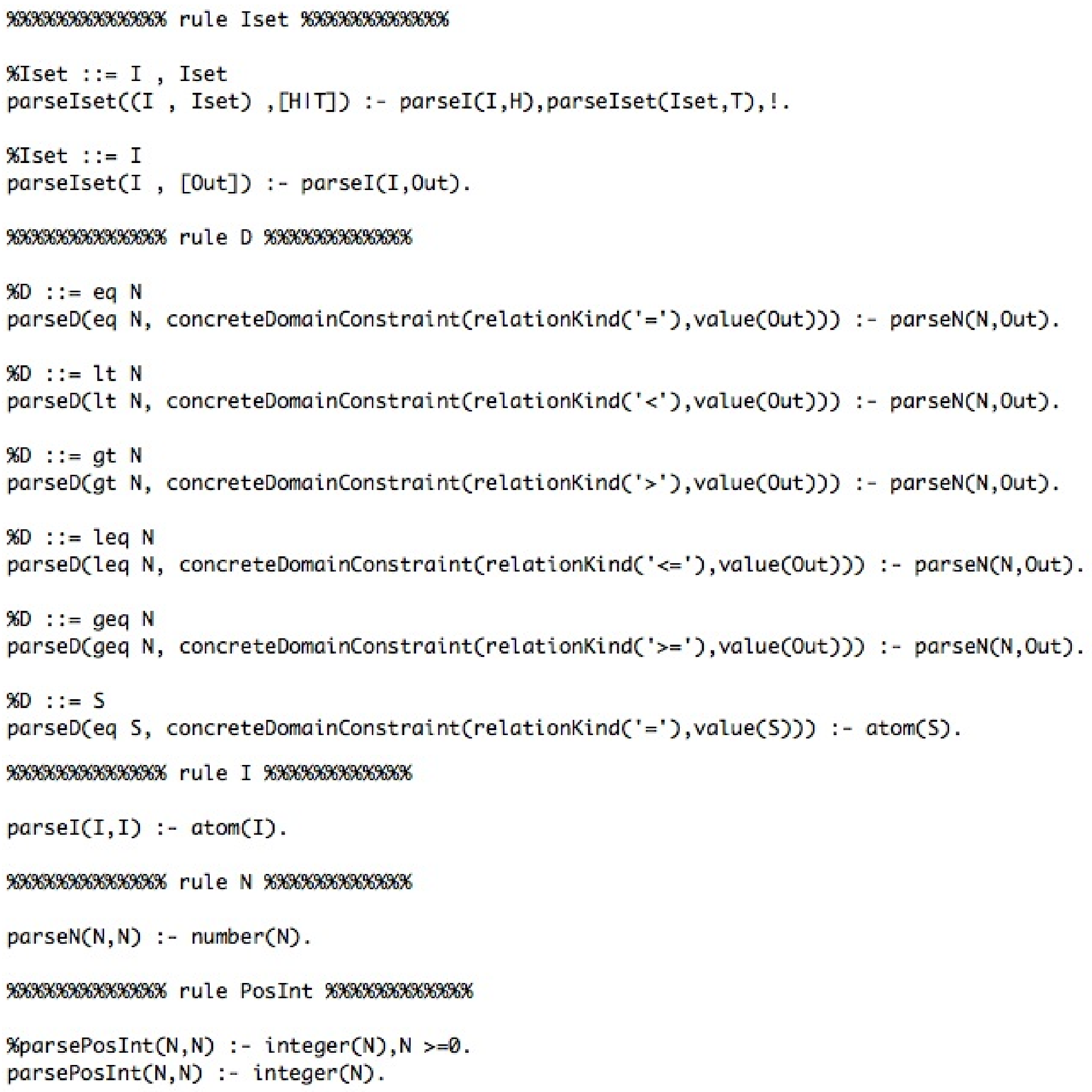


```
%%%%%%%%%%%%% grammar operator %%%%%%%%%%%%%

:-op(100,fx,'semantic').
:-op(80,xfy,':').
:-op(80,xfy,'in').

parseAssertionWithPrefix(semantic I, Out) :- parseAssertion(I,Out).

parseAssertion(Iname : Idescr, individual(name(Iname),class(X),pvlist(Y))) :- parseDescr(Idescr,X,Y).

parseDescr(Class,Class,[]) :- atom(Class),!.
parseDescr(Descr,Class,Pv) :- Descr =.. [Class|PropertyValuesList],parsePvList(PropertyValuesList,Pv).

parsePvList([Pv|T],[Elem|L]) :- parsePv(Pv,Elem),parsePvList(T,L).
parsePvList([Pv],[Elem]) :- parsePv(Pv,Elem).

parsePv((Pname : Pvalue),pv(Pname,Pvalue)) :- !.
parsePv((Pname in Vset),pv(Pname,Pvalue)) :- parseVset(Vset,Pvalue).

parseVset((V , T),[V|L]) :- !,parseVset(T,L).
parseVset(V,[V]).
```

A.2 The QUERY Prolog Theory

```
%%%%%%%%%%%%%% grammar operator %%%%%%%%%%%%

:-op(100,fx,'semantic').
:-op(98,xfy,'matching').
:-op(95,xfy,'and').
:-op(95,xfy,'or').
:-op(92,xfy,'$').
:-op(90,fx,'not').
:-op(85,fx,'exists').
:-op(85,fx,'only').
:-op(80,xfy,':').
:-op(80,xfy,'in').
:-op(78,fx,'#').
:-op(75,fx,'gt').
:-op(75,fx,'geq').
:-op(75,fx,'lt').
:-op(75,fx,'leq').
:-op(75,fx,'eq').
:-op(70,xfy,'/').


parseQueryWithPrefix(semantic Result matching Template, match(var(Result),template(Out))) :-
        var(Result),parseTemplate(Template,Out),!.

parseQueryWithPrefix(semantic Result matching Template, match(Out1,template(Out2))) :-
        parseAssertion(Result,Out1),parseTemplate(Template,Out2).

parseTemplate(Templ,Out) :- parseC(Templ,Out).
 %%%%%%%%%%%%%% assertion %%%%%%%%%%%%

 parseAssertion(Out1,individual(name(Out1))) :- !.

 %%%%%%%%%%%%%% rule C %%%%%%%%%%%%

 %C ::= all

parseC('all' ,atomicConceptDescription('top')) :- !.

%C ::= none
parseC('none' ,atomicConceptDescription('bottom')) :- !.

%C ::= C and C
parseC((C1 and C2) ,intersection(Out1,Out2)) :- parseC(C1,Out1),parseC(C2,Out2),!.

%C ::= C or C
parseC((C1 or C2) ,union(Out1,Out2)) :- parseC(C1,Out1),parseC(C2,Out2),!.

%C ::= not C
parseC(not C ,negation(Out)) :- parseC(C,Out),!.

%C ::= R
parseC(R ,Out) :- parseR(R,Out),!.

%C ::= { Iset }
parseC({ Iset } ,individualSet(Out)) :- parseIset(Iset,Out),!.

%C ::= cname( R )
parseC(C ,intersection(Out1,Out2)) :- C =.. [Cname,R], Out1 = conceptName(Cname), parseR(R,Out2),!.
```

```
%C ::= ( C )
%parseC(( C ) ,Out) :- parseC(C,Out),!.

%C ::= cname
parseC(Cname,conceptName(Cname)) :- atom(Cname).

%%%%%%%%%%%%%% rule R %%%%%%%%%%%%%

%R ::= F
parseR(F,intersection(existentialQuantification(Out),existentialRestriction(Out))) :- parseF(F,Out),!.

%R ::= exists F
parseR(exists F,existentialQuantification(Out)) :- parseF(F,Out),!.

%R ::= only F
parseR(only F,existentialRestriction(Out)) :- parseF(F,Out),!.

%R ::= M
parseR(M,Out) :- parseM(M,Out).

%%%%%%%%%%%%%% rule F %%%%%%%%%%%%%

%F ::= P in C
parseF(P in C, roleExpr(Out1,Out2)) :- parseP(P,Out1),parseC(C,Out2),!.

%F ::= P : I
parseF(P : I, roleExpr(Out1,individualSet(Out2))) :- parseP(P,Out1),parseI(I,Out2),!.

%F ::= P : D
parseF(P : D, roleExpr(Out1,Out2)) :- parseP(P,Out1),parseD(D,Out2),!.
%%%%%%%%%%%%%% rule M %%%%%%%%%%%%%

%M ::= # eq PosInt : P
parseM(# eq PosInt : P, cardinalityRestriction(Out1,relationKind('='),value(Out2))) :-
       parseP(P,Out1),parsePosInt(PosInt,Out2).

%M ::= # lt PosInt : P
parseM(# lt PosInt : P, cardinalityRestriction(Out1,relationKind('<'),value(Out2))) :-
       parseP(P,Out1),parsePosInt(PosInt,Out2).

%M ::= # gt PosInt : P
parseM(# gt PosInt : P, cardinalityRestriction(Out1,relationKind('>'),value(Out2))) :-
       parseP(P,Out1),parsePosInt(PosInt,Out2).

%M ::= # leq PosInt : P
parseM(# leq PosInt : P, cardinalityRestriction(Out1,relationKind('<='),value(Out2))) :-
       parseP(P,Out1),parsePosInt(PosInt,Out2).

%M ::= # geq PosInt : P
parseM(# geq PosInt : P, cardinalityRestriction(Out1,relationKind('>='),value(Out2))) :-
       parseP(P,Out1),parsePosInt(PosInt,Out2).

%%%%%%%%%%%%%% rule P %%%%%%%%%%%%%

%P ::= pname
parseP(Pname, property(Pname)) :- atom(Pname),!.

%P ::= pname / vname
parseP(Pname / Vname, propertyWithBinding(Pname,Vname)) :- atom(Pname),var(Vname).
```



```
%%%%%%%%%%%%% rule Iset %%%%%%%%%%%%

%Iset ::= I , Iset
parseIset((I , Iset) ,[H|T]) :- parseI(I,H),parseIset(Iset,T),!.

%Iset ::= I
parseIset(I , [Out]) :- parseI(I,Out).

%%%%%%%%%%%%% rule D %%%%%%%%%%%%

%D ::= eq N
parseD(eq N, concreteDomainConstraint(relationKind('='),value(Out))) :- parseN(N,Out).

%D ::= lt N
parseD(lt N, concreteDomainConstraint(relationKind('<'),value(Out))) :- parseN(N,Out).

%D ::= gt N
parseD(gt N, concreteDomainConstraint(relationKind('>'),value(Out))) :- parseN(N,Out).

%D ::= leq N
parseD(leq N, concreteDomainConstraint(relationKind('<='),value(Out))) :- parseN(N,Out).

%D ::= geq N
parseD(geq N, concreteDomainConstraint(relationKind('>='),value(Out))) :- parseN(N,Out).

%D ::= S
parseD(eq S, concreteDomainConstraint(relationKind('='),value(S))) :- atom(S).
%%%%%%%%%%%%% rule I %%%%%%%%%%%%

parseI(I,I) :- atom(I).

%%%%%%%%%%%%% rule N %%%%%%%%%%%%

parseN(N,N) :- number(N).

%%%%%%%%%%%%% rule PosInt %%%%%%%%%%%%

%parsePosInt(N,N) :- integer(N),N >=0.
parsePosInt(N,N) :- integer(N).
```

Appendix B

List of Pubblications

B.1 Journals

Elena Nardini, Andrea Omicini, Mirko Viroli, Michael Ignaz Schumacher. Coordinating e-Health Systems with TuCSoN Semantic Tuple Centres. *ACM Applied Computing Review.* 2(11), 43-52, 2011.

Elena Nardini, Andrea Omicini, Mirko Viroli. Semantic Tuple Centres. *Science of Computer Programming.* **Submitted**.

B.2 Edited Volumes

Maria Cristina Matteucci, Andrea Omicini, Elena Nardini, Pietro Gaffuri. Knowledge Construction in E-learning Context: CSCL, ODL, ICT and SNA in Education. *CEUR Workshop Proceedings 398*, 1-2 September 2008.

B.3 Conference Proceedings

Elena Nardini, Ambra Molesini, Andrea Omicini, Enrico Denti. SPEM on Test: the SODA Case Study. *23th ACM Symposium on Applied Computing (SAC 2008)*,16-20 March 2008.

Elena Nardini, Andrea Omicini, Maria Cristina Matteucci. Toward a Framework for Collaborative Learning based on Agent-based Technologies. *The 2008 International Education, Technology and Development Conference (INTED 2008)*,3-5 March 2008.

Elena Nardini, Matteo Casadei, Andrea Omicini, Pietro Gaffuri. A Conceptual Framework for Collaborative Learning Systems Based on Agent Technologies. *The 2008 International Conference on the Interactive Computer Aided Learning (ICL 2008)*, 24-26 September 2008.

Ambra Molesini, Elena Nardini, Enrico Denti, Andrea Omicini. Situated Process Engineering for Integrating Processes from Methodologies to Infrastructures. *24th Annual ACM Symposium on Applied Computing (SAC 2009)*, 8-12 March 2009.

Ambra Molesini, Marco Prandini, Elena Nardini, Enrico Denti. Risk Analysis and Deployment Security Issues in a Multi-Agent System. *The 2nd International Conference on Agents and Artificial Intelligence (ICAART 2010), 2010*, 22-24 January 2010.

Elena Nardini, Mirko Viroli, Emanuele Panzavolta. Coordination in Open and Dynamic Environments with TuCSoN Semantic Tuple Centres. *The 25th Annual ACM Symposium on Applied Computing (SAC 2010)*, 22-26 March 2010. **The paper was selected as a best paper**.

Mirko Viroli, Matteo Casadei, Elena Nardini, Andrea Omicini. Towards a Chemical-Inspired Infrastructure for Self-* Pervasive Applications *Self-Organizing Architectures, Lecture Notes in Computer Science 6090*, July 2010.

Elena Nardini, Andrea Omicini, Mirko Viroli. Description Spaces with Fuzziness. *The 26th Annual ACM Symposium on Applied Computing (SAC 2011)*, 21-25 March 2011. **In press**.

Elena Nardini, Mirko Viroli, Gabriella Castelli, Marco Mamei, Franco Zambonelli. A Coordination Approach to Spatially-Situated Pervasive Service Ecosystem. *The 13th International Conference COORDINATION'11*, 6-9 June 2011. **Submitted**.

B.4 Workshop Proceedings

Elena Nardini, Andrea Omicini. Agent-Based Collaboration Systems: A Case Study. *Knowledge Construction in E-learning Context: CSCL, ODL, ICT and SNA in Education, CEUR Workshop Proceedings 398*, October 2008.

Ambra Molesini, Elena Nardini, Enrico Denti, Andrea Omicini. Advancing Object-Oriented Standards Toward Agent-Oriented Methodologies: SPEM 2.0 on SODA. *9th Workshop "From Objects to Agents" (WOA 2008) Evolution of Agent Development: Methodologies, Tools, Platforms and Languages*, November 2008.

Elena Nardini, Andrea Omicini, Mirko Viroli. General-Purpose Coordination Abstractions for Managing Interaction in MAS. *The WI-IAT 2009 Workshops Proceedings*, 15-18 September 2009.

Elena Nardini, Mirko Viroli, Matteo Casadei, Andrea Omicini. A Self-Organising Infrastructure for Chemical-Semantic Coordination: Experiments in TuCSoN. *WOA 2010—Dagli oggetti agli agenti. Modelli e tecnologie per sistemi complessi: context-dependent, knowledge-intensive, nature-inspired e self-*, CEUR Workshop Proceedings 621*, 5-7 September 2010.