Alma Mater Studiorum · Università di Bologna

Scuola di Ingegneria e Architettura
Sede Di Cesena

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

GAME ENGINES AND MAS: TUPLESPACE-BASED INTERACTION IN UNITY3D

Tesi di Laurea in

Sistemi Autonomi

Relatore:
Chiar.mo Prof.
Andrea Omicini

Presentata da:
Mattia Cerbara

Correlatore:
Ing. Stefano Mariani
Esaminatore:
Prof. Mario Bravetti

III Sessione
Anno Accademico 2016/2017

To my parents,
to my family
to everyone who believes in me

Contents

Sommario

I Game Engines stanno acquisendo sempre più importanza sia in ambito industriale, dove permettono lo sviluppo di applicazioni moderne e videogiochi con relativa facilità, sia in ambito di ricerca, in particolare nel contesto dei sistemi multi-agente (MAS).
La loro capacità espressiva, unita al supporto di tecnologie innovative e funzionalità avanzate, permette la creazione di sistemi moderni e complessi in maniera più efficiente: il loro continuo avanzamento tecnologico e la ricerca di performance e stabilità, li ha portati ad essere una realtà su cui fare affidamento nella produzione di vari applicativi diversi, come ad esempio applicazioni di realtà aumentata/virtuale/mista, simulazioni immersive, costruzione di mondi virtuali e infrastrutture 3D, ecc.
Ciononostante, soffrono la mancanza di proprie astrazioni e meccanismi che possano essere affidabili e utilizzati per aggredire la complessità durante il design di sistemi complessi.
Il tentativo di sfruttare le caratteristiche della teoria dei MAS all'interno degli ambienti di sviluppo dei Game Engines procede secondo questa direzione: integrando le astrazioni costituenti i MAS all'interno dei Game Engines, con particolare riferimento alla teoria degli agenti e ai modelli di coordinazione, può portare a nuove soluzioni e possibilità di creazione di sistemi e applicazioni, riuscendo a risolvere problemi tecnologici grazie all'aiuto degli engine grafici.
Questa tesi offre il suo contributo analizzando il Game Engine `Unity3D` e proponendo due librerie C#, le quali sfruttano una precedente integrazione dello stesso framework con il Prolog per l'abilitazione di un modello di interazione e coordinazione basato su spazi di tuple, utilizzabile tramite l'implementazione di primitive `LINDA`. Le librerie offrono interfacce di programmazione (API)

sfruttabili dai programmatori C# Unity3D per integrare nelle loro creazioni il supporto a tale modello, il quale costituisce una nuova modalità per la gestione della coordinazione tra oggetti in Unity3D e fornisce importanti proprietà, essendo fondamentale nel contesto dei MAS dal punto di vista dell'ingegnerizzazione di sistemi complessi e della gestione delle interazioni tra agenti.

Abstract

Game Engines are gaining more and more importance in both industrial field - enabling an easy development of modern applications and videogames - and in the research field, with a particular regard to multi-agent systems (MAS). Their expressive power, along with modern technologies support and advanced functionalities, allow the design and creation of complex systems efficiently. While becoming more stable and supporting new functionalities, Game Engines are nowadays a well-suited reality, therefore reliable in order to create modern, advanced systems and applications (e.g. augmented/virtual/mixed reality, immersive simulations, modern videogames, 3D world design).

Nevertheless, they still lack of proper general abstractions, which could be suitable to be used for tackle complexity when designing and implementing complex systems.

For this purpose, the attempt to exploit Game Engines as integrated development environments where MAS abstractions are well-suited to tackle complexity is worth to be done, in order to bring agent-oriented software engineering and coordination models as main providers of new solutions and solving technology problems with the aid of Game Engines.

In this dissertation, we analyse the `Unity3D` Game Engine and present two prototype API, meant to exploit a previous integration of Prolog within the same framework to provide Unity3D of a new abstraction level, enabling a tuplespace based interaction and coordination model to be used by programmers via `LINDA` primitives.

Those libraries are organized around `Unity3D` features, providing an easy-to-use and accessible way to approach coordination and interactions among objects when building complex systems with Unity3D, bringing all those

properties and advantages that are well-established in MAS context, both from the complex systems engineering point of view and management of social interactions side.
All new functionalities are tested using proper case studies, which illustrate their expressiveness, correctness and efficacy.

Chapter 1

Introduction

Multi-agent systems (MAS) are nowadays becoming important as a paradigm for designing and implementing complex software systems and applications. In the computer science context, complexity claimed its own critical importance, considering that today systems and applications address many different issues, which need to be faced and solved by adopting some sort of conceptual abstraction and model.

Agent-oriented computing [27] is, indeed, well-suited to provide appropriate concepts in order to design, realize and model complex artificial systems, helping designers and software engineers to better conceive and manage their creations.

Nevertheless, what characterizes the *agent* notion are autonomy, interaction and task concepts [6]: the agent is able to proactively take decisions, as an autonomous entity able to interact with other agents and the environment to achieve its tasks.

On top of that, the role of coordination is to globally ensure system functionalities by correctly handling the local agents activities and interactions, meant to make mobile entities work together, in order to achieve the goal(s) sought for also addressing *society* as critical concept.

Merging all together, it is clear how the multiplicity of entities and tasks which dominate MAS scenarios contribute to the growth of complexity when designing and handling multiagent systems and applications. With the election of interaction as one of the key roles when designing MAS, it is possible to take advantage of its features, because coordination models and languages

allow technologies and abstractions to be used while engineer complex computational systems.

Against this conceptual background, `Game Engines` are becoming popular and used for many purposes, permeating various computational research areas due to their features able to allow for the creation of complex scenarios in an easy and fast way.
In particular, Game Engines provide functionalities in order to build modern applications: although they are successfully used in the videogame industry, allowing game development to be opened to everyone (due to their user-friendliness), they are also present in the MAS context [21], providing functionalities to enhance world creation with augmented reality possibilities, immersive simulations, and so on.
In this way, Game Engines and MAS come together, merging into a set of possibilities able to exploit strength of both worlds, in particular game engines exploitation within the agent-oriented infrastructure, providing world creation and modern technologies to be tackled using MAS theory, patterns and abstractions.

Focussing on the key role of interaction and coordination models, this dissertation contributes to this purpose by taking a Game Engine and extending its functionalities with a new abstraction level: MAS theory integration may provide novel solutions and new ways to address complex system design and engineering, where Game Engines play the role of high-quality and modern technologies enablers.
On top of that, this dissertation presents a new abstraction level developed both using Prolog and C# languages, in order to allow Unity3D designers and programmers to use a new interaction and coordination model via C# libraries, exploiting tuplespace based coordination implemented in Prolog.
In particular, basic semantics of tuplespace based model have been developed using Prolog (previously needed to be integrated in Unity3D) in the form of `LINDA` primitives [12], while a higher-level communication layer has been implemented using C# language with the creation of libraries, named `LindaLibrary` and `LindaCoordinationUtilities`, which wrap `LINDA` prim-

itives allowing their exploitation during Unity3D development.
These libraries capture and organize the tuplespace based coordination model and `LINDA` coordination language, allowing their use under Unity3D system design and development, including mechanisms to create spatial tuplespaces and region around the *SpaT* extension from [30].
The reminder of the thesis is organised as follows: Chapter 2 analyses the background of this dissertation and the motivation, discussing MAS theory, Game Engines state of the art, and then provides an overview of principal coordination models, focussing on tuplespace based ones.
Chapter 3 is an overview of the Prolog integration in Unity3D, chosen as the enabler technology in order to correctly develop `LINDA` primitives and basic semantic of tuplespace based interaction model. Chapter 4 describes how Unity3D libraries and Prolog base predicates are engineered and developed, how they are intended to be used and what kind of new abstraction level they provide to the base Unity3D IDE.
Chapter 5 presents a first extension of the C# libraries introducing spatial tuplespaces and regions as augmentations of standard ones, following the *SpaT* model.
Chapter 6 shows the test phase, analysing and discussing the efficacy of these libraries through different case studies.

Chapter 2

Background

This chapter introduces and reviews the basis of the main work, summarizing the fundamentals about Game Engines (GE) and explaining why they are so important in current industry, with a particular mention to Unity3D (the one taken as reference for this thesis), architecture and abstractions of Multi-Agent Systems (MAS). The aim is discussing and highlighting their relevance in different research fields, while focusing on interaction/coordination models, their importance in MAS theory and their exploitation in the design of complex multi-agent systems (with a particular reference to LINDA-based coordination models).
Despite their current wide adoption in modern industry, GE are well suited for developing videogames and realtime simulations, providing a wide range of functionalities and helping developers to build different kind of applications, but they still lack of general-purpose abstractions for developing complex multi-agent applications.
On the other hand, MAS theory represents the richest and most used abstraction source for complex systems design and engineering, along with the adopted interaction and coordination model, both important in order to tackle the system complexity and govern agent interactions.

2.1 Motivation

MAS and GE both provides functionalities and useful abstractions exploitable to build complex systems and applications, but they are still quite unable

to properly mix their respective features, so further investigations must be made [21].
MAS theory and abstractions provide a well-suited and rich conceptual model when designing multi-agent systems: agent-oriented software engineering, along with interaction and coordination models, are technologies which enable completeness and generality in software systems development, exhibiting a well-established technological framework where advanced conceptual tools could be used as fundamentals when computer scientists and engineers approach to systems engineering.
At the same time, Game Engines are well-settled in modern industry and they have gained a crucial importance in games and application development: nowadays, the gaming scene can count on rich qualified investors and on a billionaire industry, along with a wide variety of developers and gamers. Game Engines are thus a well-funded, modern reality exploitable to design new-fashioned, up-to-date applications using latest technologies (immersive simulation scenarios, augmented/virtual/mixed reality, and so on), providing a stable, performing and well-usable framework for building high-quality software systems.
Both worlds are enablers of features which are complementary, so their strengths can be properly mixed in order to provide novel solutions along with MAS design and developing modern applications using Game Engines functionalities.
Although GE are gaining more and more importance in the academic community and computational research areas, they still are focussed on technology-level purposes, searching for stability and frame performance while increasing their rendering possibilities and, in this development environment, a proper integration with MAS abstractions (in particular, focusing on interaction and coordination models) can certainly bring to new features, solutions and opportunities.

2.2 Goal

Among all typical MAS abstractions (that are, agent, society and environment, introduced later) what is still not properly available in the game de-

velopment scene are MAS counterparts and support for agent and society models [21]: with no first-class abstractions provided, their MAS bindings are quite distant, since the abstraction gap is still demanding of proper integrations, generally missing.
So, although the environmental support can be conceived as well-present in GE's internal features and basic structure, such as built-in functionalities supporting high-level rendering, world creation, its control and modelling, collision detection and automatic pathfinding, this is the only part with a greater support than the average MAS technologies.

In this direction, this dissertation aims to provide a first integration step between MAS and GE, in particular focusing on Unity3D and the societal abstraction, with the creation of something concrete and useful to Unity3D's designers and programmers, exploiting game engine's features and possibilities to fill the gap with interaction and coordination models. Moreover, other important achievements are providing functionalities as societal abstractions to tackle complexity, allowing communication between agents and objects and, possibly closing conceptual and technical gaps between MAS and GE under the interaction and coordination point of view.

2.3 Game Engines

Many different areas of computer science and engineering are discovering an increased popularity of Game Engines, where they are exploitable for building realistic, virtual systems tackling process complexity, with strong economic (enabling modern application development) and time efforts (an user-friendly framework allows to save time, with fast prototyping) [33]. Indeed, building scenarios with complex 3D objects, supporting user interaction and required object/environmental behaviours (collision detection, pathfinding, audio, obstacle avoidance, ...), are built-in, easily exploitable from the engine itself, which provides development toolkits and user friendly features in order to fast prototype and design a complete, complex system.

The current generation of game engines has become crucial in game de-

velopment, making the realization of virtual environments and complex systems easier using Game Engines' functionalities, which are robust and widely tested (most of the time for performance purposes). In particular, GE address the extensive reuse of the underlying technologies, like 2D/3D rendering and world creation/handling (audio, physics, dynamics, ...), basic AI support for simple agent development and movement system (pathfinding, obstacle avoidance, ...), with a profound customization and inexpensive, time-friendly development of a variety of systems and applications.
Moreover, game engines are used for research purposes in order to make immersive simulations and test AI algorithms, building scenarios for advanced solutions (AI, robotics, swarm, ...), so not only videogames but also simulations and modern applications (e.g. simulated surgical training [22]).
Among these functionalities, game engines are suitable for designing complex world faster and simulation prototyping, while having a modular composition [17], so GE can add values to the developed application (providing 3D customization and creation, hardware graphic acceleration, support for modern drivers and technologies) and can be properly reshaped for research purpose. Examples in this direction are GameBots [15], CIGA [34], QuizMASter [2], and so on, all with repurposed capabilities in order to tackle some of the MAS abstractions for different goals (from education, providing immersive learning environment, to distributed military simulations), but still without properly proposing models or patterns close to MAS ones and with the proper level of abstraction.
In addition to that, game engines as frameworks and integrated development environments (IDE) play a key role during design and development, providing libraries and functionalities where most of complex computations and control flow is delegated to the engine itself. In fact, the application structure is defined and immutable, same for how the execution has to be handled, while designers and programmers are provided with all necessary building blocks to properly manage and model the central architecture.
This indeed enables consciousness in developers when using a specific functionality, knowing that every block composing the application/system is handled, managed, organized and shaped by the game engine isolating it from the hardware in a middleware style.

Among the wide variety of game engines available to be used (`Unreal Engine`[1], `CRYENGINE`[2], `Source Engine`[3], `id Tech` 4/5/6 (different versions[4]), and so on), the chosen one for this thesis is `Unity3D`: motivations, along with its description, features and abstractions, are in the next subsection.

2.3.1 Unity3D: Features

`Unity3D`[5] is a cross-platform game engine developed by Unity Technologies, used for creation of videogames (both 2D and 3D) and simulations, supporting distribution to a variety of platforms (PC, console, mobile devices, and so on).
It features interesting abstractions which contribute to extend its usage to a variety of developers and programmers, allowing it to become one of the most used game engine and to be used by a wide variety of developers to quickly produce applications and games. This vision generated a democratization in the game development industry, with the goal of making it universally accessible and opening it to everyone due to its simplicity.
Moreover, this GE supports a number of features that are simple to use and exploitable for creating realistic videogame and immersive simulations, such as an intuitive, real-time editor, integrated physics system, dynamic lights, 2D/3D objects development and import, shaders, basic AI support (built-in pathfinding, obstacle avoidance, ...), and so on.

Unity3D is a multi-purpose game engine, it supports target graphic API of a wide range (like *OpenGL*, *Direct3D*, *WebGL*, *Vulkan* and so on) in order to create applications and systems for a variety of technologies (mobile, console, etc...), while supporting advanced 2D world renderer, texture compression, and other services in order to build complex worlds relying on both internal creation engine and professional external development software (like *Maya*, *Blender*, ...).
In addition to that, it enables quick prototyping providing a simple, modular

[1] https://www.unrealengine.com/en-US/blog
[2] https://www.cryengine.com/
[3] https://developer.valvesoftware.com/wiki/Source_Engine_Features
[4] https://www.idsoftware.com/en-gb/
[5] https://unity3d.com/

internal architecture, exploitable by designers to create, design and model their creation, properly using Unity3D abstractions to tackle complexity. it is possible to create scenes, each one with its own purpose and construction, where `GameObjects` are the most important concept present in the Unity Editor conceptualizing every object placed in the scene (from lights to complex 3D building, from particle effects to an autonomous robot), which can be composed of many build blocks, called `Components`, each of them adding proprieties, features and functionalities to the GameObject itself.
In this manner, different combinations of Components are able to shape and model the kind of GameObject the developer is going to create. In addition to that, Unity3D provides developer with an Editor Mode, with different editor windows allowing full customization of the application and making possible to directly create, position and interact with world creation, GameObject customization, enter Play Mode and test on-the-fly the currently developed application, use of Profiler to investigate performances, and so on.
Unity3D has been used not only for game development (obviously, its main computational and lucrative field) but its policy is to make it a standard for also simulations and modern applications development, providing support to a huge variety of different platforms covering the most interesting and novel technologies. Further examples are from the area of geographic information system [38] to the mobile AR development of a reality game which could tackle worldwide childhood obesity [16] and in exploiting Unity3D user experience tools to experiment urban design projects for visualization and interactivity purposes [14].
Summing up, motivations for Unity3D adoption are the following:

- simple to use, exploiting the Editor Mode it is possible to quickly and intuitively create complex worlds, along with its customization and on-the-fly testing phase (Play-Mode Window)
- simple and completely customizable GameObjects' creation and manipulation using Components as building blocks, for example *Rigidbody* (allowing the object to be subject to forces and Physics in general), a *Collider* (providing the GameObject of an editable surface, different from the Rigidbody, which is responsible for collisions with other objects and agents), an *Animator* (enabling animations on the specified

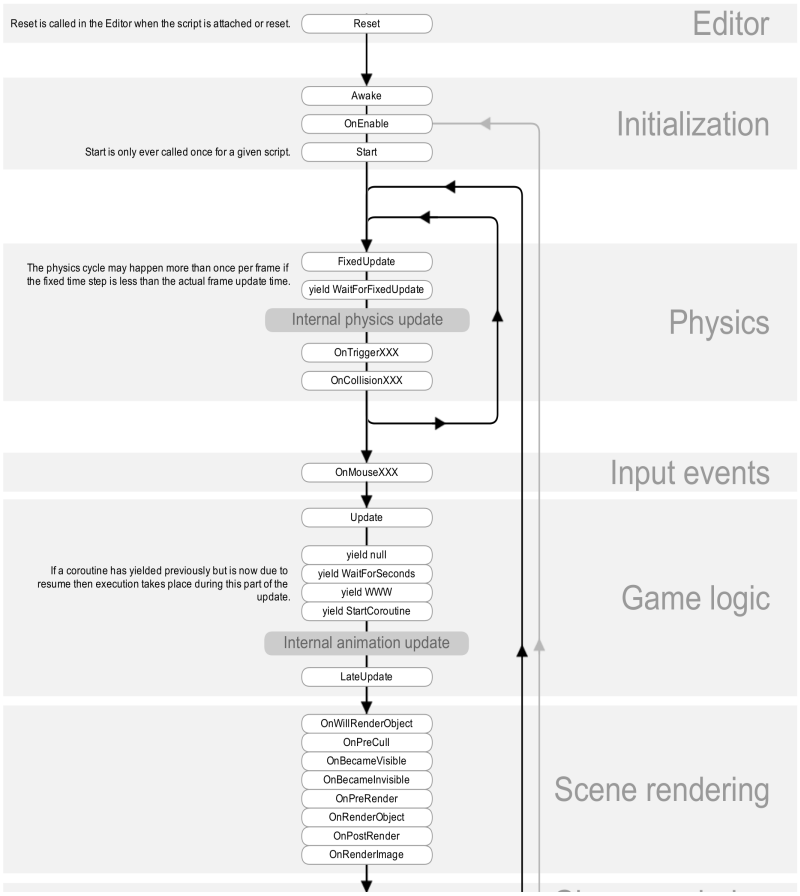


Figure 2.1: Sample of Unity3D's MonoBehaviour flowchart, from [1]

GameObject) and a *C# Script* (the behavioural module of an object/agent). Moreover, the control flow is executed by the unique game loop as depicted in Figure 2.1, a single thread which sequentially calls every MonoBehaviour Script enabled in the Unity3D scene every frame, trying to maximize performances, along with the very base one called Transform (presents in every GameObject, it allows the object to have a position,a scale, a rotation and other fundamental proprieties)

- lower abstraction gap with Unity3D game engine rather than with other GE, because of its pervasive usage and exploitation for developing academic courses' final projects and for the object-oriented paradigm for programming Scripting Components, coded using *C#* language in a more intuitive way

2.4 MAS theory

The increasing complexity of software systems engineering has led to the necessity of models and source of abstractions which could make easier their design, development and maintenance. In this direction, agent-oriented computing comes to help engineers and computer scientists to build complex, virtual or artificial systems enabling their easy and correct management [27]. In particular, MAS research and technologies brought new abstractions in order to tackle complexity while designing systems or applications composed of individuals no more acting alone but within a society. Agent oriented technologies and models have currently become a powerful technology to deal with many issues to address while designing computer-based systems in terms of entities sharing an amount of features like autonomy, intelligence, distribution, interaction, coordination, and so on.

MAS engineering is indeed about building complex systems where multiple autonomous entities called agents proactively achieve their goals exploiting interactions as society with each other and with the surrounding environment (Figure 2.2). This model can be seen like a general-purpose paradigm, so well suited also for applications in software, agent-oriented engineering practices in different scenarios [42].

Agent models ground their practical definition on the concept of autonomy as their fundamental feature and they define MAS, conceived as an aggregation of interacting agents. An agent encapsulates complexity in terms of information (what it needs to know in order to do something), actions (the process to achieve some goal), intelligence, mobility, situatedness, interactivity. In this way, the communication with other agents and interaction with the environment itself become a fundamental abstraction of the system engineering, as well as being potential enablers of behavioural novelties and actions [40]).

So, the key role played by coordination and interaction models is seen not only as an adequate supporting abstraction needed by the agent autonomous behaviour, effectively solving coordination problems and enabling interactions among agent society exchanging information and knowledge, but also as a suitable approach directly helping the design phase of MAS [6].

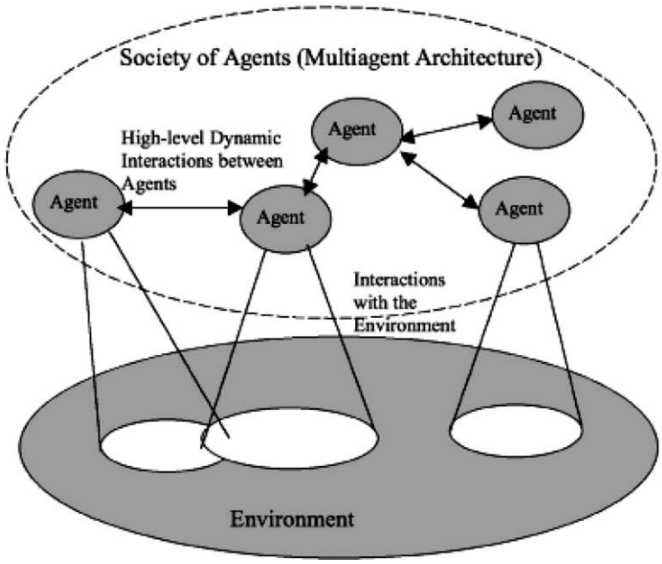


Figure 2.2: Multi-Agent System original architecture, from [42]

Here follows a description of the three basic MAS abstractions, highlighting how much autonomy is pervasive and acts like a key concept when conceiving MAS as aggregation of multiple agents able to interact with each other exchanging information.

2.4.1 Agents

Agents are the fundamental abstraction in MAS definition: proactive entities which encapsulate control, governing it through actions which allow the agent itself to pursue its goals (what they want to achieve) eventually using and changing something in the world they are immersed into (percepting the state of the environment and adapting its actions model and behaviour to it), as depicted in Figure 2.3. In this sense, agents are situated, so strictly coupled with context and surrounding environment, and most importantly they are social, so expressing autonomy with interactions among agents as living in a MAS society

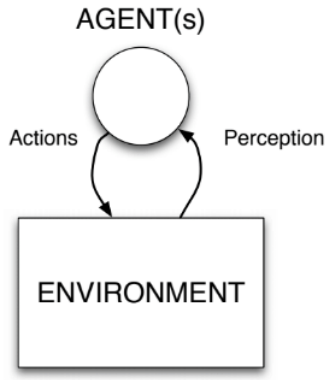


Figure 2.3: Agent perception and action cycle, from [42]

2.4.2 Society

Agents' social ability is the capability of interacting with other agents in order to obtain coordinated collective behaviours, since some goals can only be reached with collaboration and interaction with other agents. In this sense, MAS complexity can be tackled using interaction and coordination models, which are conceived to be the key issue when designing MAS and dealing with complexity of interactions.
The space of interactions, here present as a fundamental requirement for the society abstraction, is one of the main sources of complexity when dealing with MAS design and implementation. Different models have been introduced, surveyed and nowadays are well-suited to bring important features in order to harness the interaction space, some of the most relevant ones will be discussed in section 2.6 as the core topic of this dissertation.

2.4.3 Environment

Another key abstraction when dealing with MAS development is the environment, which can be changed by agent's actions [29], providing mediation to component interactions enabling indirect communication/coordination among external resources and agents and making environment feature important properties like activity, object's situatedness, autonomous dynamics influencing interactions and coordinations among components.
MAS modelling and engineering complexity lies also in the environmental

concept [40] which, along with well-estabilished and suitable abstractions provided by a proper interaction and coordination model, makes possible to exhibit and govern mechanics and actions which are strictly coupled with environment properties, in the form of situated actions and sensitivity to environmental changes.

2.5 Logic Programming and Prolog

This section provides a brief introduction to logic programming and languages, focusing on the role of Prolog when dealing with interaction and coordination models.

2.5.1 Logic Programming: overview

The main topic in logic programming is using logic both as declarative representation language and theorem-prover. With the growth of complexity in MAS, where agents encapsulate intelligence and undertake interaction with each other in order to achieve their goals, abstractions and services able to simplify designers and developers tasks are needed, along with easily deployable infrastructures but still providers of important features, such as easy configuration, intelligence encapsulation, interaction rules, and so on.
Logic programming and languages are handy when dealing with intelligence, so in AI and similar areas, but they could play a key role even in design and implementation of coordination and interaction models, providing flexible, non-trivial solutions and architectures enabling logic as coordination media. In particular, dealing with general scenarios of concurrent/parallel computation, logic programming and languages have been proved to be effectively used as enablers of effective coordination solutions, as had happened with Prolog.

2.5.2 Prolog: overview

Prolog is the reference for logic programming and it is present in different areas and infrastructures, for example supporting DSLs implementation, reasoning-like computation, theorem-prover, AI applications, and so on.

Focusing on the coordination and interaction point of view, Prolog support enables the creation of basic coordination capabilities by exploiting its features (like declarative syntax, backtracking, template matching and unification, ...) to create tuplebased coordination models, also with the adoption of Prolog based interaction metaphors. Moreover, both TuCSoN [26] and LuCe [7] coordination infrastructures are provided with Prolog engine as their core component (named tuProlog [8]), which acts as the enabler of a `LINDA`-like tuplebased interaction model and provider of all necessary properties.

2.6 Coordination and Interaction models: overview

"A coordination model
is the glue that binds separate activities
into an ensemble [13]

When modelling and implementing MAS, a key role is played by coordination which consists in an important aspect needed to be properly analysed and tackled. The growth of complexity in software applications and MAS brings the necessity of new and suitable models, able to bring specific properties to the system (flexibility, control, openness, distribution, ...). Moreover, this class of functionalities needs to be provided with appropriate concepts and abstractions in order to ensure proper functionalities both with a global vision and looking to local interactions among agents, needed to be properly coordinated.
Coordination and interaction are thus providers of mechanisms and abstractions able to tackle complexity of such dynamic, heterogeneous ensemble of agents, not delegating to them coordination responsibilities (constraining interaction protocols directly within agents themselves) but considering an higher-level design focusing on the agent interaction space, approaching to it as a design building block [6]. Embedding social rules and policies, building abstractions able to manage and govern the interactions among agents [39],

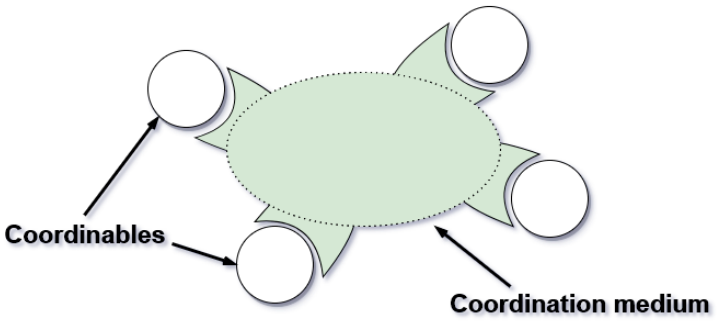


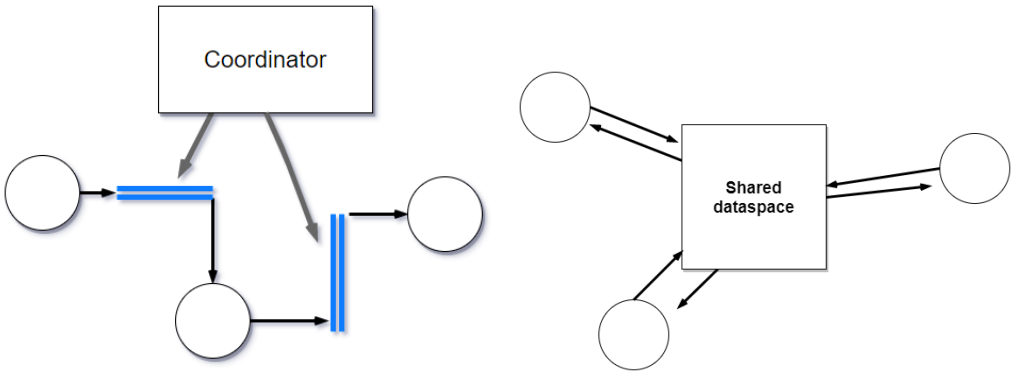
Figure 2.4: Coordination medium and coordinables explained

are the key roles of coordination and thus they are provided by interaction and coordination models.

2.6.1 Major classes and models

According to [4], coordination and interaction models can be conceived as 3 components: (*i*) *coordination entities*, so the type of entities (*coordinables*, agents in MAS) whose interactions are governed by the model, (*ii*) *coordination media*, the abstractions allowing agent interactions and organization among coordinated components, (*iii*) *coordination laws*, which express the behavioural part of the model, how coordination media and coordinables are meant to be ruled w.r.t. behaviour and interactions expressed in terms of communication and coordination languages (providing syntax and admissible primitives to be used).
Coordination models can be divided into two classes [28]:

- *control-driven*, where communication among components (agents) is governed by channels/ports, whose observation of involved coordination patterns defines the state of the computation, with no focus on data types and information involved during the interaction (Figure 2.5(a)). Input/output interfaces are clearly defined and the coordinator component is clearly separated from coordinables, managing event/signals among them and determining/changing the topology of communication space
- *data-driven*, where coordinables interact using channels like shared spaces



(a) Control-driven coordination model (b) Data-driven coordination model

Figure 2.5: Classes for coordination models

or memory abstractions, coordinating by exchanging data structures and information chunks through the coordination media, so the state of the computation is defined by both data structures involved and configuration of coordinated entities (Figure 2.5(b)). So, rather than control-driven models, the coordination medium does not control the topology of communication space, but the act of governing interactions means determining data structures representation, their usage, access, manipulation and synchronization exploiting coordination primitives provided by the model itself.

Many models have been proposed and used in literature and different trends are currently shaping research domains, all of them showing that interaction and coordination are a key issue when developing complex MAS. For example, nature-inspired coordination models for MAS engineering [24] provide abstractions to deal with complexity while building artificial systems, bringing to them those properties (self-regulation, autonomy, adaptation, fault-tolerance, ...) and patterns which feature natural systems (such as chemical, biochemical, physical, biological, ...) making them sources of inspiration for strategies and for tackling and governing complexity.
In this direction different areas have been explored, for example studies on social insects behaviour and stigmergic coordination in ant colonies with environmental mediation [23] [32], field-based models inspired by mass and particles movements [19], ...). Moreover, [18] shows how a proper coordina-

tion language, which extends a chemical inspired abstract model, can be used with multiple agents to realize an interactive computational model useful in MAS development.
A different approach to coordination is described in [35], where it has been developed a declarative approach to model coordination of agents using coordination spaces, in order to relieve programmers from implementing interactions manually.
Among them, one of the first (and most used) interaction and coordination model is the tuplespace based one, from `LINDA`, and it is the chosen one for this thesis: next section describes and explains motivations, strength and properties.

2.6.2 `LINDA` and tuplespace based model

The ancestor of every tuple based coordination model is `LINDA` as the very first coordination model and language embracing dataspaces, mechanisms and abstractions for concurrent agent programming and generative communication. Initially exploited and used in the field of parallel programming [13], tuple-based coordination models feature important properties which make them well-suited for the coordination of heterogeneous and distributed systems, tackling complexity with simple yet well-established concepts and abstractions letting them play a key role when designing and building complex MAS.
In tuple-based models, coordinables interact with each other exchanging tuples as information chunks on which coordinated entities are able to synchronize by associatively accessing, consuming and producing using tuplespaces as coordination media.
More in details, `LINDA` captures and formalises generative communication to be used as a coordination and communication language, by providing concepts and simple, yet expressive primitives.
A program in `LINDA` is a collection of ordered and possible heterogeneous tuples, which incorporate the information meant to be exchanged among agents and are available in the tuple spaces, working as the abstraction of the coordination media and containers of tuples. Moreover, in order to browse and retrieve specified tuples, `LINDA` allows the use of an associative access

abstraction, making possible to manipulate the shared tuple space by specifying templates as set/classes of tuples, or via tuple-matching mechanisms, such as pattern matching, unification, and so on.
The `LINDA` coordination language provides 3 basic primitives, able to handle tuples' manipulation and the tuple space itself with simplicity and expressiveness:

- `out(T)`: inserts the tuple `T` into the tuple space
- `in(T)`: retrieves the tuple `T` from the tuple space, with different properties:
  1. `destructive semantic`: the retrieved tuple is destroyed from the tuple space
  2. `suspensive semantic`: if no matching tuple is found, the operation is blocked and the execution suspended, until a valid tuple `T` is found
  3. `non-determinism`: if multiple matching tuples are found, `LINDA` chooses one of them non-deterministically
- `rd(T)`: retrieves the tuple `T` from the tuple space, with different properties:
  1. `non-destructive semantic`: the retrieved tuple is not destroyed from the tuple space
  2. `suspensive semantic`: if no matching tuple is found, the operation is blocked and the execution suspended, until a valid tuple `T` is found
  3. `non-determinism`: if multiple matching tuples are found, `LINDA` chooses one of them non-deterministically

`LINDA` also provides predicative, non-blocking operations, `inp(T)` and `rdp(T)`: although they maintain basic properties of non-determinism, destructive/non-destructive reading and syntax structure, they introduce failure semantic, which means that if no matching tuple is found, a failure is reported, otherwise they return successfully.
`LINDA` operations define a coordination language, suitable for dealing with MAS complexity and providing attractive properties [31], such as:

- *extensibility*, originally conceived for closed, parallel systems, it has been extended with new powerful mechanisms preserving its simplicity, as the result of a continuous development process resulting in new

models and implementations (for example, nature-inspired coordination models)

- *expressiveness*, solving typical coordination problems and tackling distributed systems and MAS design complexity with few, well-suited and easy-to-use primitives
- *suspensive semantic*, both basic `rd` and `in` primitive calls could led to suspend the execution when no matching tuple is found within the targeted tuple space: depending on the unavailability of searching tuples, the blocking semantic occurs both in the coordination medium (via the operation suspension, then its resuming) and in the coordinable entity (internal wait-state until the performed operation returns successfully)
- *associative addressing*, with the possibility of retrieving different kind of tuples via pattern matching or unification mechanisms, with a content-based coordination which accesses information in an abstracted way, based on data availability
- *generative communication*, where tuples generated by coordinable entities are independent from each other while living within the tuple space: tuples are intended to be means of coordination and objects of communication, providing communication orthogonality (space, time and name uncoupling) with no bounds to the generating entity.
- *uncoupling*, abstracting coordinables and tuples from time, reference and space issues: agents do not have to be in the same place to interact (the tuple space resolves locality issues), as well as no names, references or simultaneity are needed (tuples are unbounded, they have their own existence)
- *separation of concerns* (focus only on coordination issues), *asynchrony* and *concurrency*.

As a further evidence of extensibility and expressiveness, from `LINDA` and tuple-based models stem dozens of different and widespread implementations of coordination and interaction models, all featuring the same, simple abstractions and mechanism of the base one but with different shapes and variations (in order to deal with different areas and requirements), suitable both for research purposes and industrial.

Examples are the following: from `T Spaces` [41] and `JavaSpaces` [11] to `TuCSoN` [26], passing through (bio)chemical-inspired models [36] and space-time tuplespaces [37], also shifting from tuplespaces to tuple centres models, defining behavioural modules within them [25], and so on.

Chapter 3

Prolog integration: feasibility study

This section describes the Prolog engine, chosen as the technology enabler of coordination and interaction models as the main supporter of coordination infrastructures (like `TuCSoN` and `LuCe`, as explained before), discussing two different projects, `tuProlog` and `UnityProlog` as candidates for Unity3D integration.
In particular, the aim of this chapter is to explain their strengths and weaknesses and discuss why to chose Prolog as the enabling technology for this project, acting as a necessary precondition for starting the integration process of coordination models, along with how the integration with Unity3D was tackled and motivating all project choices (why take UnityProlog as the official Prolog engine for this project).

3.1 Prolog integration in Unity3D

One of the main unexplored field when using a game engine like Unity3D is the interaction and communication among objects in the scene during Play-Mode and their behavioural design (the so called social abstraction in MAS). In particular, while Unity3D provides the support to easily create 3D complex worlds, non trivial architectures, physics responsive objects and other features, the communication part is still relying on the procedure call mech-

anism[1].
However, despite it is well suited for game engines and performance (most of the computation must be finished within the single frame), it is not expressive enough to provide a general communication and interaction mechanism, which could be useful in order to design and implement collaborative and situated systems where multiple objects and agents could generally interact with each other within the same physical environment.
In this thesis, the main purpose is to provide the game engine of a better interaction and communication support between agents and, more generally, among GameObjects within the same scene.
This expressive coordination power is based on tuplespaces, where chunks of information are used to provide information-based coordination support and with `LINDA` primitives as the main manipulative actions in order to interact with tuplespaces by adding, removing and retrieving tuples. Moreover, `LINDA` primitives, tuples and tuplespaces concepts have been used in `TuCSoN` [26] and `LuCe` [7] projects where these concepts were specified with a logic-based communication model using a first-order logic notation, so exploiting the Prolog engine features like partial template specification using logic variables, backtracking, declarative syntax and unification as the matching mechanism.
With this strong background it was chosen to add the Prolog engine support to Unity3D, allowing logic programming and coordination mechanisms to be exploited during system development, providing tuples and tuplespaces as first-order logic terms to be used along `LINDA` primitives and other high-level communication utilities as well.

So, as an engineering process, steps are required in order to properly divide the workflow:

- basic integration of a Prolog engine within Unity3D environment
- design and development of a `LINDA`-like library in Prolog, fully accessible from Unity3D's C# Scripting mechanism
- provide general but still simple high-level interaction and coordination mechanisms to be exploitable during agent behavioural development

[1] Interacting with some GameObject means calling a specific procedure within its C# script, via the Unity3D's `SendMessage` utility

In the next sections is described which Prolog engine was chosen to be integrated in Unity3D, in particular is explained why the first choice (tuProlog) has been rejected in order to proceed with UnityProlog, analysing strengths and weaknesses of both proposals.

3.1.1 tuProlog attempt

tuProlog [8] is a Java-based Prolog designed to be exploited for distributed applications and Internet-based devices, providing a Prolog inference engine as a Java class in order to be executed by many different applications or processes at the same time, while the configuration of each node is done independently according to differences, needs and prerogatives of the system components. The support of tuProlog could be interesting also because of its integration with basic coordination capabilities (like tuple spaces), in fact it is the core of both `TuCSoN` and `LuCe` infrastructures as coordination feature of Internet-based MAS.
Moreover, it provides a minimal, yet efficient, ISO-compliant Prolog VM, with a simple interface and a light-weight engine containing only the minimal and essential properties of a Prolog engine, an important characteristic if the system development is strongly influenced by performance frame by frame as Unity3D and all other game engines do.

3.1.1.1 Why it is a failure (for now)

The main problem with integrating tuProlog middleware with Unity3D resides within its intrinsic implementation: the Prolog VM is natively written in Java, while Unity3D provides a scripting environment available in C# and a core written in C++, so a standard integration using tuProlog JAR was not possible.
`tuProlog` also comes with a native support for multi paradigm languages, in fact the same library was ported with IKVM[2] in order to be available on .NET platforms: this version, called `tuProlog.NET`, provides a clean and possibly seamless integration with systems based on .NET framework, libraries and object-oriented programming languages (such as C#), so it could possi-

[2] https://www.ikvm.net/

bly be integrated with no big effort in Unity3D.
Actually, Unity3D runs on a different version of .NET (in particular, Unity3D is a native application built in C++, but uses *Mono* as scripting framework on quite old C# versions, like 3.5), so one of the big problem of Unity3D is that it is stuck on old .NET versions, not supporting all new features of recent framework, and only recently has provided an experimental version of its framework, supporting .NET 4.5 and newer versions of C#, as follows:

- **Unity "STABLE" Version:**
  `Scripting Runtime Version`: Stable (.NET 3.5 Equivalent)
  `Scripting Backend`: Mono
  `API Compatibility Level`: .NET 2.0 Subset
- **Unity "EXPERIMENTAL" Version:**
  `Scripting Runtime Version`: Experimental (.NET 4.6 Equivalent)
  `Scripting Backend`: Mono
  `API Compatibility Level`: .NET 4.6

The tuProlog.NET release version is 4.X, while Unity3D provides a stable version using .NET 3.5, so the only way is trying to integrate tuProlog via Unity3D's experimental version: here, encountered problems are different, as follows:

- Unity3D stable version uses a very old .NET framework, which is not supported by the current tuProlog.NET release (it requires .NET Framework 4.X)
- Unity3D experimental version could be the supported one, but it is not yet stable and the integration process was impossible due to internal problems end errors
- tuProlog.NET was generated using IKVM, both for the cross-platform support and the automatic translation of Java bytecode into dll libraries, which could be used within .NET frameworks, but Unity3D ran into several problems probably caused by .NET 3.5 clashing with the experimental version and runtime resources problem

Due to all these problems, the tuProlog-Unity3D integration was set aside,

but future investigations could be useful when the experimental version will be more stable[3].

3.1.2 UnityProlog attempt

`UnityProlog`[4] is a mostly ISO-compliant Prolog interpreter for Unity3D developed by Ian Horswill, from EECS Department, Northwestern University: this project is a simple Prolog interpreter developed on purpose for Unity3D, so fully compatible with the game engine, providing Prolog theory, rules and code to be properly mixed with the standard Unity supported languages. The following section will explain why UnityProlog was the chosen one, its features and limitations.

3.1.2.1 Features and limitations

UnityProlog comes with a list of features useful to bring the Prolog inference engine within Unity3D, here are described the main features of the interpreter:

- ISO-compliant, so compatible with the majority of ISO standards, supporting a variety of the built-in Prolog predicates and utilities
- direct access to Unity objects, methods and functionalities from within Prolog code and vice versa
- Unity GameObjects are supported to be each one with a different (and personal) knowledge base (KB). that is the place where Prolog theory and code will be executed and stored, with the possibility of interacting with a global KB
- Indexical name support (global values with dynamic binding), a way to easily obtain references to the GameObject currently running the Prolog theory, the current time, and so on
- Thread-safety of the interpreter, so the Prolog engine could be executed on multiple threads but all assertions/modification of the Prolog database must be done safely

[3] Actually, it is a bug of the current release of Unity3D, so needed to be re-investigated once fixed
[4] https://github.com/ianhorswill/UnityProlog

- Basic tool for debugging and logging within Unity3D (using built-in console)

Besides all these features, the project itself has some issues and limitations:

- It is an interpreter rather than a compiler, so the usual Prolog engines are faster than this version
- It uses the C# stack as its execution stack, so tail call optimization is not yet supported
- Doesn't support rules with more than 10 sub-goals
- Doesn't support the existential quantification construct '^' and the full version of *setof/3*
- Some changes in data type for a better integration with CLR languages (like C#), as follows:
  1. `strings` in Prolog are real CLR `strings`, not lists of numbers
  2. `true` and `false` in Prolog are real `true` and `false` boolean values
  3. the special character $ is used as a prefix value for special values referral, so it is not possible to use $ as prefix operator for legacy and custom code.
- Poor documentation support, this thesis will also provide with the explanation of the core UnityProlog concepts and how to use them within Unity3D
- Because Prolog uses the C# stack, debugging traces generated after unhandled exceptions show the correct series of predicates called, but show variables as being unbound (because they were unbound during the process of exception handling), this is actual a bug reported in the documentation.

Among all features provided by UnityProlog, there are indeed a couple of extreme interest regarding tuplebased interaction models integration: interoperability between Prolog and Unity's scripting environment and CLR code in a quick and simple way, described in details in next sections. Furthermore, the ability to enable different KBs for each different Unity object is extremely interesting, along with the fact that we could potentially enable the inference engine all over the scene, making each GameObject as a tuplespace carrier.

Although all these features are listed and described, the project documentation doesn't cover properly how to use it in order to enable Prolog in Unity3D (the code explaining is not correct most of the time), so in the next sections it is described in details how UnityProlog can be used in Unity3D, which are the fundamental components and structures to use, how to enable KBs in every object and how the interaction Unity/Prolog works.

3.2 Coordination and interaction in Unity3D

Besides enabling the expressive power of the Prolog engine within Unity3D as a new paradigm to be exploited when developing complex systems and videogames, this integration makes possible to achieve and realize the coordination dimension under different points of view.
So, the objective of this section is to introduce and explain how the Unity-Prolog integration process has been tackled and developed.

3.2.1 Prolog support in Unity3D

This section describes how UnityProlog constructs and functionalities can be used to provide Unity3D with the Prolog support, in particular it is explained how to set up a simple scene with objects supporting local Prolog theories (embedded in the GameObject itself), the concept of Knowledge Base (`KB`) and GlobalKB as a global GameObject with general theory loaded, how to call Prolog predicates and how the interoperability Unity3D-Prolog works.
Plus, it is presented the fundamental script in order to enable a Prolog engine within every object.

3.2.1.1 UnityProlog's KnowledgeBase (KB)

The `KnowledgeBase` is the fundamental construct of UnityProlog, which enables the usage of the Prolog engine within every GameObject with a script implementing it.
In details, it is available as a C# object, making the expressive power of a declarative language like Prolog to be available and utilized within Unity3D scripts, scenes and applications (videogames and simulations).

The KB can be created within every scripts as a usual C# object, in order to use the Prolog engine for its computing purpose, and it can be accessed in a straightforward way simply using methods and functionalities provided by UnityProlog library, which will be properly explained in next sections. The same KB concept is available to be used in 2 ways:

- `local KBs`, which could be private to every GameObject (KBs could be used by external agents or not, like normal objects, by reference), provide a Prolog engine to be loaded and executed locally, so no other object are able to interact with it
- `global KB`, that is a global object called "*GlobalKB*", available for every object willing to use a general Prolog engine with a general theory loaded

it is possible to provide each GameObject of a local, specific KB, in order to enable the Prolog engine to be exploited within the same object: in order to do this, UnityProlog system requires adding a KB Component to the target GameObject directly from Editor Mode (like any other GameObject's Component, e.g. Rigidbody or Collider, not by code).
However, this default mechanism allows only the creation of a KB that inherits from the global one, not providing the closed-world assumption needed for future developments, resulting in the GlobalKB's Prolog theory to be automatically inherited by every local KBs.
To solve this problem, while still allowing objects to interact with the GlobalKB for general info spreading, it is necessary to force each local KB to be created from scratch, not inheriting from the global one.
In this way, what is written in GlobalKB will not be visible also in all KBs and vice versa (this has been done providing programmers with a base script called AbstractLinda, explained below). When we want to include in our KBs some prewritten Prolog code (a standard Prolog theory), we can do it with code calling the `Consult` method on KB's objects (specifying the path of the Prolog file) and also in 2 ways, requiring no code to be written:

- regarding the GlobalKB, which is a Unity component that wraps the internal KnowledgeBase object representing Prolog equivalent of a namespace, all we have to do is to specify the theory path to the Source Files

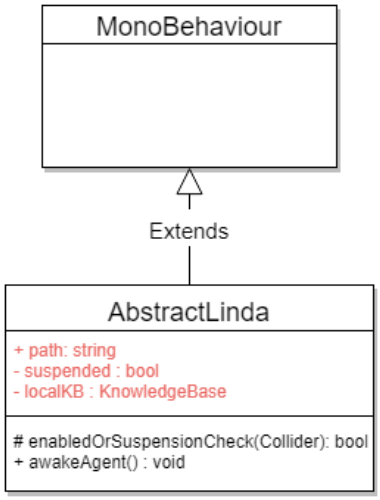


Figure 3.1: AbstractLinda script structure, which inherits from MonoBehaviour C# class (the base class from which every Unity script derives)

property of the KB Component in the Unity Editor (all files must be inside Assets folder of Unity3D project, be with extension *.prolog*, and the path must be complete, with root in Assets folder).

- regarding all local KBs it was implemented a base script, called `AbstractLinda`, which contains useful constructs to programmers meant to be easily used for all GameObjects with the need of Prolog engine enabled, which are a local KB object (generally available via API to external GameObjects) and methods useful for suspensive semantic and other stuff, explained in details in the next sections.

For now, let's say that the `AbstractLinda` script is a `MonoBehaviour` C# Script which enriches a base Unity one with Prolog support and other fundamental features, described later.
When a programmer wants to provide a GameObject support of local execution of Prolog code in a local KB, it must create a script which inherits from the AbstractLinda one (Figure 3.1).
In Listing 3.1 is provided the AbstractLinda C# script (full supporting suspension and trigger collider handling, explained during next sections and with domain specific examples).

```
public class AbstractLinda : MonoBehaviour {
   // Path to KB, useful only for SituatedKB scripts.
   public string path = "blabla.prolog";
   // Bool value representing whether the gameobject is suspended in some suspensive Linda
        call or not.
   bool suspended = false;

   // The local KB, can be exchanged among GOs.
   private KnowledgeBase localKB = new KnowledgeBase("",null,null);

   // Gets or sets the local KB.
   public KnowledgeBase LocalKB {
      get {
         return localKB;
      }
      set {
         localKB = value;
      }
   }

   // Gets or sets a value indicating whether this <see cref="AbstractLinda"/> is suspended.
   public bool Suspended {
      get {
         return suspended;
      }
      set {
         suspended = value;
      }
   }

   //protection against external and wrong awakening calls or unhandled trigger events
   protected bool enabledOrSuspensionCheck(Collider coll){
      if (!enabled || suspended) {
         return true;
      }
      return false;
   }

   // Awakes the agent suspended on some tuple.
   public void awakeAgent(){
      if(suspended){
         LindaLibrary.AwakeAgent (this);
      }
   }

}
```

Listing 3.1: AbstractLinda C# Script, the base one supporting local execution of Prolog engine and awakening from Linda suspensive calls: every Script which needs to exploit these funcionalities must inherit from the AbstractLinda one.

3.2.1.2 UnityProlog's constructs: Structures, LogicVariables, ISOPrologReader

Unity3D's main workflow (as any other game engine and videogame product) consists of calling a specific set of methods in every active script present in the current scene, which means only those attached to an active GameObject.
In fact, it follows an update loop structure, in which there exists some predefined callback methods (like *Update()*, *FixedUpdate()*, *LateUpdate()*, and so on) which are (if implemented) sequentially called every frame, so we won't be able to let the Prolog engine to run indefinitely because it will block the main loop bringing low performances.
The UnityProlog internal structure makes possible to periodically call Prolog code within these base methods as a usual object method call, allowing the Prolog engine to only run when needed.

Generally speaking, each Prolog concept is directly mapped into CLR language, but since strings and bools are the actual CLR types, the other types (integer, float, char, etc...) must be properly casted when used.
The Unity-Prolog interoperation for both C# concepts/Prolog entities and methods allow computation to be carried out in a twofold way, choosing the most suitable level of abstraction while designing and programming the system and switching between object-oriented style and declarative style.
For the interaction model integration, it was adopted the C# API Library in order to give programmers the right means at the right abstraction level while delegating to the Prolog engine the interaction and coordination core.

So, when integrating Unity3D and Prolog comes the necessity to have a C# representation of the relevant Prolog entities, including methods and standard data objects in order to make possible the exploitation of UnityProlog resources.
All possible data objects could be obtained using C# `Structure` objects: they are a representation of a term to be inserted into queries as parameters or return types, so they are a fundamental brick of the Unity-Prolog communication pattern. For example, Structures could be used in a query to

represent the fact to be unified, a functor with some arguments, and so on, as shown in Listing 3.2. It is possible to pass any CLR objects as arguments to Prolog code. A Structure could have another Structure as argument, enabling Compound terms to be passed in the query.

```
//the Tuplespace with Linda support
KnowledgeBase kb = new KnowledgeBase("linda",gameObject,null);

//struct0 is the tuple ‘‘pippo’’
Structure struct0 = new Structure ("pippo");

//struct1 is the tuple ‘‘test(pippo)’’
Structure struct1 = new Structure ("test","pippo");

//struct2 is the tuple ‘‘test(2)’’
Structure struct2 = new Structure ("test",2);

//struct3 is the tuple ‘‘test(hello(3),seven)’’
Structure struct3 = new Structure ("test",new Structure (’’hello’’,3),’’seven’’);
```

Listing 3.2: KnowledgeBase usage with Structures, enabling Prolog predicates to be called

Prolog variables are mapped into C# `LogicVariables` objects, each identified with a string: variables must be at least with the first char in upper-case, as the standard Prolog variable requires. LogicVariables are meant to be used with Structures in unification queries, in which the variable will be bounded with the actual term unified by the Prolog engine (Listing 3.3).

```
//the Tuplespace with Linda support
KnowledgeBase kb = new KnowledgeBase("linda",gameObject,null);

//struct1 is the tuple ‘‘pippo(X)’’ to be unified
LogicVariable x = new LogicVariable ("X");
Structure struct1 = new Structure ("pippo", x);

//struct2 is the tuple ‘‘test(follow,Y)’’ to be unified
LogicVariable y = new LogicVariable ("Y");
Structure struct2 = new Structure ("test", Symbol.Intern ("follow"), y);
```

Listing 3.3: KnowledgeBase usage with Structures and LogicalVariables, in order to use Prolog engine for template matching and unification

3.2.1.3 Unity-Prolog interactions

Interacting with Prolog from script files using C# code means calling specified methods on the KB object, so it is an object-oriented method call. There are several ways to ask something to Prolog engine, here follows the main used in this thesis:

- `bool IsTrue(object goal, object thisValue=null)`: returns *true* if *goal* is provable within the targeted KnowledgeBase, in this case *goal* is meant to be a Structure object which builds the query to be executed
- `object SolveFor(LogicVariable result, object goal, object thisValue, bool throwOnFailure=true)`: used for unification requests, in particular *goal* is meant to be a Structure object containing the LogicVariable *result*, so the unified *result* will be bounded to the passed LogicVariable. The function itself returns a boolean, meaning success or failure

Note that when predicates with compound terms as argument are intended to be called, it is necessary to manually build all Structures with a big effort. UnityProlog provides the class *ISOPrologReader* which allows the creation of a query in a seamless way, writing it as a normal string following the Prolog specified syntax, so that same queries can now be written in a simpler way (Listing 3.4).

```
//the Tuplespace with Linda support
KnowledgeBase kb = new KnowledgeBase("linda",gameObject,null);

//true if the KnowledgeBase has the fact agentX(follow,agentY)
var str = ISOPrologReader.Read ("agentX(follow,agentY).") as Structure;
kb.IsTrue (str);

//the bounded variable X will be available within its struct
var str2 = ISOPrologReader.Read ("X:agentX(follow,X).") as Structure;
kb.SolveFor (str2.Argument (0) as LogicVariable, str2.Argument (1), this);
var result = str2.Argument (0);
```

Listing 3.4: KnowledgeBase usage with ISOPrologReader object, avoiding Structure compositions and simplifying query creation

3.2.1.4 Prolog-Unity interactions

This version of Prolog includes some extensions to enable the interoperation with CLR code making it relatively transparent. If you want to call into Unity or your scripts you can do it directly from Prolog, without the necessity to write glue code or complex sentences.
Referring to Unity GameObjects and CLR types is simple: with the notation $name it is possible to obtain the reference of a GameObject with the name property.
For example, `$light` refers to the GameObject with name "light", while every object's name which starts with a capital letter or contains special chars like spaces, underscores, and so on, it should be enclosed in single quotes in order to don't cause problems within the Prolog engine.
Also, CLR types, classes, non-class types (value types, delegates) can also be named as stated before: `$String` stands for `class System.String`, `$'Camera'` means the Unity `Camera` class, `$'Vector3'` means Unity's `Vector3` type.
It is also allowed to use full names like `$'System.Object'`.
Most importantly, it is possible to access the object that currently is running the Prolog code but does not know its name: here UnityProlog provides the *indexical* feature, meaning that writing `$this` will refer to the component that called Prolog, and `$me` will be the reference to the GameObject that called it. This feature is useful in order to keep track of which GameObject has currently called a Prolog predicate, and it will be extremely useful for the suspensive semantic of `LINDA` primitives implementation, as explained in the next sections. Some caveats:

- the `$name` bindings are resolved during the game startup time, while code loads, which means that objects which are created at runtime will not be found if no GameObjects with that name has been created yet, but it is possible to call `GameObject.Find()` Unity built-in function within Prolog code in order to find a GameObject with a specific name:

  Z is $gameobject.find(|object_name|).

- *$me* and *$this* indexicals are bounded during load time and are indexical objects, not values

- objects with the same name will not be guaranteed to be correctly called
- case-insensitivity: Prolog does case-insensitive searching of object members, so two members whose names vary only by case will get Prolog confused

Accessing object properties and methods is as simple as referring to object's names: it is possible to use Prolog functional expressions like *'is/2'* to bind Prolog variables into values returned by some function calls:

- `X is $me.name` binds X to the GameObject's string name
- `Y is $me.getcomponent($'MyScript')` gets a particular component from the GameObject running at that time a Prolog query

Here are listed some of the useful standalone predicates to be used within Prolog theory and code (in order to enhance functional expressions with method calls and return values handling), they are only a subset, for other predicates please refer to the UnityProlog documentation:

- `property(*object,*property_name,>value).`, which unifies *value* with the value of object's property named *property_name*, succeeding exactly once or throwing an exception
- `call_method(*object,*method_and_args,>result).` , which calls the specified method on object with the specified arguments and unifies the return value with result, here an example of how to use it:

  ```
  call_method($this, 'TestM', A), log(A)).
  ```

  will call `TestM` function with no arguments on object bounded with *$this*, unifying *A* with the result and printing it on the Unity Console with command *log(A)*, while:

  ```
  call_method($this, 'TestMWithArgs'(2), A), log(A)).
  ```

  will call `TestMWithArgs` method with argument *2* (int) of the object bounded with *$this*, finally printing the result
- `is_class(?object,?class)`, True if `object` is of the specified `class`. If class is a subclass of Unity.Object, and object is uninstantiated, then it will enumerate objects of the specified type.

Chapter 4

MAS and Unity3D

This chapter is the core of the thesis, providing the necessary abstraction and interaction concepts design and formalization, based on fundamentals about `MAS`, `Prolog` and `Unity3D` introduced in Chapter 2 and 3.
Here, the main focus is on how to extend Unity3D with a new level of abstraction, providing logic programming support and trying to bridge the gap between MAS systems and Unity3D along with providing a tuplespace based interaction and communication model.
The organization of the chapter is as follows: Section 4.1 is the most important one, explaining idea and logic architecture behind the development of tuplespace based coordination and interaction in Unity3D based on Prolog theories.
Section 4.2 shows how the Prolog engine has been used for design and implementation of basic Linda primitives (as a background support, not visible to Unity3D designer and programmers) along with tuples and tuplespaces support.
Section 4.3 introduces Unity3D side of the development, presenting `LindaLibrary` API as low-level C# functionalities capable of directly exploiting Prolog background side of Linda implementation (realizing Unity3D-Prolog interoperation) and `LindaCoordinationUtilities` API as high-level functionalities exploiting `LindaLibrary` API to provide general communication primitives and more high-level interaction module.

4.1 Tuplespaces and LINDA in Unity3D: main idea

After enabling Unity3D with the possibility of using a Prolog engine aside its script control mechanism, the next step is to provide a coordination and interaction media in order to coordinate different entities.
This is a necessary step, especially when designing and developing collaborative applications and games, where a multiplicity of agents must interact with each other in order to achieve some goals inside the same physical environment, properly controlled by Unity game engine. For this purpose, coordination strategies and patterns could be as well designed, in order to provide different kinds of coordination models and technologies, since many of them have been proposed by literature and efficiently used in many research fields.
Indeed, a coordination model is required to handle the interaction space among components in multicomponent systems and, in the MAS context, they provide for the right metaphor and abstraction in order to properly build agent societies, where they play a key role.
For these reasons, the tuple-based coordination model originated from LINDA was the chosen one: a basic interaction protocol which provides features making its usage and implementation simple yet well-suited and powerful enough for our purpose, that is enabling a coordination model which could properly model the space of interaction among components, agents and Unity GameObjects.
In tuplebased models, agents interact with each other by exchanging tuples, which are collections of information items, chunks of info. Agents, objects and every GameObject should be able to communicate, synchronize and cooperate through tuplespaces by reading, storing, consuming tuples while associatively interacting with tuplespaces.
Benefits are:

- separation between computation and coordination, important for agent architecture design and keep computation and coordination in a distinct way [13]
- generative communication [12] and associative access to the interaction

space [5], decoupling agents both spatially and temporally, which is important regarding the possible unpredictable environment which could be designed thanks to Unity3D engine, while dealing with heterogeneous and dynamic information-based systems like the ones we want to develop

The design of the tuplebased interaction and coordination model follows 2 interoperating sides: *(i)* the Prolog one, in which tuples, tuplespaces and `LINDA` primitives are designed and implemented with respect to declarative syntax, using unification and backtracking, *(ii)* the Unity one.
So, in order to provide to C# and Unity developers of a simple yet powerful communication library it was implemented the `LindaLibrary` API, enabling the usage of the Prolog tuplespace communication model within Unity3D scripts and game logic.

4.1.1 System design

In order to design and implement a prototype library which could bridge the gap between Unity3D internal structure and Linda-based interaction and communication model, a proper system design must be clear to be the development base of new functionalities.
Basically, Unity3D is characterized to be an integrated development environment with a proper design model, which provides mechanisms and facilities to programmers and developers.
One of the most exploited yet powerful feature is the hierarchic GameObject composition, in which the most important concept in Unity3D Editor (GameObject) can be provided of properties (called Components) which contributes to the object complexity growth, like an empty container filled by ingredients. In particular, the GameObject-Component relationship is deep and important: every GameObject could have attached one or more Components, while each Component is able to provide and enable different kind of properties.
For example, the Transform Component defines the GameObject position, scale and rotation in the developed game world and it is critical to all GameObjects, the Rigidbody one enables the object to be subject to Physics

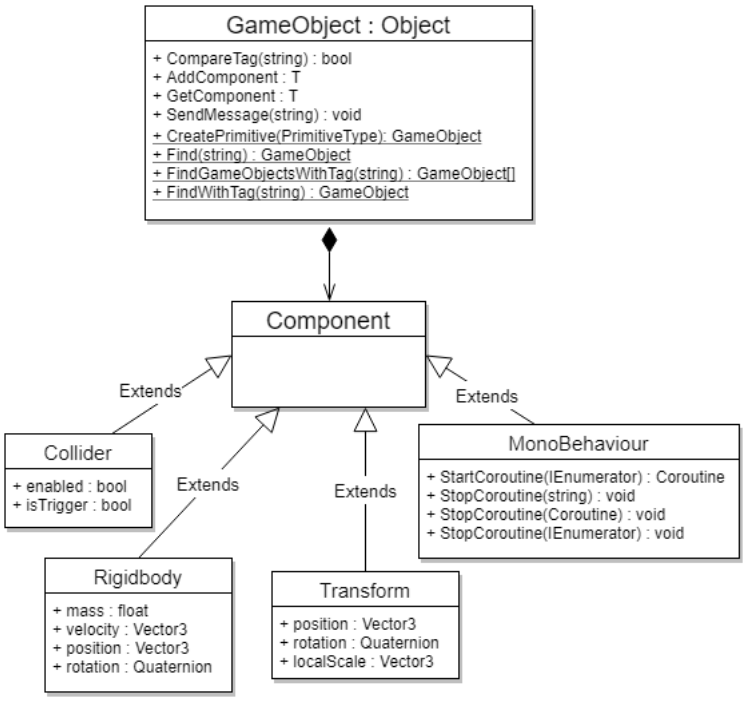


Figure 4.1: GameObject and Components hierarchy in Unity3D

forces, moving it in realistic ways, the Collider Component is responsible for enabling collision between GameObjects, the Script Component adds code and behavioural support, and so on, with a consequent increase and modification of objects complexity.
Unity3D acts as a middleware support, providing these features to be seen as simple and customizable building blocks: since everything inside Unity3D is a GameObject, from now on we will call "`agent`" only those provided with a C# Script Component (so with behavioural semantic and coding support, defining how they have to move, to react to events, and so on) and making a distinction between proactive and potentially movable entities and passive ones.

`MAS` vision of proactive agents living in a complex environment and interacting with each other is becoming a pervasive paradigm in design and implementation of complex software systems and applications. Interactions and communication models play a key role, managing the coordination among

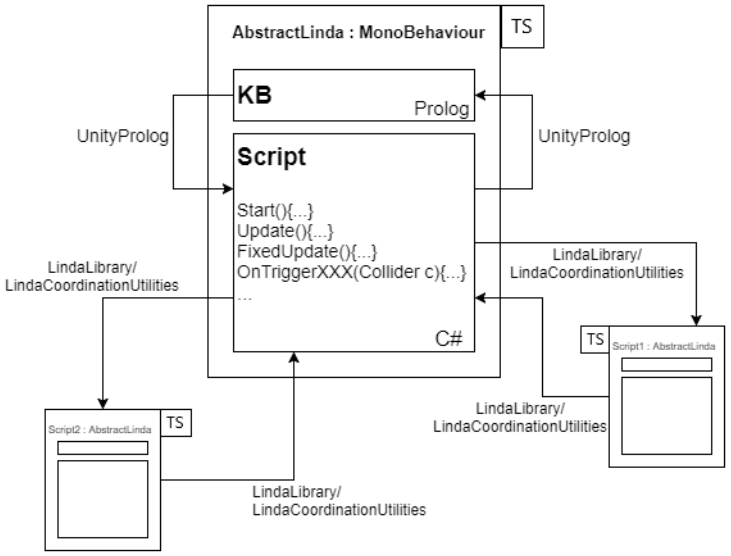


Figure 4.2: AbstractLinda Component in details: it inherits from MonoBehaviour class (the base one for all C# Script Components) and provides a local KB with Prolog support and basic functionalities. The Script section provides the usual functions of a standard C# Script.

components defining coordination media, coordinables and all the necessary abstractions at the right level of abstraction required to build agent societies. In particular, tuple-based coordination models are well-suited to be applied in heterogeneous and open systems, providing an associative mechanism to deal with dynamic, information-based applications.
In details, considering all previously stated system features and attributes, the system design idea is as follows (also depicted in Figure 4.3).
Since in Unity3D everything is a GameObject and every object could have a C# Script component defining its behavioural module, we say that if the C# Script Component inherits from a specified one (which enables Prolog support via UnityProlog concepts) is said to be a tuplespace itself, because the Prolog support brings the possibility to use Linda communication model. For this reason, the base C# Script is called `AbstractLinda` (Figure 4.2), and it is the one which enables Prolog support within a single GameObject, making the same object able to use the tuplespace based interaction model.

Moreover, since every GameObject with Prolog support is a tuplespace, the Linda communication model is empowered with all features provided by Unity3D, such as using Colliders to enable communication based on colli-

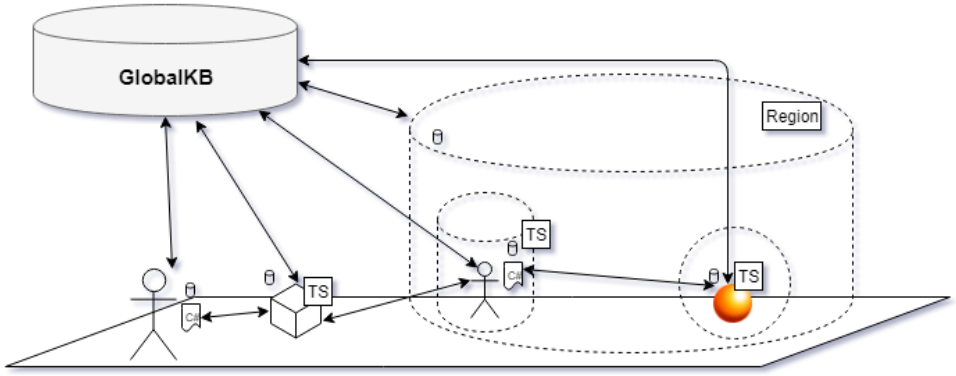


Figure 4.3: System design: every GameObject composed of a C# Script Component (and potentially with any other kind of Component) which inherits from the AbstractLinda C# class is a tuplespace itself (TS tag). A GlobalKB object is the global tuplespace which could be enabled or not, visible to every GameObject.

sion detection, with suspensive semantic.
With this concept, MAS and Unity3D are ready to be mapped, adopting the constructive coordination meta-model as explained in [4], as follows:

- *coordination media* resides in tuple spaces, which means that every GameObject could potentially be a tuplespace, with all features and functionalities of a regular, living, Unity3D GameObject
- *communication language* is the ordinary tuple, meant to be a collection of (possibly) heterogeneous information chunks, ready to be exploited within Unity3D using UnityProlog and a Prolog engine
- *coordination language*, meant to be a set of operations for insert, retrieve and read tuples from the space, available in Unity3D using Prolog via UnityProlog and, for Unity3D programmers and system developers, via developed communication libraries (`LindaLibrary` and `LindaCoordinationUtilities` API)

4.2 Prolog side background support: tuples, tuplespaces and Linda primitives

Although all new functionalities are important to be provided to programmers, it must be present a separation between Prolog background support

and C# static library and API.
This section explains how the `LINDA` primitives background support has been designed and developed, in order to provide `LINDA` functionalities to be exploited by C# libraries and API.
The Prolog side is not intended to be directly exploitable by Unity3D's programmers, but it will be used by C# API and wrapped into C# functions in a library in order to be used in a simpler way while programming C# Scripts (Section 4.3 describes the Unity3D side of the development).

For this thesis it was developed a `LINDA`-like tuplespace based coordination model using the previous integrated and analyzed Prolog engine as basic support: tuples are meant to be Prolog facts, exploiting the UnityProlog feature of interoperability with Unity3D C# space, while `LINDA` primitives are predicates with a specified arity and all Prolog queries are performed on a specified KB, described as follows:

- `out(T)` : inserts the tuple `T` into the targeted `TS`, it doesn't allow duplicates due to design motivations (Subsection 4.3.1.2 introduces the *out_d(T)* version, which allows duplicates)
- `in(T)` : retrieves the tuple `T` from the targeted `TS`, destroying it
- `rd(T)` : reads the tuple `T` from the targeted `TS`, not destructive

The implementation follows a standard declarative style, where each predicate is created with a specific purpose, enhanced by Prolog engine features like unification, backtracking, inference.
Prolog theories are meant to be both global shared tuplespaces, in order to globally coordinate agent activities (with the actual representation in the GlobalKB GameObject), and local yet shared memories used for data exchange in terms of tuples among agents and objects, enabling their synchronization and interaction control (actual representation: `AbstractLinda` script with the `localKB` knowledge base in the hosting GameObject).
Each tuple, as stated before, is saved as a Prolog ground fact, meaning that it could contains different data types while allowing Unity to correctly handle their diversity.

4.2.1 Suspensive semantic - Prolog support

Besides this very simple implementation comes the necessity to deal with suspensive semantic, as follows: if no tuples are found after `rd` or `in` calls, an agent may suspend its execution until some matching tuple is inserted into the target KB.
Since this semantic is not provided directly by the UnityProlog environment, it was decided to tackle some valuable features from both worlds (Unity and UnityProlog), in particular:

- from UnityProlog it is useful to exploit indexical variables, interoperability of methods/predicates for direct communication with Unity and the unification/inference power of the Prolog engine: for this purpose previous `LINDA` `rd` and `in` primitives are extended with the suspensive counterpart as follows:
  1. `in_susp(T)`
  2. `rd_susp(T)`
- from Unity it is exploited the *async/await* and *coroutine* mechanism in order to "simulate" a suspensive semantic (in particular, async/await construct is only available under the experimental version of the engine, so it is still unstable)

Prolog side, the idea is: in order to keep the engine aware of which agent is currently suspended on which `LINDA` primitive suspensive call, it is introduced a special tuple, with a specific syntax:

```
tuple_s(M,T,X)
```

The semantic of these 2 new predicates, enriched with the `tuple_s(T)` new ground fact, is as follows:

- if an agent calling `in_susp/rd_susp` founds the searched tuple, Prolog returns it after the unification process
- if an agent calling `in_susp/rd_susp` is looking for a tuple not yet available within the targeted KB, the Prolog engine saves a special tuple with a specific syntax:

```
tuple_s(Primitive,Tuple,$this),
```

where:

- *Primitive* is the suspended method (in/rd)
- *Tuple* is the searched tuple not yet available
- $*this* is the GameObject callee reference.

Then, the agent is suspended (the "*suspended*" meaning is explained in details in the next section) until an `out(Tuple)` call occurs, which means that the Prolog engine is able to retrieve which GameObject is currently awaiting for that specific tuple, eventually awakening it

This behaviour is made possible modifying the `out(T)` semantic and implementing new predicates, as follows:

- after an `out(T)` it could happen that some agent is waiting for that particular tuple, so the Prolog engine must search within the targeted KB if one or more `tuple_s(_,T,_)` are present, meaning that there exists suspended agent which must be awakened: this search is made by predicate `serveWaitQueue(T)`
- `serveWaitQueue` is responsible for `tuple_s(...)` search in the Prolog theory in order to retrieve all agents suspended on the tuple inserted just now, then it is time to awake them eventually destroying the tuple

This new mechanism brings several issues:

- fairness of `serveWaitQueue` predicate, meaning that if more than one agent is suspended on the same tuple, it must be awaken following the temporal property, like `FIFO` queues
- distinguish whether an agent is suspended after a rd or in call, because of possibly destructive semantic: the solution is to add the *Primitive* parameter to `tuple_s` fact in order to tag the special tuple with the current suspended method (along with the GameObject reference)
- wait queue handling, so implement `serveWaitQueue` knowing how to respond to different situations:
  - agents suspended on `rd` call, they must all be awakened after the right single `out(T)`
  - agents suspended on `in` call, only the first must be awakened (so the one which is waiting for a longer time, first come first served)

  - agents suspended on `rd` and in on the same tuple, which means that after an `out(T)` agents must be awakened following the temporal property until a `tuple_s(in,T,_)` is found: all agent suspended on rd calls must be awakened until the first agent suspended on in call occurred, while if other agents are suspended on the same tuple after reached the `tuple_s(in,T,_)`, they are not awaken until a new `out(T)` occurs (due to the temporal property)

- every awakening must be followed by the correspondent `tuple_s(M,T,R)` removal

To overcome these problems, the `serveWaitQueue` predicate were developed as follows: the idea is to handle `tuple_s` special Prolog facts as a `FIFO` queue of waiting agents, awakening the ones blocked in `rd` until the Prolog engine founds the first one waiting in `in`, ensuring fairness.
Listing 4.1 shows `LINDA` primitives implementation using Prolog: it is not intended to be directly modified by programmers, since it is exploited by `LindaLibrary` API, but it can potentially be extended and improved.

```
:- set_prolog_flag(unknown,fail). %closed world assumption

%example of ordinary tuples as ground facts
chop(0).
chop(1).
chop(2).

%example of special tuples, with different syntax than before
tuple_s(rd,test(2),$ref). %the agent with reference $ref is suspended on rd call searching
    test(2)

%adds tuples only if they are not already present and control the waitQueue with priorities
out(T) :- \+ T, assert(T), serveWaitQueue(T).

rd(T) :- T.
rd(T) :- \+ T, fail.

rd_susp(T) :- T.
rd_susp(T) :- \+ T, log(T), assert(tuple_s(rd,T,$this)), fail. % $me stands for the
    GameObject that called this method

in(T) :- T, retract(T). %retrieves the tuple T, destructive

in_susp(T) :- T, retract(T). %retrieves the tuple T, destructive
in_susp(T) :- assert(tuple_s(in,T,$this)), fail.
```

```
%handle the FIFO wait queue of suspended agents
serveWaitQueue(T) :- loopUntilIN(T,[],L), log(L), serveAgents(L).

%process special tuples with temporal property until tuple_s(in,_,_) is founded
loopUntilIN(T,Acc,L) :- \+ member(tuple_s(in,T,_),Acc), tuple_s(M,T,V),
    retract(tuple_s(M,T,V)), append(Acc,[tuple_s(M,T,V)],Y), loopUntilIN(T,Y,L).
loopUntilIN(_,L,L).

%serve suspended agents, awakening them
serveAgents([]).
serveAgents([H|T]) :- processAgent(H), serveAgents(T).

%awakening of the agents calling the ``awakeAgent'' method of AbstractLinda script
processAgent(tuple_s(rd,_,A)) :- call_method(A, 'awakeAgent', _).
processAgent(tuple_s(in,T,A)) :- retract(T), call_method(A, 'awakeAgent', _).
```

Listing 4.1: Linda primitives implementation, Prolog side (exploitable by Unity programmers using LindaLibrary and LindaCoordinationUtilities API)

4.3 Unity3D side: the LindaLibrary and LindaCoordinationUtilities API

What really is in the hands of Unity3D programmers and developers are C# libraries and API supporting the new coordination and interaction model: following this direction, this section engineers `LindaLibrary` and `LindaCoordinationUtilities` API on the basis of the previously described and implemented `LINDA` primitives using Prolog engine as basic support.
Each subsection is a logical step followed during API design. Subsection 4.3.1 describes the `LindaLibrary` API and how it provides all basic and low-level Linda primitives support using the Prolog side developed functionalities.
Subsection 4.3.2 is about the making of `LindaCoordinationUtilities` API as a new abstraction level from `LindaLibrary`, providing generic and high-level interaction and coordination functionalities.

4.3.1 LindaLibrary API: Linda primitives and suspensive semantic support

`LindaLibrary` provides all low-level functionalities interacting with the Prolog engine and previously written `LINDA` primitives: these C# functions are directly exploitable by Unity3D programmers while developing games or MAS-like systems.
Enabling the tuplebased interaction and communication model to be used for design and programming purpose within Unity3D means that it must be available to be exploited in C# scripts, so in those Components where all logic is written.
With this purpose it is implemented the `LindaLibrary API`, a static class full of utilities aiming to help the programmer to interact with the Prolog engine via `LINDA` primitives in a easy and straightforward way (Figure 4.4).
The library provides many features:

- it enables `LINDA in`, `rd` and `out` primitives to be used like C# methods, with parameter and return values, even with parametric variables to be unified by the Prolog engine, both on GlobalKB and localKBs
- provides the support to obtain the local KB of some targeted GameObject even only knowing its name, meaning that every GameObject with an AbstractLinda script as component is able to interact with each other via `LINDA` primitives and Prolog queries
- it is possible to properly set both GlobalKB and some targeted localKB with a Prolog theory, loading and unloading it both using the Editor and during runtime using the API (Prolog theories must be inside Assets/KB folder in order to be correctly loaded)
- it provides methods and utilities to implement a suspensive semantic when requested, both exploiting new async/await construct and using old but widely used coroutine mechanism
- thanks to UnityProlog features, it is possible to write searched tuples directly in a single string, without the need of composting Structures, LogicVariables and data types

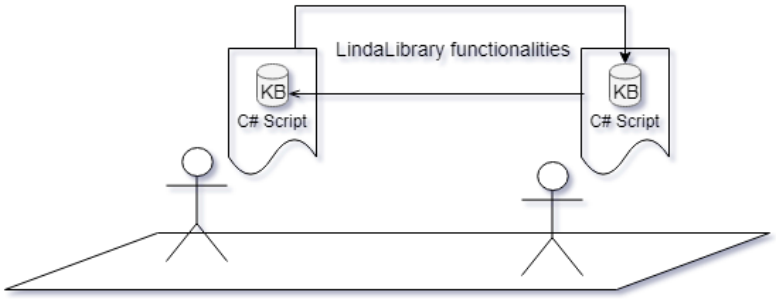


Figure 4.4: Straightforward agent interaction using LindaLibrary, as normal C# method calls used within Script Components

The main focus here is the implementation of a suspensive-like semantic in order to support the `LINDA` communication protocol: for this purpose it is exploited the Task-based Asynchronous Pattern (`TAP`), introduced with the experimental version of Unity3D but still quite unstable, and the `coroutine` built-in mechanism which enables heavy functions to be gradually executed with frame-by-frame basis, with no need to be finished within a single frame. So, the API enables `LINDA` primitives to be used like C# methods in a easy and quick way, simply interacting with KBs as standard objects: in this way, a GameObject can become a tuplespace carrier, each one with a possible different Prolog theory.
This means that everything we create and model inside a Unity scene (agents, normal objects, invisible objects, areas, and so on) could be a tuplespace with personal KB, tuples and logic, able to interact and coordinate with other GameObjects and KBs.

4.3.1.1 Suspensive semantic: Unity3D mechanisms

The core of the suspensive semantic implementation resides on Unity3D C# constructs and features, in particular using the Task-based Asynchronous Pattern (TAP) and the coroutines functionality, both chosen for their asynchronous nature to be exploited in a synchronous way, in particular:

- the `TAP` pattern is currently supported by the experimental version of Unity3D, so not particularly stable, but the support of .NET Framework 4.6 brings new features, such as *Tasks*, to be utilized within the

game engine, enabling new and interesting ways to tackle game design complexity by opening new implementation and project roads[1]

- the `coroutines` functionality, a built-in mechanism used to distribute heavy and complex computations along multiple frames, delaying them to the future

The API provides `LINDA` `rd` and `in` implemented both using *TAP* and *coroutines* (Listing 4.3), while Prolog side the `tuple_s` construct allows to keep track of agent suspended, method suspended and tuple requested. This decision was made with respect to the old but officially supported coroutine mechanism while looking to the future, to new features and allow using Tasks (which are still subjected to instability and runtime problems within Unity3D).
It is important to state here that the TAP adoption has been made in order to exploit some Task functionalities, but not Tasks themselves.
While executing an async method, the control flow is still on the Unity3D main-thread, meaning that although multi-threaded programming is potentially available to be used (thanks to .NET 4.6 support), it has not been exploited in this thesis, since it is still pretty unstable[2].
Here follow examples and semantic of *TAP* implementation and idea (depicted in Figure 4.5):

- the `LINDA` primitive call (either `rd` or `in`) starts as a normal one, asking to the Prolog engine to find a specific tuple `T`
- if `T` is found, the method returns positively, otherwise the agent must suspend its execution until someone makes a `out(T)` on the same KB of the awaited tuple
- the suspension occurs on a particular variable of class `TaskCompletionSource<T>`, which is an initially unbound variable that will be set to a value by the Prolog engine when the tuple becomes available.

[1] while in play mode, choosing to stop the simulation will not cancel previously launched async/await functions, resulting in background async calls still running even when the simulation has stopped, eventually terminating when trying to access to GameObjects not existing any more, but it surely is a problem due to the experimental version

[2] MissingReference errors when closing the game, which means that Unity is unable to terminate async methods running when shutting down the simulation

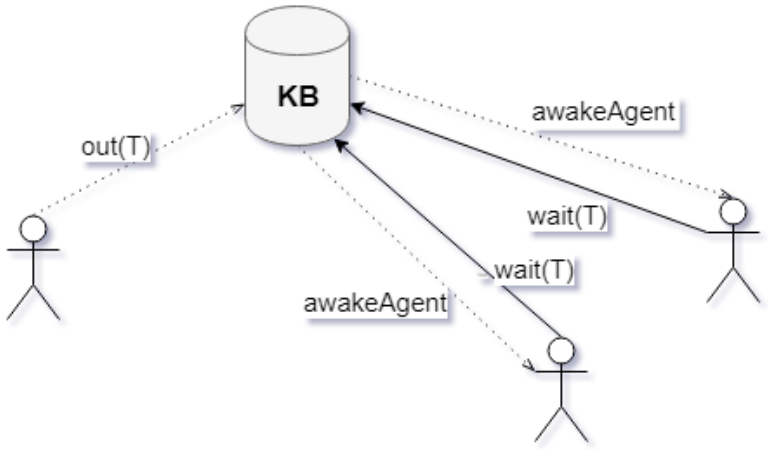


Figure 4.5: Suspensive semantic: agent performing a "*wait*" call (so, `LINDA in` and `rd` suspensive functions) on a specific tuplespace (the KB one) waiting for the tuple `T`; once arrived (a third agent performing a `LINDA out` of tuple `T`), both suspended agents are awakened via Prolog-LindaLibrary calls.

The `LINDA` primitive suspends the execution of the agent on this specific variable, until someone performs the awaited `out(T)`, which means that the Prolog engine will call the `awakeAgent` method of the base agent's script (one of the fundamental methods available from the `AbstractLinda` script), so the `TaskCompletionSource` variable will be set to a real value and the agent will resume its computation.

- in order to properly use the `LINDA` suspensive primitives, the programmer must follow some advise: calling a method using the TAP pattern means that the API must be executed with the keyword "`await`" before it, within a method tagged with the keyword "`async`". Otherwise, a Unity warning will occur saying that an asynchronous method is not currently awaited, so it is effectively executed in an asynchronous way, resulting in a constraint for the programmer

As far as coroutine method is concerned, its implementation is not much different from the TAP based one, only the *async/await* constructs are replaced by a `StartCoroutine(...)` and "*yield return null*", meaning that until the searched tuple is not available, the coroutine must suspend its execution repeating a statement the next frame until it finishes correctly. In the current version of the API, each agent can suspend its execution on 1 tuple and 1 method at a time: this constraint was necessary in order to awake agents re-

specting the temporal property, both saving agents timing on Prolog theory (with `tuple_s` special tuple) and on `LindaLibrary` static class (creating a Dictionary where each agent can be present only once), avoiding suspended agents to overwrite each other on the same TaskCompletionSource.
To overcome this limitation is required a more complex structure, in order to save agents multiple times and retrieving them in according to some temporal property saved (along with method name).

```
//Linda primitive examples
KnowledgeBase k = new Knowledgebase (‘‘linda’’,gameObject,null);
LindaLibrary.Linda_OUT ("agentX(knows,kungfu)",k);
...
LindaLibrary.Linda_RD ("agentX(knows,kungfu)",k)); //returns true
LindaLibrary.Linda_IN (‘‘X’’,"agentX(knows,X)",k)).toString(); //returns kungfu
LindaLibrary.Linda_RD ("agentX(knows,kungfu)",k)); //returns false
```

Listing 4.2: Example of a simple LindaLibrary functionalities usage

```
/*
   suspensive semantic example
*/

//TAP asynchronous version (async/await)

void Start(){
   AwaitSomething();
}

async Task AwaitSomething(){
   //will find a GameObject with name ‘‘Test’’ and retrieve its KB
   KnowledgeBase k = LindaLibrary.GetGameobjectLocalKB (GameObject.Find ("Test"));
   //execution suspended until someone inserts the tuple in the same targeted KB
   await LindaLibrary.Linda_RD_SUSP ("test(susp)",k2,this));
   ...
   //some agent performs: LindaLibrary.Linda_OUT ("test(susp)",k);
   ...
   //the agent awakes here, continuing the control flow
}

//coroutine version

void Start(){
   StartCoroutine(AwaitSomething());
}

IEnumerator AwaitSomething(){
   //will find a GameObject with name ‘‘Test’’ and retrieve its KB
   KnowledgeBase k = LindaLibrary.GetGameobjectLocalKB (GameObject.Find ("Test"));
```

```
    //execution suspended until someone inserts the tuple in the same targeted KB
    yield return LindaLibrary.Linda_RD_SUSP_COROUTINE ("test(susp)",k2,this));
    ...
    //some agent performs: LindaLibrary.Linda_OUT ("test(susp)",k);
    ...
    //the agent awakes here, continuing the control flow
}
```

Listing 4.3: Example of a simple LindaLibrary suspensive semantic usage within a target C# Script, both using TAP Asynchronous pattern and coroutines mechanism

4.3.1.2 Creating unpredictability: (random) primitives

A way to create non-deterministic behaviours is to give agents a choosing capability or allowing non-determinism when retrieving some information: this feature was captured and implemented in uniform-like `LINDA` primitives [20], `u_rd(T)`, `u_in(T)`. Randomization is important in game AI, but more generally in game design, MAS and simulations: although UnityProlog provides the ability to selectively declare randomizable predicates, which are then executed and solved in a non-deterministic mode, the Prolog engine tries to solve sub-goals in a random shuffled order, resulting in unpredictable behaviours. However, this is not what we want to achieve: from the interaction and coordination point of view, the idea is to let matched tuple to be retrieved in according to a uniform probability distribution when more than one match is found.

With this purpose, `u_rd(R,T)` and `u_rd(R,T)` predicates exploit the UnityProlog functionality of allowing multiple tuples to be present at the same time on the same KB, while providing meta-predicates capable of doing a random choice.

In more details, new predicates work as follows (Listing 4.4): they gather all matching tuples in a list, then using the Prolog meta-predicate *random_member* an element of the list is chosen non-deterministically.

This feature allows multiple scenarios, adding non-determinism could lead Prolog inference engine to take new logic or behavioural routes.

For example if multiple agents are waiting for the same tuple on the same method, one of them could be awaken non-deterministically and not following the standard temporal property, carrying new situations to happen in the Unity3D scene.

Besides these new functionalities, it is necessary to provide `out(T)` of a new semantic: as explained before, the base predicate does not allow tuple duplicates, which means that each tuple has the same probability to be chosen (due to implementation decisions, with the idea of make both semantics available).
For this reason it was introduced a new version of `out(T)` called `out_d(T)`, in which duplicates are allowed. However, this semantic could lead to unpredictable and wrong scenarios, for example agents not correctly awakened, but could also bring new features, which must be analyzed and evaluated regarding the communication model and what we want to achieve. This new feature is not officially supported (meaning that it has not been tested yet) but it is implemented and ready to be carefully used.

```
out_d(T) :- assert(T), serveWaitQueue(T). %creates a tuple even if already present

%uniform rd and in implementation: list of matching tuples, then random selection of one
    of them

u_rd(R,T) :- atomic(T), atom_string(X,T), findall(F,call(X,F), L), random_member(R,L).
u_rd(R,T) :- findall(R, T, L), random_member(R,L).
u_in(R,T) :- atomic(T), atom_string(X,T), findall(F,call(X,F), L), log(L), retract(T),
    random_member(R,L).
u_in(R,T) :- log(T), findall(R, T, L), retract(T), random_member(R,L).
```

Listing 4.4: Uniform-like Linda primitive implementation

4.3.2 High-level communication library: LindaCoordinationUtilities API

This section provides an informal introduction to the main elements of the `LindaCoordinationUtilities` API by showing how it enhances the basic `LINDA` model previously developed.
This new step is required in order to provide designers and programmers with a more abstract and high-level interaction and coordination utilities, with no need of knowing the specific `LINDA` syntax or primitives semantic. Here, this new abstraction level wraps the `LindaLibrary` one, enabling the possibility of general agent communication, providing new functionalities with a particular regard to the `SpaT` coordination model proposed in [30] (explained in

the next Chapter).
In particular, it is possible to create spatial tuples, tuplespaces living in the physical Unity3D environment, creating locations and regions exploiting some of the Unity3D features and using spatial primitives, looking forward to modern technologies such as augmented reality, directly supported by Unity3D game engine.

4.3.2.1 LindaCoordinationUtilities API: analysis

In order to build the API as a tool which could be more systematically used in potentially all projects developed using the Unity3D game engine with the need of interaction and coordination among GameObjects, we drew upon the enabling technologies and mechanism useful for this purpose.
At the core of `LindaCoordinationUtilities` library is the system of `LINDA` primitives introduced and explained in previous sections, which provide a suitable coordination environment, so importing and adapting the logic and mechanisms and properly mapped onto Unity3D constructs and patterns.
Moreover, a Unity3D programmer must be provided with API at the right abstraction level which could led to an easy system development, based on an utility API capable of sending messages of any type to any GameObject present in the scene, enabling interactions and information exchange.
The analysis of interaction and coordination models, along with Unity3D constructs, suggests the design and implementation of a wide range of C# functionalities properly tested within the game engine, exploitable in scripts Components which are the behavioural module of a GameObject.
Indeed, features provided by the `LindaCoordinationUtilities` library can potentially address applications targeting a wide range of scenarios (games, MAS simulations, augmented/mixed reality applications) with Unity3D fundamental support, which tackle the complexity of agent modelling and interaction models with many communication types (point-to-point, broadcast, multicast, collecting, and so on).

4.3.2.2 LindaCoordinationUtilities API: implementation

The API implementation followed a layered-like approach, in which single `LINDA` primitives are wrapped and composed into more general static functions, each of which implements functionality independent from the others and organized with general syntax.
Moreover, regarding to `MAS` and agent-based communication, it is important to state that `LindaCoordinationUtilities` provides and enhance agent interactions with a `Jason`-like syntax and semantic inspired by the `KQML` agent communication language [10]. Here, it has been adopted a specific syntax such as *ask*, *retrieve*, *tell*, to indicate more readable and self-explanatory communicative semantics, well suited to be used to coordinate agents and inspired by `AgentSpeak` [3].
The `LindaCoordinationUtilities` static library is divided into regions (exploiting the C# region tag), each one providing a specific functionality described as follows.

Ask

The `Ask` region (Listing 4.5) provides an high-level set of functionalities in order to read some tuple from a target KB: many different implementations have been made, targeting different KBs, such as the global one or a particular GameObject's one (if present, only knowing its name or its reference, feature directly supported by Unity3D).
Moreover, it is possible to ask for the same tuple on many KBs all over the scene or located in a particular area, ask for a particular tuple with a parameter to be unified, ask with suspensive semantic, and so on.
All ask implementations are based on `LINDA rd` primitive, interacting with GameObject's KBs but following a specific behaviour defined by C# script Component.

```
#region Ask

static bool Ask (string message) {...}
static bool Ask (string message, string name) {...}
static bool Ask (string message, GameObject target) {...}
static object AskParam (string var, string message) {...}
static object AskParam (string var, string message, string name) {...}
```

```
static object AskParam (string var, string message, GameObject name) {...}
async static Task<bool> AskSuspend (string message, string name, Component me) {...}
async static Task<bool> AskSuspend (string message, GameObject name, Component me) {...}
static IEnumerator AskSuspend_Coroutine (string message, string target, Component me) {...}
static IEnumerator AskSuspend_Coroutine (string message, GameObject target, Component me)
    {...}
static bool AskAll (string message, string tag) {...}

#endregion
```

Listing 4.5: LindaCoordinationUtilities Ask region

Tell

The `Tell` region (Listing 4.6) offers functionalities to write some tuple message into KBs: it could be inserted in the GlobalKB or on GameObject's KBs as well, targeting the GameObject reference or searching for a specific object by name (feature provided by Unity3D).
Tell functionality is critical when dealing with suspensive semantic: it is responsible of awakening agents if the right tuple is going to be inserted in some specific KBs: it is a function which diffuses information away from a source object while populating other KBs of knowledge.

```
#region Tell

static bool Tell (string message) {...}
static bool Tell (string message, string name) {...}
static bool Tell (string message, GameObject name) {...}

#endregion
```

Listing 4.6: LindaCoordinationUtilities Tell region

Retrieve

The `Retrieve` region (Listing 4.7) is composed of many different high-level functionalities similar to the ask region ones, but with the significant semantic difference that all searched (and found) tuples are destroyed from the source KBs when correctly retrieved.
Both ask and retrieve have to specify a template (a term with variables and literals starting with uppper-case letter) intended to be syntactically matched with the tuple being searched for: moreover, retrieve functions cover the same

functionality as the ask ones, so they provide suspensive semantic, target KBs from GameObject's reference or searching by GameObject's name, and so on.

```
#region Retrieve

static bool Retrieve (string message) {...}
static bool Retrieve (string message, string name) {...}
static bool Retrieve (string message, GameObject name) {...}
static object RetrieveParam (string var, string message) {...}
static object RetrieveParam (string var, string message, string name) {...}
static object RetrieveParam (string var, string message, GameObject name) {...}
async static Task<bool> RetrieveSuspend (string message, string name, Component me) {...}
async static Task<bool> RetrieveSuspend (string message, GameObject name, Component me)
    {...}
static IEnumerator RetrieveSuspend_Coroutine (string message, string name, Component me)
    {...}
static IEnumerator RetrieveSuspend_Coroutine (string message, GameObject name, Component
    me) {...}
static bool RetrieveAll (string message, string tag) {...}

#endregion
```

Listing 4.7: LindaCoordinationUtilities Retrieve region

4.3.2.3 LindaCoordinationUtilities API: towards exploiting Unity3D constructs

The main feature of working with a game engine like Unity3D is that it offers many and well supported functionalities, allowing the designer and programmer to build in a simplified and fast manner complex systems, applications and MAS-like simulations.
As an integrated development environment, Unity3D supports features like a visual editor (useful to build scenes), hierarchal object partitioning, gameplay preview, but what it still remains more exploitable for programmers, designers and engineers are base constructs (GameObjects, Components, Rigidbody, Collider, and so on) which are completely visible in details during project time (named Editor time).

Exploiting the Unity3D's GameObject composition features (everything within Unity3D scene is a GameObject, every object could be composed of many Component blocks) as stated in the previous chapter, the `LindaCoordinationUtilities` API follows this direction.

Interactions may occur with different means and flavours, for example sending a message to a remote object or communicating with some agent only when colliding with it, so the developed library is provided with the purpose of covering different types of interaction models exploiting Unity3D built-in features and constructs in an easy way.

Listing 4.8 shows a brief description of `LindaCoordinationUtilities` API signatures and functionalities:

```
static GameObject[] GetSituatedObjectsFromArea (Vector3 location, float radius, int
    maxNumColliders) {...}
static void BroadcastMessage (string message) {...}
static void BroadcastMessage (string message, string tag) {...}
static ReturnTypeKB AskMessageFromSituatedObjectTriggered (string message, Collider coll)
    {...}
static ReturnTypeKB RetrieveMessageFromSituatedObjectTriggered (string message, Collider
    coll) {...}
async static Task<ReturnTypeKB> AskMessageSuspendedFromSituatedObjectTriggered (string
    message, Collider coll, Component me, bool disableCollider = true) {...}
async static Task<ReturnTypeKB> RetrieveMessageSuspendedFromSituatedObjectTriggered
    (string message, Collider coll, Component me, bool disableCollider = true) {...}
static IEnumerator AskMessageSuspendedFromSituatedObjectTriggeredCoroutine (string
    message, Collider coll, Component me, bool disableCollider = true) {...}
static IEnumerator RetrieveMessageSuspendedFromSituatedObjectTriggeredCoroutine (string
    message, Collider coll, Component me, bool disableCollider = true) {...}
static bool MakeObjectSituated(GameObject go) {...}
static bool RemoveSituatednessFromObject(GameObject go) {...}
static GameObject IsObjectSituated(string name) {...}
static bool IsObjectSituated(GameObject target) {...}
```

Listing 4.8: LindaCoordinationUtilities functionalities able to exploit Unity3D's basic concepts and Components

This part of the library has been developed with the idea of exploiting base Unity3D constructs in order to make possible the usage of `LINDA` primitives and the interaction model dealing with simple yet meaningful situations, like the following:

- it is possible to provide GameObjects of multiple Collider components, defining an invisible shape, external from the body and not necessary with the exactly same shape and mesh, which enables physical collisions: if the object can be moved during gameplay, it is required to also attach a Rigidbody component, which enables Physics, forces applications and physical interactions with other GameObjects

- GameObjects with Colliders may exploit special functionalities within C# Scripts: the scripting system is able to detect when a collision occurs, in particular with the Trigger system (using *isTrigger* property of every Collider) the collider's behaviour is not like a solid object, meaning that during a collision the other Collider is allowed to pass through

- So, when a collider A enters the space of a second collider B, this one with *isTrigger* property enabled, the trigger will call the `OnTriggerEnter(...)` function on the Collider B GameObject's script. This chain of events is exploited for the creation of functionalities, like `AskMessageFromSituatedObjectTriggered(...)` and `RetrieveMessageFromSituatedObjectTriggered(...)`, where collided Objects are retrieved using the triggered collider, so they are methods to be used within `OnTriggerEnter(...)` functions.
  This opens up a variety of situations: there could be created worlds in which objects are only visible on fixed distances (Colliders are created specifying radius property, meaning that a radius of 3 units will provide a GameObject of a Collider visible only within 3 meters from its location), or collecting all GameObjects within a certain area (even relative to the actual position of the target object), or only when colliding with them. There could also be objects with a trigger collider functioning like areas, where objects may pass through them and being informed of something.
  Actually, it is possible to also exploit the `OnCollisionXXX(...)` methods, dealing with real collisions between Colliders and Rigidbodies, but they are not covered by this version of API.

- sometimes it could be useful to create objects with a base `LINDA` support, for example placeholder objects like chairs, tables or blackboards which do not need a complex behavioural script but may act just like containers with only `LINDA` primitives and Prolog support (so, with an initial empty theory).
  The API provides the `MakeObjectSituated(...)` function which enables a particular GameObject the possibility of running Prolog re-

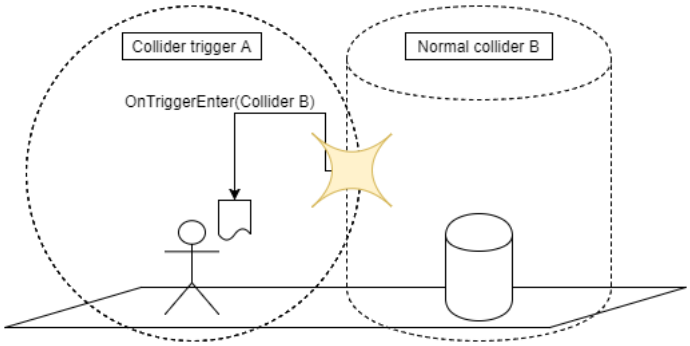


Figure 4.6: Working with colliders and triggers: when a collision between 2 Colliders of different GameObjects is detected, if one of the objects is provided of a Rigidbody Component, an event is raised and it can be captured by functions like OnTriggerEnter(...), ...

quests and `LINDA` primitives: in details, it is basically a second C# Script called `SituatedKB` attached to the same object which provides a local KB to be used. Moreover, the `RemoveSituatednessFromObject(...)` function removes that possibility from a certain GameObject. This enables the possibility to make every GameObject a tuplespace itself both in Editor Mode (design time) and during runtime, eventually removing this possibility restoring the old object properties.

Although these new functionalities enable a variety of interaction possibilities among objects and agents, functions involving Colliders, triggers and suspensive semantic are meant to be used in a specific manner and implemented following a particular technique:

- when designing an agent using the `OnTriggerEnter(...)` functionality in order to catch and deal with collisions, programmers must be aware that it is a periodically called function, meaning that it is executed every time a collision occurs within the Unity3D game engine: suspending the execution on these kind of functions is impossible (in a similar way like the Update() function, it is called always every frame from scratch)
- in order to "*simulate*" a suspension when calling LindaCoordinationUtilities suspensive API within methods like this, the `AbstractLinda` script was extended with new concepts and functionalities, explained as follows:

1. `bool suspended` : variable which has been added in order to check when a specified agent is suspended on something or not, useful for methods that handle collisions and to prevent external calls to the `AwakeAgent` function (provided by AbstractLinda script) to awake agents not currently suspended (check performed both by AbstractLinda and the `LindaLibrary` itself when attempting to awake some agent)
2. `bool enabledOrSuspensionCheck(Collider coll)` : method that controls if the current agent/object is suspended on something, this method must be called as a base one within the `OnTriggerXXX(...)`[3] personal implementation, in order to prevent the same agent to continually suspend itself on the same tuple (due to `OnTriggerXXX(...)` calling by the game engine explained before)
3. every `LindaCoordinationUtilities` suspensive function has been implemented with an optional parameter, *disableCollider* (true by default), which can be used for different semantics. The programmer can decide if the agent trigger collider must be disabled during its suspension or not, in order to avoid the `OnTriggerXXX(...)` method to be called for incoming collisions even if the agent is currently suspended, by simply setting the optional parameter (by default, trigger colliders are disabled during the whole suspension time).

   This semantic is **NOT** alternative to the suspended variable setting, but it is an additional semantic provided to the programmer, which can decide if its designed agent/object must be responsive to trigger collisions even when suspended

Listing 4.9 shows a snapshot of a typical `OnTriggerEnter(...)` implementation enriched with suspension check and `LindaCoordinationUtilities` call:

[3] the "`XXX`" string written on `OnTriggerXXX(...)` and `OnCollisionXXX(...)` methods means that different kind of functionalities are exploitable when using colliders, not only `OnTriggerEnter(...)` (capturing when the trigger Collider begins touching another one) but also `OnTriggerExit(...)`, `OnTriggerStay(...)`, and so on.

```
//inside a GameObject's C\# Script inheriting from AbstractLinda, with a trigger Collider

void Start(){...}

void Update(){...}

async void OnTriggerEnter(Collider coll) {

   //check if suspended on something
   if (base.enabledOrSuspensionCheck (coll)) {
      return;
   }

   //logic here

   //agent suspension with trigger colliders disabled
   await LindaCoordinationUtilities.RetrieveMessageSuspendedFromSituatedObjectTriggered
        ("allGood", coll, this);

   //done suspension: trigger colliders are back, continue doing things
}
```

Listing 4.9: OnTriggerEnter usage with suspensive semantic following the TAP asynchronous pattern

Chapter 5

API extension: Spatial Tuplespaces, Regions and BagOfTuples concepts

Unity3D is a game development environment and cross-platform engine which enables and makes it easy to develop complex systems and applications even supporting distribution and deployment to Augmented/Virtual/Mixed Reality platforms, including `Vuforia`[1], `Microsoft Hololens`[2], `Apple ARKit`[3].
Exploiting Unity3D functionalities, development patterns and internal constructs to create worlds, videogames, applications and MAS empowered with a proper coordination model, could allow research ideas to be actually realized within Unity3D.
With that idea, deeply influenced by the *SpaT* work in [30], comes a first extension of the `LindaCoordinationUtilities` library, in order to support creation, modelling and utilization of all coordination constructs (tuples, tuplespaces, ...) as a usual GameObject construct.
In this way, the same object may be accessed like any other object within the game engine and yet modelled like a real, living concept (so with a Rigidbody enabling Physics and forces calculation, Collider for collisions, Animation with animation support, and so on).

[1] https://www.vuforia.com/
[2] https://www.microsoft.com/it-it/hololens
[3] https://developer.apple.com/arkit/

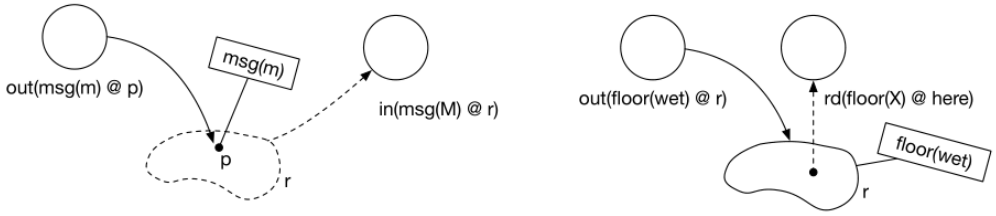


Figure 5.1: Spatial tuples and regions original idea, where agents insert and retrieve spatial tuples possibly defining specific locations and regions. Adapted from [30]

This is the direction: whoever wants to use `LindaCoordinationUtilities` API knows that it is possible to create `MAS` in which agents and objects are coordinated in the same way, both interacting with each other exchanging information between local KBs and coordinating with a GlobalKB or standard objects living in the scene (a table, a door, a region, whatever you like) in many different fashions (seeing them from a specified distance, when colliding with them, when someone disclose a GameObject existence, and so on).
As depicted in Figure 5.1, the original idea was dealing with spatially located tuples and tuplespaces, along with exploiting regions as a decoration of physical space with information: next section describes how this idea has been realized as a first prototype into concrete GameObjects living in a real, complex, designed Unity3D world.
Summing up, this chapter aims to introduce specific features, like spatial tuples and tuplespaces, regions, finally proposing a simple extension to the library with the `BagOfTuples` concept.

5.1 Spatial Tuplespaces and Regions: architecture and main concepts

Two concepts are being introduced here in order to better understand how to properly use these API, they are both C# classes and tags/layers:

- `SpatialTupleSpace`, a C# class representing a tuplespace which lives physically in the Unity3D scene: it could have several properties, such as a fixed shape (all Unity3D's *PrimitiveTypes* are allowed, e.g. square,

sphere, cylinder, cube, and so on), a trigger Collider, a Rigidbody (the tuplespace could be subject to Physics or not), an orientation, and so on.

The most important Component is a C# Script called `SituatedKB`, which enables a Prolog engine and `LINDA` utilities within it, with the KB as the Prolog core of the GameObject (so, the KB could be seen as the tuplespace itself).

Every SpatialTupleSpace must have tag or layer set to `SpatialTupleSpace`, explained later.

- `Region`, a C# class representing an external area, living and situated in a fixed location on the Unity3D scene, composed of a shape, location as the SpatialTupleSpace object, but it is provided with a a transparent, fully customizable Collider, providing radius and shape, and of a C# Script called `SituatedKB` (just like the SpatialTupleSpace object).

  This is the basic construct when is needed to create a Region, which could be crossed[4] by an agent in order to interact with it using the `LindaCoordinationUtilities` API with the `OnTriggerEnter(...)` method.

  This concept can be seen as a tuplespace which lives inside a spatial region, with fixed location, size and shape, and which could be a nest of coordination tuples, informing every object that crosses its trigger collider about something useful.

  As for the SpatialTupleSpace object, the Region object must be tagged and layered with `Region`, all reasons are explained next.

- "`SpatialTupleSpace`" and "`Region`" tag and layers, are a Unity3D construct useful to identify objects in the project ad to create groups that share some particular properties.

  Indeed, they are completely editable properties during Editor Mode and they are pervasively used within `LindaCoordinationUtilities` library in order to better and faster group similar objects, in particular:

  1. every time an object with an AbstractLinda script is created at runtime, it is provided by default of the correspondent tag or layer

[4] Trigger Colliders do the trick: colliders with that property active are crossable without Physics collision forces application.

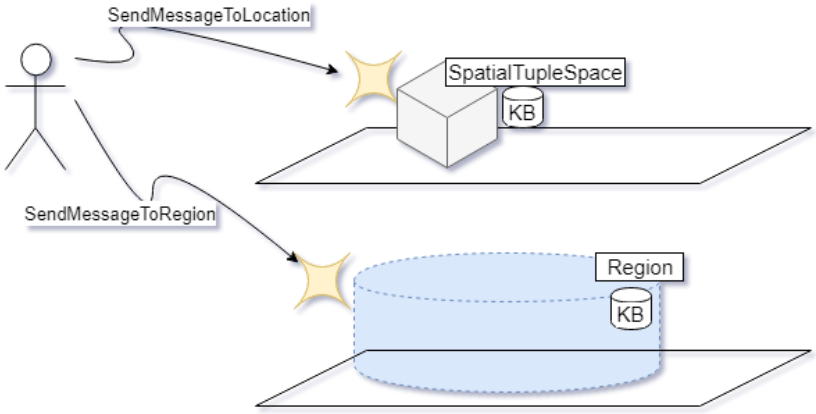


Figure 5.2: SpatialTupleSpaces and Region GameObjects creation using LindaCoordinationUtilities API: these functions create spatial tuplespaces as concrete physical objects.

("`SpatialTupleSpace`" if it is a spatial tuplespace, "`Region`" if it is a Region) in order to allow future functions to find them, but constraining the programmer to use these tag or layer values every time it need to create these GameObjects.
Moreover, "`SpatialTupleSpace`" and "`Region`" have to be manually added during Editor Time in order to let `LindaCoordinationUtilities` API to be properly used (otherwise, a system error will be thrown), while every GameObject must have a different name (in order to avoid the game engine to mistake objects)

2. tags and layers are used in functions where it is needed to search for GameObjects of a fixed category, restricting the searching field only to ones which could be targeted, so objects with Prolog support.
   In addition to that, tags an layers are very useful when searching for GameObjects using functions which need the Collider component (for example, the `GetSituatedObjectsFromArea(...)` method returns a fixed number of GameObjects within a certain Spherical Collider generated in a fixed point, returning only objects with layer `SpatialTupleSpace` or `Region`, limiting the game engine's searching field and increasing performance), so all GameObject with tag or layer not properly set to the previous ones will not be found.

This extension expands as well the interaction model and Unity3D's `LINDA` architecture: every object with a C# Script attached which extends the AbstractLinda script is provided with a Prolog engine and `LINDA` communication utilities (Prolog side), so it can be seen as a situated, spatial tuplespace moving and living in the scene with the GameObject to which is attached. Moreover, the fundamental construct is the KnowledgeBase (KB), provided in each AbstractLinda script as the GameObject's local one, which is moving with the object itself around the scene and could be targeted by `LindaLibrary` and `LindaCoordinationUtilities` functions, enabling a general tuplespace interaction model.
On top of that, every GameObject could be considered as the physical representation of a particular concept, w.r.t. the abstraction provided by Unity3D. Since everything is a GameObject within Unity3D scene modelling, and those which possess a Script Component inheriting from the AbstractLinda one are enabled with Prolog engine and tuplespace based interaction model, we can state that tuplespaces are seen like GameObjects themselves, so tuplespaces can react to Physics and collisions while living within a physical, evolving environment.

Listing 5.1 provides a snapshot of the implemented functions exploitable using `LindaCoordinationUtilities` API and SpatialTupleSpaces and Region class signatures, while Listing 5.2 shows simple examples of SpatialTupleSpaces and Regions creation:

```
//SpatialTupleSpace class
public SpatialTupleSpace(string name, Vector3 location, bool invisible, bool isTrigger,
    Vector3 scale, bool isRigid) {...}
//Region class
public Region (string name, Vector3 centre, Vector3 scale, PrimitiveType type, bool
    isTrigger, Quaternion rotation) {...}

//LindaCoordinationUtilities API for creating Spatial Tuplespaces and Regions
static bool SendMessageToLocation (string message, SpatialTupleSpace ts) {...}
static bool SendMessageToRegion (string message, Region reg) {...}
static bool SendMessageToAllObjectsInRegion (string message, Region reg, int num) {...}
static bool SendMessageToAllObjectsInRegion (string message, Vector3 centre, float radius,
    int num) {...}
static ReturnTypeKB SendMessageToRegionName (string message, string name) {...}
static void SendMessageToAllRegions (string message) {...}
```

```
static void SendMessageToAllRegionsName (string message, string name) {...}


//Examples

//creation of a SpatialTupleSpaces object with a single tuple localized in the current
    agent's spatial position, visible to everybody (with a mesh), not triggerable itself,
    with scale of (1,1,1), so 1m x 1m x 1m, with a Rigidbody (enabling Physics forces to
    be applied) and with a BoxCollider sized (3,3,3) (currently, only SpatialTupleSpaces
    with a boxed shape are supported)
SpatialTupleSpace sts = new
    SpatialTupleSpace("Location"+counter,transform.localPosition,false,Vector3.one,true,new
    Vector3 (3,3,3))
LindaCoordinationUtilities.SendMessageToLocation ("iWasHere("+this.name.ToLower
    ()+","+counter+")", sts);

//Creation of a Region named ``Caution'' in the current object's location, with a cubic
    shape sized (10,10,10) (so the region will be visible from that location to a 10
    meters distance), the region is triggerable (meaning that it could be trespassed on,
    with no collision forces) with a standard rotation
Region reg = new Region ("Caution", transform.position, new Vector3 (10f, 10f, 10f),
    PrimitiveType.Cube,true, Quaternion.identity);
LindaCoordinationUtilities.SendMessageToRegion ("stop("+counter+")",reg);
```

Listing 5.1: LindaCoordinationUtilities block making available SpatialTupleSpaces and Region creation and handling

```
//Examples

//creation of a SpatialTupleSpaces object with a single tuple localized in the current
    agent's spatial position, visible to everybody (with a mesh), not triggerable itself,
    with scale of (1,1,1), so 1m x 1m x 1m, with a Rigidbody (enabling Physics forces to
    be applied) and with a BoxCollider sized (3,3,3) (currently, only SpatialTupleSpaces
    with a boxed shape are supported)
SpatialTupleSpace sts = new
    SpatialTupleSpace("Location"+counter,transform.localPosition,false,Vector3.one,true,new
    Vector3 (3,3,3))
LindaCoordinationUtilities.SendMessageToLocation ("iWasHere("+this.name.ToLower
    ()+","+counter+")", sts);

//Creation of a Region named ``Caution'' in the current object's location, with a cubic
    shape sized (10,10,10) (so the region will be visible from that location to a 10
    meters distance), the region is triggerable (meaning that it could be trespassed on,
    with no collision forces) with a standard rotation
Region reg = new Region ("Caution", transform.position, new Vector3 (10f, 10f, 10f),
    PrimitiveType.Cube,true, Quaternion.identity);
LindaCoordinationUtilities.SendMessageToRegion ("stop("+counter+")",reg);
```

Listing 5.2: LindaCoordinationUtilities examples: creation of SpatialTupleSpaces and Regions as standard Unity3D objects

`SendMessageToLocation(string message, ...)` and `SendMessageToRegion(string message, ...)` will create the specified SpatialTupleSpace and Region GameObjects in the scene at runtime, eventually adding the "*message*" tuple in their local KBs: from now on, they will live in the simulation and could be exploited as a standard tuplespace, so using `LindaLibrary` tuplespace based interactions or with trigger colliders exploiting *OnTrigger* collisions on moving agents (Figure 5.2).
There also exist many different variants, making simpler to interact with objects, SpatialTupleSpaces and Regions within a specified area. In particular, the `SendMessageToAllObjectsInRegion(...)` method is provided with 2 variants, the first one makes possible to create a Region, collecting all SpatialTupleSpace GameObjects within it and performing a `LINDA` out on every KB found (targeting the Region too), while the second one will only target objects within a specified spherical region, not creating it.
Regions and SpatialTupleSpace GameObjects will be the physical representation of a tuplespace which lives and interacts with other objects in the created world. Both GameObjects could be created specifying an invisible mesh (so they will be invisible in the scene but still reachable via collider or name searching) and, using the full customizable C# classes `SpatialTupleSpace` and `Region`, it is possible to create objects only visible within a specified distance.
For example, setting the collider size to 3 units means that the specified GameObject will only be visible within that distance, meaning that an agent willing to interact with that object via `OnTriggerEnter(...)` method and `LindaCoordinationUtilities` API is required to get close to it, eventually overlapping both colliders.

5.2 BagOfTuples: idea and design

It may be possible that tuples share the same spatial location, for example they are inserted into the same GameObject, the same SpatialTupleSpace or Region, so multiple tuples could be created in the same place or attached to some agent in order to be moved with it: placing different tuples in the same spatial position (so, multiple overlapping GameObjects) is not possible to be

implemented in Unity3D, due to different reasons:

- it must be possible to add a variety of tuples in the same spatial location, so creating a multiplicity of GameObjects in the same place (even with a very small size, or without Collider and Rigidbody) causes Unity3D to not correctly handle all objects
- in order to create different objects in the same location they must be without Colliders or Rigidbody, so no Physics and no real forces: moreover, Colliders must all be triggers, in order to prevent Physics to throw them away, meaning that all triggering events generated by a multiplicity of objects in the same place could bring to performance problems

This problematic brings the idea of a new concept: `BagOfTuples`.
It consists of providing a target GameObject of a child object, called `BagOfTuples`, which is capable of carrying tuples as information chunks like a bag: child objects are complete GameObjects dependent on the parent one, meaning that their relative position, size, movement, and so on are strongly bound to the parent object (if the parent GameObject starts to move, all child objects will be moved with it).
BagOfTuples GameObject is provided with its own tag/layer named `BagOfTuples` in order to be properly retrieved by new `LindaCoordinationUtilities` API. Moreover, it could be provided with a Rigidbody and with a customizable trigger Collider, which will depend on the parent one (if the parent is provided with a Collider, the children one will be scaled accordingly: the API functions will provide the possibility of customizing this size).
This means that `BagOfTuples` is meant to be a basic tuplespace itself, allowing many features:

- it could be seen like a separate yet interactive tuplespace, which could potentially carry various different type of informations even not present within the parent GameObject's KB: it is like a bag of information to be carried and exchanged among agents
- this item is provided with its own consistency: it is possible to be managed during Editor Mode and runtime with new LindaCoordinationUtilities API, determining its Collider size and, since it has its own body,

could be subject to different situations (could be lost, exchanged, destroyed, discovered, and so on) like any other object

- BagOfTuples is capable of enabling Prolog and `LINDA` interaction model in GameObjects not capable of, expanding their possibilities for the amount of time they possess and handle it

This is a collection of information, like a bag or a sack, which could be used by agents with coordination or information purposes, and it could be attached and carried by GameObject even without letting it aware of that, so it might contains information and knowledge unknown even to the parent GameObject.
The new extension of `LindaCoordinationUtilities` API provides new functionalities in this direction: it will be possible to create BagOfTuples attaching them to GameObjects, know if a target object is provided with a BagOfTuple, interact with them using `LINDA` primitives, duplicate/acquire/send the entire BagOfTuple GameObject to a specified target/location. Moreover, agents suspended on the BagOfTuples KB will remain in that state even if the object is exchanged with other agents, letting them be awakened only at the right time.

```
//LindaCoordinationUtilities API for creating, exchanging and interacting with BagOfTuples

static GameObject IsObjectWithBagOfTuples(string name) {...}
static GameObject IsObjectWithBagOfTuples(GameObject target) {...}
static bool AttachBagOfTuplesToObject(string name, float radius = float.NaN) {...}
static bool AttachBagOfTuplesToObject(GameObject parent, float radius = float.NaN) {...}
static bool TellToObjectBag(string message, string parentName) {...}
static bool TellToObjectBag(string message, GameObject parentObject) {...}
static bool AskToObjectBag(string message, string parentName) {...}
static bool AskToObjectBag(string message, GameObject parentObject) {...}
static bool RetrieveFromObjectBag(string message, string parentName) {...}
static bool RetrieveFromObjectBag(string message, GameObject parentObject) {...}
static bool AcquireBagOfTuples(GameObject myself, string target) {...}
static bool AcquireBagOfTuples(GameObject myself, GameObject target) {...}
static bool SendBagOfTuples(GameObject bag, string name) {...}
static bool SendBagOfTuples(GameObject bag, GameObject target) {...}
```

Listing 5.3: LindaCoordinationUtilities block implementing BagOfTuples concept

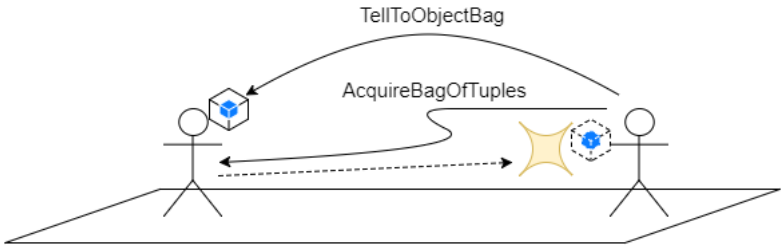


Figure 5.3: BagOfTuples idea and concept: the BagOfTuples GameObject (white cube with a blue one inside) is a normal object with a shape and a physical body, which could be attached to other GameObject and be carried all over the scene. Moreover, since it is a tuplespace itself, it will be interactible using LindaCoordinationUtilities specific API, and can be moved or cloned on every GameObject on the fly (duplicating its Knowledge Base with the same theory loaded, even suspended agents)

The new `LindaCoordinationUtilities` API extension features are the following (Listing 5.3):

- `IsObjectWithBagOfTuples(...)` checks if a target GameObject is provided with a BagOfTuples object, returning it if successful
- `AttachBagOfTuplesToObject(...)` creates a new BagOfTuples GameObject attached to a parent object with a Spherical Collider of the specified radius (by default is not provided, meaning that it will have the same radius as the parent one, if the parent GameObject is not with a Collider, the BagOfTuples one will be defined with a standard size of 5 meters)
- standard functions in order to interact with BagOfTuples via the tuplespace based communication model and `LINDA` primitives from `LindaLibrary` (`TellToObjectBag(...)`, `AskToObjectBag(...)`, `RetrieveFromObjectBag(...)`)
- suspensive semantic on BagOfTuples is not currently supported, but it could be achieved with standard `LindaCoordinationUtilities` API dealing with BagOfTuples' KB
- `AcquireBagOfTuples(...)` and `SendBagOfTuples(...)` are able to clone a BagOfTuples GameObject (along with its KB and all tuples within it), attaching it to the specified object (eventually failing if the target GameObject is already provided with a BagOfTuples child object)

Chapter 6

Case Studies

This chapter presents 2 different case studies, aiming to test and show how `LindaLibrary` and `LindaCoordinationUtilities` API can be exploited to reach desired interaction and coordination purposes.
In order to cover all API, 3 different application scenarios have been created:

- the first one is the classic and well-known *Dining Philosophers* coordination problem, where are tested `LindaLibrary` API and functionalities not focusing on concurrency problems (since they are absent in Unity3D, because its base single threaded structure, but this can change with C# 4.6 and Task support) but only on function validations. A special variation of this problem has been developed in order to also verify and validate some of the `LindaCoordinationUtilities` functionalities, in a way like the one stated in [30], with tuplespace and chairs spatially situated, placed within the tuplespace region and automatically retrieved (exploiting `LindaCoordinationUtilities` functions and Unity3D basic constructs) and assigned by chopstick pairs
- the second one is directly inspired by the *Breadcrumbs* simple example explained in [30], where an agent able to freely move in the scene implements the breadcrumbs pattern, depositing spatial tuples while moving and allowing other agents to retrieve that and follow the same path, testing and validating the `LindaCoordinationUtilities` API. Here, too, it has been implemented a second variation of the test, allowing the agent which makes breadcrumbs to create regions with warning tuples, where a second agent must stop whenever it collides with the same

region, finally resuming its follow actions once some “resume” message occurs in the created Region.

The goal in each scenario is to test and validate how developed API are able to correctly tackle complexity resulting into the expected behaviour.
Moreover, the main purpose is to demonstrate how the integration of a new interaction and communication model, along with the exploitation of Unity3D built-in features and concepts could be achieved and well-used with the developed libraries and MAS abstractions, expanding the standard interaction model already present in Unity3D with a new, more general level.
For these reasons, the design and implementation work was not focussed on concurrency and multi-threading problems, since Unity3D is entirely based on a single thread model, so no concurrency allowed until the new experimental version of the game engine, which brings .NET Framework 4.6 to be used with all significant features, such as the Task API.
Therefore, the multi-threaded and concurrent sides have not been taken into consideration for this case studies for time reasons and instability of the experimental version, but could be addressed by future investigations.

6.1 Experiment n°1: Dining Philosophers (LindaLibrary API test)

In this scenario, a simple multi-agent system modelling the *Dining Philosopher* problem has been set up [9], in order to test and evaluate the `LindaLibrary` functionalities. The setup is as follows: `N` philosopher agents (represented as spheres, but they could be any 3D or 2D assets) share `N chopsticks` and a `central bowl` (the main tuplespace), each philosopher either eats or thinks and, in order to eat, each one needs 2 chopsticks which are shared by 2 adjacent agents, so they have to be atomically acquired an released while thinking, ensuring fairness and avoiding deadlock problems.

The Unity3D scene was actually build in a very simple way, exploiting base 3D constructs like cube, spheres, cylinders, since the main objective is not the graphics or performance but rather verify correctness and ease of use of

`LindaLibrary` functionalities. Again, we exploit Unity3D's world fast prototyping, using built-in simple 3D objects but yet with all necessary features potentially enabled, such as NavMeshAgent in order to move the object, trigger Colliders, Rigidbody, and so on, leaving apart complex 3D prefabs, Animations, and so on.
In particular, philosophers are proper agents, able to move in the scene and they are represented as spheres able to change mesh color in according to their behavioural state (`think`, `wait`, `eat`), while chairs are represented with a cylindric shape and the central tuplespace is a bigger black sphere.

We do not focus here on deadlock analysis and issues by adopting the trivial solution of the resource hierarchy, where tuples are partially ordered and philosophers will always pick up the lower-numbered fork first, meaning that the last agent will choose the chopstick with the opposite order unlike all other philosophers (picking up first the right one instead the usual left one). Chopsticks are represented as ordinary tuples, so Prolog facts within the table tuplespace (in this scenario, it is the GlobalKB one), in the form of `chop(X)`, where `X` is the chopstick number (from `0` to `N-1`), all of them placed in the GlobalKB tuplespace. In particular, each chair is provided by an `Id`, from `0` to `N-1`, retrieved by the philosopher currently using it and each chair needs a chopstick pair in order to let its own philosopher to correctly eat (`chop(i)` represents the left chopstick of the `i`-th chair, while philosopher `i` needs `chop(i)` and `chop(i+1%N)` to eat).

The philosopher's stages are the following:

- each philosopher starts with a "`Think`" phase which lasts a random number of seconds, releasing all previously obtained chopstick pair to the tuplespace
- next, it will perform the "`Wait`" phase, acquiring left and right chopsticks with separate `LINDA in` primitives, suspending its execution if not found
- once acquired both of them, the "`Eat`" phase can start, where the philosopher eats for a random number of seconds

- once finished eating, the last phase called "`ReleasingChops`" will release both chopsticks on the GlobalKB tuplespace, eventually awakening agents currently suspended on some tuple
- the philosopher cycle restarts

Visually it is possible to understand which phase each philosopher is currently performing by the color assumed by the philosopher's mesh, in particular: **blue** represents the "`Think`" phase, **red** represents the "`Wait`" phase and green stands for the "`Eat`" phase. Next subsections explain in details both situated and spatial versions.

6.1.1 Situated version

The situated version of the Dining Philosophers scenario is developed using chairs as situated objects appointed to interact with the tuplespace, retrieving and releasing the needed 2 chopsticks necessary to the actual philosopher. In a real scenario, chopsticks are assigned based on the table position, so they are distributed to chairs following this property, so chairs are responsible to interact with the tuplespace, while philosophers only need to find an available chair.
Moreover, this situated approach makes the coordination abstraction aware of the surrounding space, in which the central tuplespace is able to interact with chairs and to provide the required chopsticks only relying on the actual place (and not talking directly with the philosopher with specific Ids).
The world design and development follows a very simple and basic implementation, but it can be made arbitrarily complex using 3D assets, specific agent prefabs and world customization using external softwares or directly exploiting Unity3D creation features, but for the API test and validation purpose it is not required.
In details, philosophers' and chairs' C# Script Components define their behavioural model and, for this particular scenario, it is not necessary to inherit from AbstractLinda Script (since it is not required for philosophers and chairs to be a tuplespace themselves, they interact with the global one, but the example code does it anyway). Moreover, philosophers deal with chairs GameObjects communicating their needs (`IWantToEat` and `ReleaseChops`),

while the real interaction and communication model performs in table-chair connection, handling chopsticks as tuple concepts among philosophers which suspend their execution if a needed tuple is not currently available.
This version has been developed in order to test and validate the `LindaLibrary API`, dealing with tuplebased coordination using `LINDA` primitives, a central Prolog KB and suspensive semantic.

```
public class SituatedPhilosopher : AbstractLinda {

   public int idPhilosopher;
   public int numPhilosophers;

   public Material thinkMaterial;
   public Material eatMaterial;
   public Material waitMaterial;

   private Chair chair;

   public int IdPhilosopher {
      get {
         return idPhilosopher;
      }
      set {
         idPhilosopher = value;
      }
   }

   public int NumPhilosophers {
      get {
         return numPhilosophers;
      }
   }

   // Use this for initialization
   void Start () {
      chair = GameObject.Find ("Chair" + IdPhilosopher).GetComponent<Chair> ();
      DoThings ();
   }

   // Update is called once per frame
   void Update () {
   }

   async void DoThings ()
   {
      while (true) {
         await Think ();
         await GetChops ();
         await Eat ();
         ReleaseChops ();
      }
   }

   public async Task Think ()
   {
      GetComponent<Renderer> ().material.color = thinkMaterial.color;
      await Task.Delay (Random.Range (3,10)*1000);
   }

   async Task GetChops ()
   {
      GetComponent<Renderer> ().material.color = waitMaterial.color;
      if (chair.IsAvailable) {
```

```
        chair.IsAvailable = false;
      }
      await chair.IWantToEat (this);
   }

   public async Task Eat ()
   {
      GetComponent<Renderer> ().material.color = eatMaterial.color;
      await Task.Delay (Random.Range (5,10)*1000);
   }

   private void ReleaseChops ()
   {
      chair.DoneEating ();
   }
}
```

Listing 6.1: SituatedPhilosopher C# Script Component code

```
public class Chair : AbstractLinda {

   private bool isAvailable;
   private int id;
   private int totalPhils;

   public Material availMaterial;
   public Material notAvailMaterial;

   public bool IsAvailable {
      get {
         return isAvailable;
      }
      set {
         isAvailable = value;
      }
   }

   public int Id {
      get {
         return id;
      }
      set {
         id = value;
      }
   }

   public async Task IWantToEat(SituatedPhilosopher phil){
      Id = phil.IdPhilosopher;
      totalPhils = phil.NumPhilosophers;
      print (Id + " : awaiting chop(" + Id + ")");
      await LindaLibrary.Linda_IN_SUSP (string.Format ("chop({0})", Id), this);
      print (Id + " : awaiting chop(" + (Id+1)%(totalPhils) + ")");
      await LindaLibrary.Linda_IN_SUSP (string.Format ("chop({0})", (Id+1)%(totalPhils)), this);
      print (Id+" DONE AWAITING");
   }

   public void DoneEating(){
      print (Id + " : RELEASING chop(" + Id + ")");
      LindaLibrary.Linda_OUT (string.Format ("chop({0})",Id));
      print (Id + " : RELEASING chop(" + (Id+1)%(totalPhils) + ")");
      LindaLibrary.Linda_OUT (string.Format ("chop({0})", (Id+1)%(totalPhils)));
      print (Id + " : DONE OUT");
      IsAvailable = true;
   }
}
```

Listing 6.2: Chair C# Script Component code

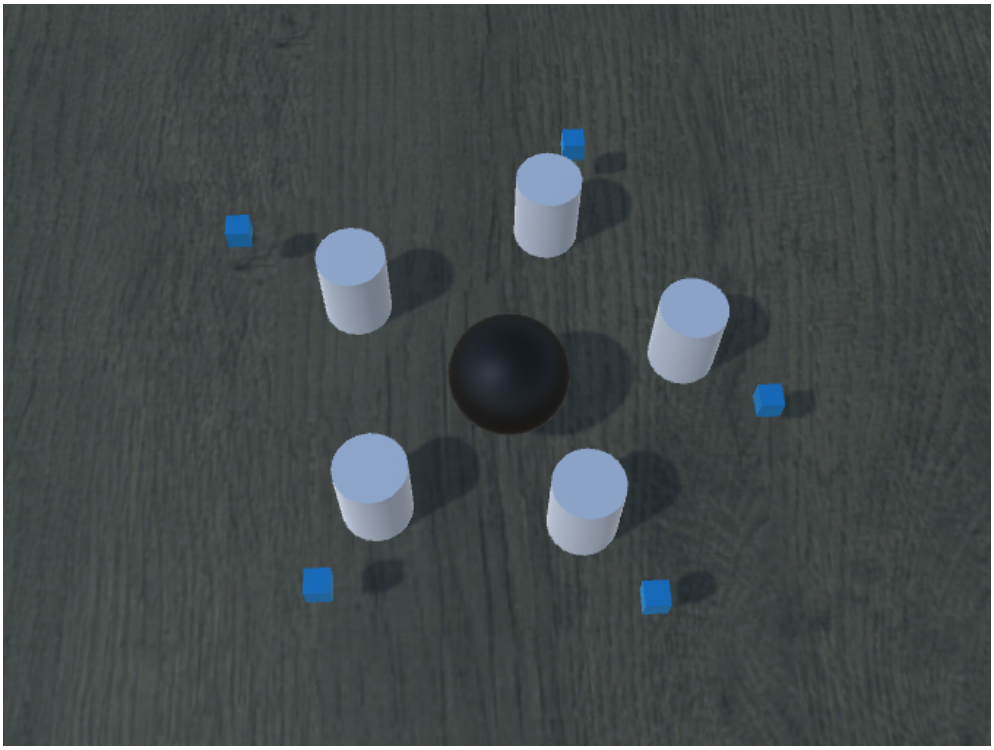


Figure 6.1: Unity3D situated Dining Philosophers: Start Phase

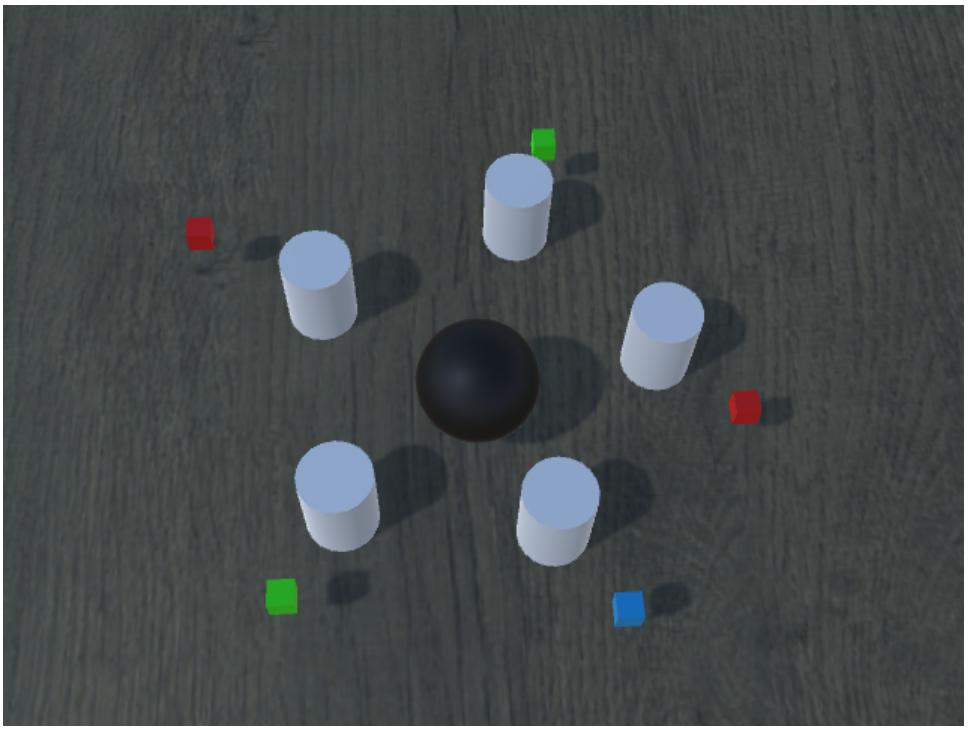


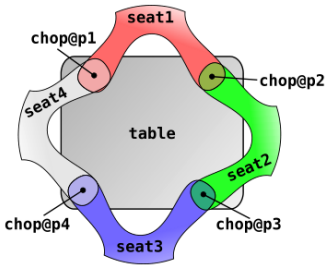
Figure 6.2: Unity3D situated Dining Philosophers: Eat Phase

6.1.2 Spatial version

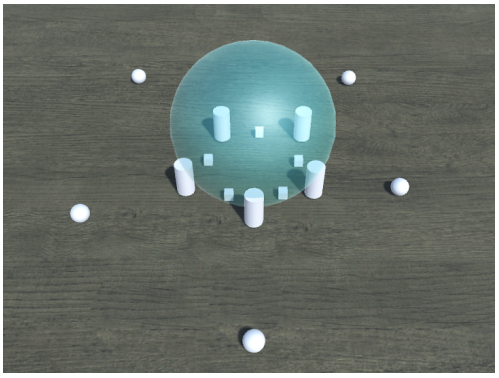
The spatial version of the Dining Philosophers test is directly inspired by [30] and it is developed in order to validate if the `LindaCoordinationUtilities API` support of Regions and spatial tuplespaces is correctly handled.
In particular, this scenario is designed differently from the situated one: since from the philosopher and chair behaviour sides all remains the same (again, philosophers are composed of the same Script Component version and interact with chairs, which are situated and responsible of the actual coordination), the scene is provided with additional GameObjects (representing chops as cubic objects) and the GlobalKB tuplespace is designed as a spherical Region with radius of 9 meters. These elements provide a new scenario, with new properties:

- the central tuplespace called `Table` is now a `Region`, with a definite `trigger` Collider of a specific size (spherical in this case) and it is situated, meaning that it is aware of which GameObjects are within its radius
- both Chops and Chairs are `SpatialTupleSpaces` GameObjects, which means that they are tuplespaces living in a physical environment: the real fact here is about all Chop GameObjects, which are indeed provided with the SpatialTupleSpaces tag and layer in order to be spatially situated and correctly found by `LindaCoordinationUtilities` searches

The spatial version make use of `LindaCoordinationUtilities` functionalities in order to deal with physical objects: the central tuplespace Region is aware of which and how many Chop GameObjects are available to be used by exploiting the `GetSituatedObjectsFromArea(...)` function, which returns a limited amount of objects with SpatialTupleSpaces or Region tags that are currently inside the specified area.
In this way, the aim is to limit the search to only SpatialTupleSpaces objects with the `Chop` name, while the Table Region is aware of how many chops are currently spread in the scene (eventually inserting a standard tuple in its tuplespace in the form of `chop(X)` for each retrieved Chop object).
Moreover, the same behaviour is followed by all Chair object, which are not



(a) *SpaT* Dining Philosophers idea, adapted from [30]



(b) Unity3D spatial Dining Philosophers: Start Phase

aware of which pair of chopstick are available next to them until via `GetSituatedObjectsFromArea(...)` call it retrieves which are the closest to be correctly used.

Thus, a philosopher which wants to eat from a specific chair is able to move to the chair location and interacts with it providing its needs just like the situated scenario described before.

Therefore, since every spatial tuple Chop is represented by a physical concept (and it is placed on the Table Region in order to be found by the central tuplespace, and shared by two adjacent seats because of its location is within the chairs' collider intersection), the philosopher would receive by its actual chair the chosen two chopstick spatial tuples (`chop(X)` and `chop((X+1)%N)`, because the chopstick positions spatially match with the searched ones.

The philosophers' behaviour is basically the same as the situated example explained before, but it exploits `LindaCoordinationUtilities` API version in order to use all necessary `LINDA` primitives with an higher-level of abstraction (using *Retrieve*, *Ask*, *Tell*, *RetrieveSuspend*, and so on).

Functions provided by `LindaCoordinationUtilities` are useful to reach different kind of behaviours and have been validated from the interaction and coordination point of view: considering that Unity3D basic communication and interaction mechanisms consist of fixed procedure call, this new abstraction level brings a new, more general interaction model extending Unity3D functionalities.

In the classic Unity3D mechanism, functions meant to be called as communi-

cation actions have to be previously written in every C# Script Component (the coordinable entity), so Script Components are able to communicate with each other only with procedure call mechanism.

```
public class SpatialTable : AbstractLinda
{
   private GameObject[] objs;

   // Use this for initialization
   void Start () {
      LindaLibrary.SetLocalKB (path,gameObject);
      SphereCollider c = gameObject.GetComponent<SphereCollider>();
      objs = LindaCoordinationUtilities.GetSituatedObjectsFromArea (transform.localPosition, c.bounds.extents.x, 25);
      for (int i = 0; i < objs.Length; i++) {
         if (objs[i].name.Contains("Chop")) {
            string thename = objs [i].name;
            LindaCoordinationUtilities.Tell ("chop(" + thename.Substring (thename.Length - 1) + ")",gameObject));
         }
      }
   }

   // Update is called once per frame
   void Update () {
   }
}
```

Listing 6.3: SpatialTable C# Script Component code

```
public class SpatialChair : AbstractLinda {

   private bool isAvailable;
   private int id;
   private int totalPhils;
   private GameObject[] objs;
   private GameObject table;

   private int chop1, chop2;
   private GameObject chop1go, chop2go;

   public Material availMaterial;
   public Material notAvailMaterial;

   public bool IsAvailable {
      get {
         return isAvailable;
      }
      set {
         isAvailable = value;
      }
   }

   public int Id {
      get {
         return id;
      }
      set {
         id = value;
      }
   }

   void Start(){
      table = GameObject.Find ("Table");
      CapsuleCollider c = gameObject.GetComponent<CapsuleCollider> ();
      objs = LindaCoordinationUtilities.GetSituatedObjectsFromArea (transform.localPosition, c.bounds.extents.x, 10);
      int count = 0;
      for (int i = 0; i < objs.Length; i++) {
```

```
        if (count == 2) {
          break;
        }
        if (objs[i].name.Contains("Chop")) {
          string thename = objs [i].name;
          if (count == 0) {
            chop1 = Convert.ToInt32(thename.Substring (thename.Length - 1));
            chop1go = objs [i];
            count++;
            continue;
          }
          if (count==1) {
            chop2 = Convert.ToInt32(thename.Substring (thename.Length - 1));
            chop2go = objs [i];
            count++;
          }

        }
      }
    }

    public async Task IWantToEat(SpatialPhilosopher phil){
      print (name + " : awaiting chop(" + chop1 + ")");
      await LindaCoordinationUtilities.RetrieveSuspend (string.Format ("chop({0})", chop1), table, this);
      chop1go.GetComponent<Renderer> ().material = Resources.Load ("Materials/Red") as Material;
      print (name + " : awaiting chop(" + chop2 + ")");
      await LindaCoordinationUtilities.RetrieveSuspend (string.Format ("chop({0})", chop2), table, this);
      chop2go.GetComponent<Renderer> ().material = Resources.Load ("Materials/Red") as Material;
      print (name + " DONE AWAITING");
    }

    public void DoneEating(){
      print (name + " : RELEASING chop(" + chop1 + ")");
      chop1go.GetComponent<Renderer> ().material = Resources.Load ("Materials/Green") as Material;
      LindaCoordinationUtilities.Tell (string.Format ("chop({0})", chop1), table);
      print (name + " : RELEASING chop(" + chop2 + ")");
      chop2go.GetComponent<Renderer> ().material = Resources.Load ("Materials/Green") as Material;
      LindaCoordinationUtilities.Tell (string.Format ("chop({0})", chop2), table);

      IsAvailable = true;
    }
}
```

Listing 6.4: SpatialChair C# Script Component code

Next test scenarios will be useful to test if MAS-like abstractions, along with high-level and more abstract coordination functionalities can be exploited and easily used.

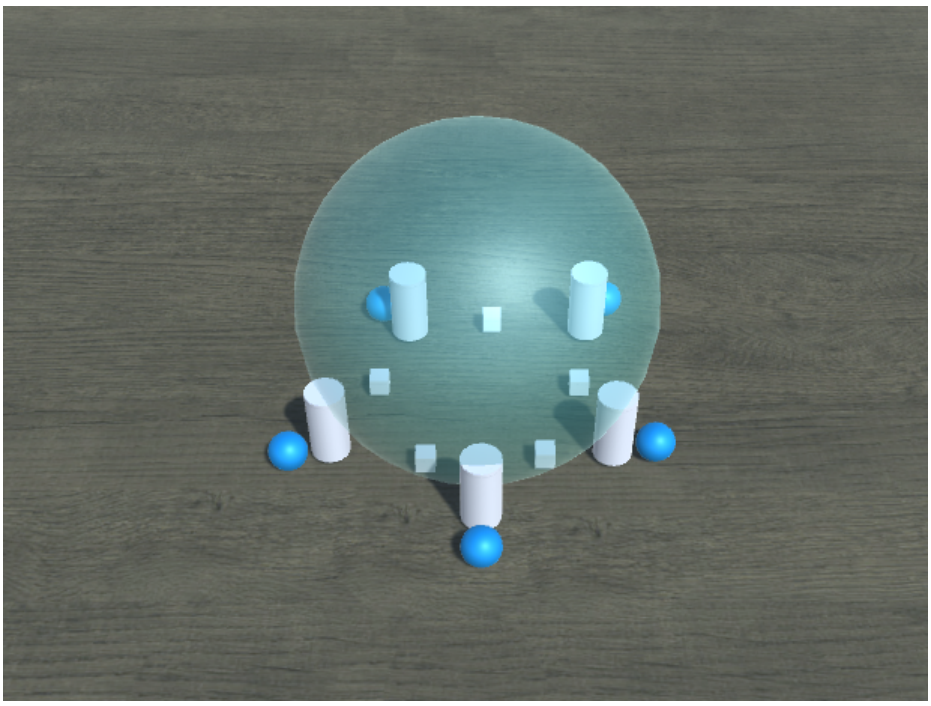


Figure 6.3: Unity3D spatial Dining Philosophers: first Think Phase

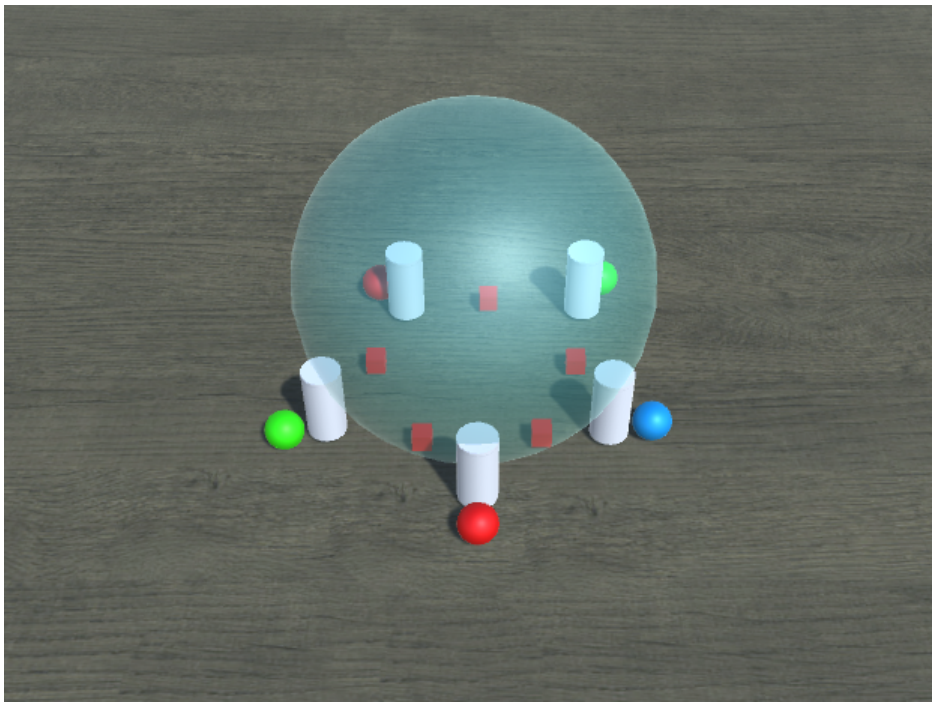


Figure 6.4: Unity3D spatial Dining Philosophers: Eat Phase

6.2 Experiment n°2: Breadcrumbs (LindaCoordinationUtilities API test)

As stated in the beginning of this chapter, the current scenario has been directly inspired by [30] example: an agent, able to move in the scene, is able to deposit spatial tuples as a physical object while moving with an operation like

`out(wasHere(me,C) @here)`,

resulting in the actual trajectory completely covered and observable by other agents, which could follow it simply observing the spatial distribution of `wasHere/2` tuples.
From this description, the *SpaT* extension of the basic tuple-based model lacks of a concrete representation of possible heterogeneous tuples and tuplespaces conceptually located in a physical space and able to move. The `LindaCoordinationUtilities` API comes in this direction, directly following the *SpaT* idea and, with Unity3D support and features, making possible to build MAS-like systems, letting the coordination primitives to behave accordingly to GameObject spatial properties and to be space-based and space-aware. In details, the current scenario proceeds as follows (also showed in Figure 6.5):

- the agent called `Hansel` is a standard GameObject with a spherical shape, able to move in the created arena (by attaching the *NavMeshAgent* Component), and every `N` seconds (where `N` is a *int* property directly customizable in Editor Mode) leaves a sign of its passage creating, in its current spatial location, a situated tuple by using the `SendMessageToLocation(...)` function and creating a SpatialTupleSpace GameObject (which is the physical conceptualization of a tuplespace) with a specific tuple in the form of

  `iWasHere(name,C)`,

  where *name* is the GameObject's name and *C* is a counter, incremented every time it performs this operation.
  So, the path followed by Hansel agent is described by spatial tuples left on it, like the breadcrumbs pattern (Listing 6.5).

- a second agent named `Follower` is appointed to follow Hansel's path by consuming the right spatial tuples (so, only those with `iWasHere(name,C)` tuple within its KB), by using its trigger Collider (in order to capture collisions with other object's Colliders) and the `OnTriggerEnter(...)` method while exploiting `LindaCoordinationUtilities` functions:

  ```
  AskMessageFromSituatedObjectTriggered (‘‘iWasHere(hansel,_)’’,coll)
  ```

  performs a `LINDA in` primitive on the collided SpatialTupleSpace, interacting with its KB searching for that particular tuple, if found the spatial tuple object is consumed and its location followed (the counter number obtainable from the tuple template using

  ```
  AskParam (‘‘X’’, ‘‘iWasHere(lindaagent,X)’’, coll.gameObject),
  ```

  marks temporarily the spatial tuples, following them in an ordered and temporal way) (Listing 6.6).

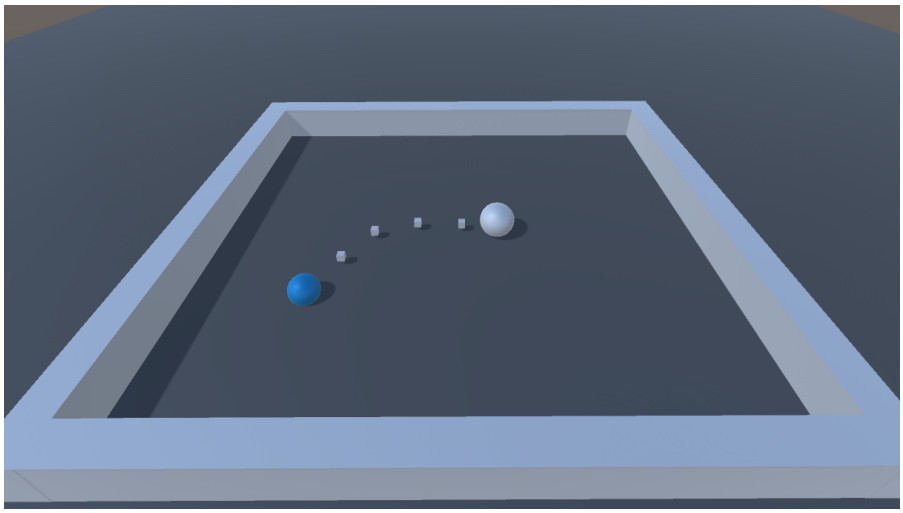


Figure 6.5: Unity3D Breadcrumbs example: the Hansel agent (blue sphere) is placing breadcrumbs as spatial tuplespaces with specific tuple (white little cubes), while the Follower agent (white sphere) is following it collecting all crumbs along the path

This simple example shows how `LindaCoordinationUtilities` API are useful to exploit tuples and tuplespaces as living information layers, with a concrete representation in the physical environment and enhancing the generative communication power with the breadcrumbs pattern adoption.

Moreover, Unity3D built-in constructs and features come in handy introducing new interaction patterns and models, like collision handling, GameObject searching, and so on, allowing the programmer and designer to exploit them in order to develop complex, MAS-like systems and pervasive computing scenarios supporting space-based coordination and interaction models.

```
public class Hansel : AbstractLinda {

   public float deltaTime;
   private int counter = 0;
   private Coroutine coroutine;
   private bool doing;

   // Use this for initialization, called once in the beginning
   void Start () {
      //starts the ‘‘creation of breadcrumbs’’ coroutine
      coroutine = StartCoroutine (DoThingsPeriodically ());
      doing = true;
   }

   // Update is called once per frame
   void Update () {
      //things to do once per frame
   }

   private IEnumerator DoThingsPeriodically ()
   {
      while (true) {
         yield return new WaitForSeconds (deltaTime);
         //creation of breadcrumbs as spatial tuplespaces with one initial tuple, with a cubic shape of standard
               dimensions (1x1x1)
         LindaCoordinationUtilities.SendMessageToLocation ("iWasHere("+this.name.ToLower ()+","+counter+")",
            new SpatialTupleSpace("Location"+counter,transform.localPosition,false,Vector3.one,true,new Vector3 (3,3,3)));
         counter++;
      }
   }
}
```

Listing 6.5: Hansel C# Script Component code

```
public class Follower : AbstractLinda {

   private NavMeshAgent agent;
   private int counter;
   private Collider destroyable;
   private bool once = true;

   // Use this for initialization
   void Start () {
      agent = gameObject.AddComponent<NavMeshAgent> ();
      agent.autoBraking = true;
   }

   // Update is called once per frame
   void Update () {

   }

   void OnTriggerEnter(Collider coll) {
      if (LindaCoordinationUtilities.AskMessageFromSituatedObjectTriggered ("iWasHere(lindaagent,_)",coll).Equals
            (ReturnTypeKB.True)) {
         var tupleCounter = Convert.ToInt32 (LindaCoordinationUtilities.AskParam ("X", "iWasHere(lindaagent,X)",
               coll.gameObject));
         destroyable = coll;
         Destroy (destroyable.gameObject,1.7f);
```

```
        if (tupleCounter > counter) {
          agent.destination = coll.transform.position;
          counter = tupleCounter;
        }
      }
    }
}
```

Listing 6.6: Follower C# Script Component code

The next version of the Breadcrumbs scenario shows how Regions and suspensive semantic API can be exploited.

6.2.1 Second version: Regions and suspension using trigger Colliders

Extending the Breadcrumbs experiment with a new scenario which could be useful to test and validate the second part of `LindaCoordinationUtilities` library means that it must provide and test other constructs, such as `Regions` and suspensive semantic, demonstrating and validating this side of the library.
In particular, the Breadcrumbs experiment has been extended and the behavioural dynamic is expected to be as follows (also depicted in Figure 6.6):

1. the Hansel-Follower breadcrumbs pattern is still used, so Follower agent's purpose is to follow the path of Hansel agent correctly consuming spatial tuples left on the way
2. a new mechanic has been added to Hansel agent extending its C# Script: while moving, agent Hansel is able to create a spatial Region any time, inserting a warning tuple in the form of

   `stop(hansel,X),`

   where X is the counter explained before (Listing 6.7)
3. the previously created cubic Region named `Caution` is placed on the Hansel agent path, meaning that the agent Follower will unavoidably collide with it[1]: since Follower is provided with a trigger Collider, the collision generates an event caught by the `OnTriggerEnter` function

[1] The collider size is a multiplicator of the actual GameObject's scale (defined in Transform Component), for instance a 3D object with spherical shape and scale of 1x1x1 units (3D Cartesian dimensions, x y z) provided by a spherical collider with a radius of 3 means that the actual object's body is 1x1x1 meters

(as shown in Listing 6.8) of Follower C# Script, which controls if the collided GameObject is a Region and if the stop tuple is present, the agent stops and waits for some good tuple on the same Region in order to resume its path

4. the awaited tuple is in the form of `allGood` and it is intended to be inserted into the Caution region by Hansel agent: once performed the `SendMessageToAllRegionsName(''allGood'', ''Caution'')`, the Follower agent will resume its previous path, while Hansel restarts producing breadcrumbs as spatial tuplespaces with specific tuples

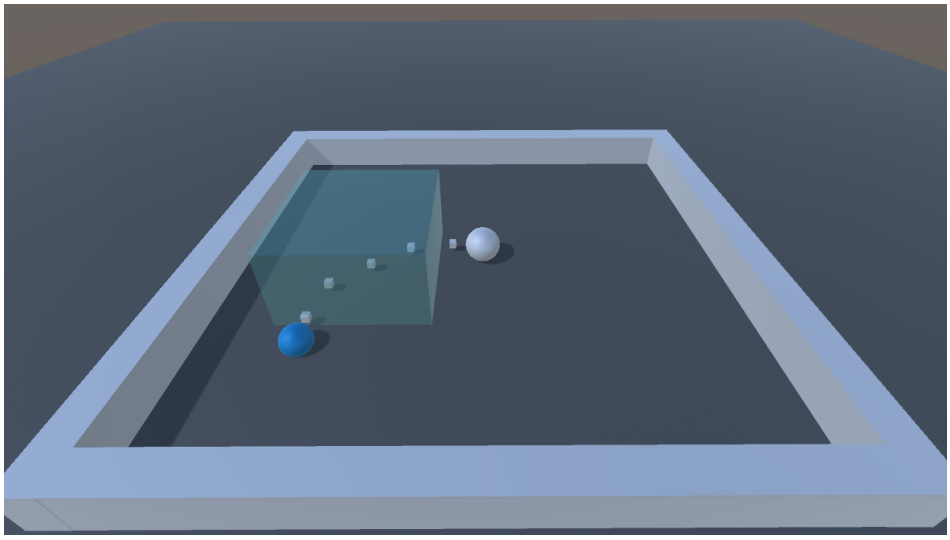


Figure 6.6: Unity3D Breadcrumbs extended example: the Hansel agent (blue sphere) leaves breadcrumbs (white little cubes) over its path, the Follower agent (white sphere) follows the path consuming breadcrumbs and stopping when reaching a `Caution` Region (light blue transparent cube), until a good tuple is inserted into the same region; finally, when the expected tuple is inserted into Region tuplespace, the Follower agent resumes from suspension, crossing the Region and resuming the breadcrumbs collection

This scenario shows how `LindaCoordinationUtilities` API can be successfully used to create Regions and spatial tuples of different dimensions and locations, as well as manipulate them in an easy way while designing and implementing the agent's behavioural module and semantic, enhancing Unity3D

scaled (if spherical, it will have a diameter of 1 meter, radius of 0,5 meter), while the Collider will have a radius of $1 * 3$ meters (so, following the simple multiplication $size * scale$), concluding in having a Collider with 3 meters radius (this is located to spherical scales/sizes: for example, a cubic object sized 1x1x1 units and with collider scaled 3x3x3 units means exactly what it says, a 1x1x1 meter body with a 3x3x3 meter collider).

basic functionalities with a new abstraction level provided to the Unity3D middleware. In particular, spatial and situated knowledge sharing is enabled by providing Regions to be tuplespaces themselves, with tuples as knowledge to be shared to every GameObject (with Prolog support) interacting with it or currently situated inside the Region influence, using `LindaCoordinationUtilities` API, as well as basic forms of awareness and spatial mutual exclusion, as following:

- *awareness*: Follower agent awaiting a specific tuple or stopping when entering or seeing a Region with danger messages, so new information to the agent which could change the behaviour introducing a new knowledge level needed to be faced
- *spatial mutual exclusion*: multiple Follower agents will stop and suspend their execution when bumping into the `Caution` Region, only to be all awakened by Hansel tuple insertion into the same region, like a checkpoint or a region lock

```
public class Hansel : AbstractLinda {

  public float deltaTime;
  private int counter = 0;
  private Coroutine coroutine;
  private bool doing;

  // Use this for initialization, called once in the beginning
  void Start () {
    //starts the ''creation of breadcrumbs'' coroutine
    coroutine = StartCoroutine (DoThingsPeriodically ());
    doing = true;
  }

  // Update is called once per frame
  void Update () {
    if (Input.GetKeyDown (KeyCode.Space)) {
      print ("SPACE, CREATING REGION HERE...");
      print ("STOP MAKING BREADCRUMBS");
      doing = false;
      StopCoroutine (coroutine);
      Region reg = new Region ("Caution", transform.position, new Vector3 (20f, 20f, 20f), PrimitiveType.Cube,true,
          Quaternion.identity);
      LindaCoordinationUtilities.SendMessageToRegion ("stop("+counter+")",reg);
    }
    if (Input.GetKeyDown (KeyCode.C)) {
      print ("C, SENDING OK MESSAGE TO REGION");
      LindaCoordinationUtilities.SendMessageToAllRegionsName ("allGood","Caution");
      coroutine = StartCoroutine (DoThingsPeriodically ());
    }
  }
  private IEnumerator DoThingsPeriodically () {
    while (true) {
      yield return new WaitForSeconds (deltaTime);
      //creation of breadcrumbs as spatial tuplespaces with one initial tuple, with a cubic shape of standard
          dimensions (1x1x1)
      LindaCoordinationUtilities.SendMessageToLocation ("iWasHere("+this.name.ToLower ()+","+counter+")",
        new SpatialTupleSpace("Location"+counter,transform.localPosition,false,Vector3.one,true,new Vector3 (3,3,3)));
```

```
        counter++;
      }
    }
}
```

Listing 6.7: Hansel C# Script Component code

```
public class Follower : AbstractLinda {

   private NavMeshAgent agent;
   private int counter;
   private Collider destroyable;
   private bool once = true;

   // Use this for initialization
   void Start () {
      agent = gameObject.AddComponent<NavMeshAgent> ();
      agent.autoBraking = true;
   }
   // Update is called once per frame
   void Update () { }

   //async keyword necessary when using TAP asynchronous pattern with LindaCoordinationUtilities suspensive API
   async void OnTriggerEnter(Collider coll) {
      if (base.enabledOrSuspensionCheck (coll)) {
         return;
      }
      if (coll.CompareTag ("Region") && coll.name.Equals ("Caution")) {
         print ("REGION: SHOULD I STOP?");
         if (LindaCoordinationUtilities.RetrieveMessageFromSituatedObjectTriggered ("stop(_)",coll).Equals
               (ReturnTypeKB.True)) {
            print ("STOPPING AND AWAITING A GOOD MESSAGE");
            NavMeshPath navpath = agent.path;
            agent.ResetPath ();
            await LindaCoordinationUtilities.RetrieveMessageSuspendedFromSituatedObjectTriggered ("allGood", coll, this);
            print ("DONE AWAITING GOOD MESSAGE");
            agent.SetPath (navpath);
         } else {
            print ("SHOULD I STOP? NOPE, CONTINUING BREADCRUMBS COLLECTION");
         }
      }
      if (LindaCoordinationUtilities.AskMessageFromSituatedObjectTriggered ("iWasHere(lindaagent,_)",coll).Equals
            (ReturnTypeKB.True)) {
         var tupleCounter = Convert.ToInt32 (LindaCoordinationUtilities.AskParam ("X", "iWasHere(lindaagent,X)",
               coll.gameObject));
         destroyable = coll;
         Destroy (destroyable.gameObject,1.7f);
         if (tupleCounter > counter) {
            agent.destination = coll.transform.position;
            counter = tupleCounter;
         }
      }
   }
}
```

Listing 6.8: Follower C# Script Component code

6.2.1.1 Further experiment: `BagOfTuples` interaction and cloning

As a further investigation, in order to test and validate the `BagOfTuples` functions of `LindaCoordinationUtilities` API it was decided to slightly modify the Breadcrumbs example as follows:

- when the Follower agent triggers the `Caution` Region collider and founds the stop tuple, it creates a `BagOfTuples` object on the Hansel agent (using the `LindaCoordinationUtilities` function `AttachBagOfTuplesToObject(...)`), then it stops its execution awaiting the good tuple on the Region field and a special tuple on the `BagOfTuples` just created
- the Hansel agent, when providing the `Caution` Region of the good tuple, performs a `TellToObjectBag(...)` call on its `BagOfTuples` (if attached, this call would have failed previously) inserting the special tuple awaited from Follower after `N` seconds, where `N` is a random number from 3 to 5 seconds
- in this way, Follower agent execution is restarted only after both tuples from different tuplespaces are correctly retrieved. If the same `BagOfTuples` GameObject were previously passed on a different object (attaching it to its surface, eventually destroying the original one), the Follower agent would still have to wait for the special tuple to be inserted to the same GameObject, meaning that `BagOfTuples` can carry every kind of information, both simple tuples and special tuples of suspended agents (with syntax `tuple_s(X,Y,$ref)`)

6.3 Results

Both case studies have been engineered in order to validate the library functionalities and to demonstrate improvements and benefits from their use.
Functions from `LindaLibrary` and `LindaCoordinationUtilities` API are indeed useful to provide a new way to deal with interactions and communication during Unity3D design and implementation, while exploiting some features to provide new interaction ways (like collisions) and a new abstraction level.
Moreover, in addition to simple validation tests and functionalities verifications, both libraries (also with Prolog and `LINDA` support) enhance Unity3D with a new abstraction level, offering general and novel interaction possibilities rather than the simple procedure-call mechanism.
Without `LindaLibrary` and `LindaCoordinationUtilities`, the only avail-

able mechanism in Unity3D to deal with interaction and communication is procedure-call: all functions intended to be called as communication acts must be written inside the target GameObject's Script Component. This mechanism is requested by Unity3D performance research, but it lacks of generality: the tuplespace based interaction model provides a new abstraction level, bringing to Unity3D properties and features not present before.
These solutions can be easily improved and extended, searching for performances and optimality, but they are representative of a first integration step between Game Engines and MAS, enabling a new interaction and coordination model to be used and closing the gap from the societal point of view.

Chapter 7

Conclusions and Future Work

Complex system engineering is going to be deeply impacted from coordination models, languages and technologies, in particular when talking about methodologies, abstraction levels and software processes as well.
Many real-world application scenarios are (and will be) subjected to technical challenges, where design and development of complex systems can be tackled by adopting a proper coordination and interaction model.
In this way, this dissertation surveyed the integration possibilities of interaction and coordination models with Game Engines, presenting two C# libraries, `LindaLibrary` and `LindaCoordinationUtilities` which can be taken as a introductory step in exploiting MAS abstractions and mechanisms within Unity3D.
Along with a simple Prolog development of Linda primitives, the Unity3D libraries bridge the gap between the theoretical MAS societal abstraction and the Game Engines world, providing a new interaction and coordination model to be directly exploitable in the construction of complex software systems and videogames.
Moreover, Unity3D is an IDE with lots of features and supported technologies, so general and complex systems/applications are enabled to be built: the `LindaLibrary` and `LindaCoordinationUtilities` libraries is organized and engineered around Unity3D functionalities, allowing tuplespace based model to feature a new abstraction level, bringing important properties of coordination and interaction models typical of MAS abstractions.

This is a prototype work, started with the purpose of investigate to what extent it is possible to support tuplespace based interaction in Unity3D and gone far beyond, so improvements and future works are surely possible and needed.
On top of that, refinements to libraries themselves and Prolog code: contents and organization are open to improvements and reshaping, as well as the code structure and, most importantly, a better use of Unity3D functionalities and Prolog integration, in order to improve performance and overall system organization.
Also, more complex examples and case scenarios are needed to be done, in order to analyse both libraries in terms of expressiveness, flexibility, efficacy and (most importantly) utility: it's indeed interesting to build applications using `LindaLibrary` and `LindaCoordinationUtilities` with modern technologies fully exploiting Unity3D features, such as augmented/physical reality, distributed systems with multiplayer support and immersive simulations, in order to verify how this work succeeded in its purpose to bring concrete functionalities and new ways of dealing with complexity in MAS development.

Ringraziamenti

Questo lavoro sancisce la conclusione del mio percorso accademico durato più di 5 anni, grazie al quale ho imparato tante cose ma, soprattutto, mi ha fatto crescere sia professionalmente che come uomo. Il periodo trascorso a Cesena è stato meraviglioso, sebbene difficile all'inizio: molta della mia crescita la devo a questa città, alle persone che ho incontrato quì, alle amicizie maturate e a tutte le sfide a cui sono stato sottoposto.
Inizio con il ringraziare i miei genitori, mia sorella e tutta la mia famiglia, la quale mi è sempre stata vicina e sostenuto in ogni momento, senza mai dubitare delle mie capacità e talvolta scontrandosi con il mio carattere: senza di voi non sarei quello che sono ora.

Ringrazio la mia ragazza Valentina, la quale da oltre 10 anni rappresenta il mio punto di riferimento, sempre pronta a dimostrarmi fiducia e a rassicurarmi nei momenti più difficili: si apre una nuova pagina, ma saremo ancora più uniti.

Ringrazio il prof. Andrea Omicini e il dott. Stefano Mariani per la fiducia e la disponibilità dimostratami lungo tutto il mio percorso accademico e soprattutto in questo progetto: la loro passione, interesse e professionalità sono stati fondamentali e coinvolgenti, le discussioni con loro sempre fonte di ispirazione e di miglioramento.

A tutti gli amici che ho conosciuto durante questi 2 anni di magistrale, e a tutti quelli che ormai fanno parte della mia vita da molto tempo, dedico questo traguardo: per chi c'è stato, per chi ci sarà, siete stati importanti e, se lo vorrete, lo sarete in futuro.

Si conclude quindi un capitolo importantissimo della mia vita, costellato da sfide e da soddisfazioni, quindi si apre una nuova pagina, tutta da scrivere: così com'è stato 5 anni fa, ora sarà tutto nuovo, ma ciò che sicuramente per me rimarrà invariato sarà la volontà di avere tutti voi ancora parte della mia vita, per cui grazie ancora a tutti con il cuore!

Bibliography

[1] Unity Manual: execution order of event functions. `https://docs.unity3d.com/Manual/ExecutionOrder.html`. Accessed: 2018-01-26.

[2] Blair, J. and Lin, F. (2011). An approach for integrating 3d virtual worlds with multiagent systems. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 580–585. IEEE.

[3] Bordini, R. H., Hübner, J. F., and Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*, volume 8. John Wiley & Sons.

[4] Ciancarini, P. (1996). Coordination models and languages as software integrators. *ACM Computing Surveys (CSUR)*, 28(2):300–302.

[5] Ciancarini, P. and Gelernter, D. (1992). A distributed programming environment based on logic tuple spaces. In *FGCS*, pages 926–933.

[6] Ciancarini, P., Omicini, A., and Zambonelli, F. (1999). Multiagent system engineering: The coordination viewpoint. In *International Workshop on Agent Theories, Architectures, and Languages*, pages 250–259. Springer.

[7] Denti, E. and Omicini, A. (1999). Engineering multi-agent systems in luce. In *Proceedings of the ICLP*, volume 99.

[8] Denti, E., Omicini, A., and Ricci, A. (2001). tuprolog: A light-weight prolog for internet applications and infrastructures. In *International Symposium on Practical Aspects of Declarative Languages*, pages 184–198. Springer.

[9] Dijkstra, E. W. (1968). Cooperating sequential processes. In *The origin of concurrent programming*, pages 65–138. Springer.

[10] Finin, T., Fritzson, R., McKay, D., and McEntire, R. (1994). Kqml as an agent communication language. In *Proceedings of the third international conference on Information and knowledge management*, pages 456–463. ACM.

[11] Freeman, E., Hupfer, S., and Arnold, K. (1999). *JavaSpaces principles, patterns, and practice*. Addison-Wesley Professional.

[12] Gelernter, D. (1985). Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112.

[13] Gelernter, D. and Carriero, N. (1992). Coordination languages and their significance. *Communications of the ACM*, 35(2):96.

[14] Indraprastha, A. and Shinozaki, M. (2009). The investigation on using unity3d game engine in urban design study. *Journal of ICT Research and Applications*, 3(1):1–18.

[15] Kaminka, G. A., Veloso, M. M., Schaffer, S., Sollitto, C., Adobbati, R., Marshall, A. N., Scholer, A., and Tejada, S. (2002). Gamebots: a flexible test bed for multiagent team research. *Communications of the ACM*, 45(1):43–45.

[16] Kim, S. L., Suk, H. J., Kang, J. H., Jung, J. M., Laine, T. H., and Westlin, J. (2014). Using unity 3d to facilitate mobile augmented reality game development. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 21–26. IEEE.

[17] Lewis, M. and Jacobson, J. (2002). Game engines in scientific research - introduction. 45:27–31.

[18] Ma, W., Tran, D., and Sharma, D. (2007). Using tuple space to coordinate multiagent activities. In *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, pages 589–596. Springer.

[19] Mamei, M. and Zambonelli, F. (2006). *Field-based coordination for pervasive multiagent systems*. Springer Science & Business Media.

[20] Mariani, S. and Omicini, A. (2013). Tuple-based coordination of stochastic systems with uniform primitives. *From Objects to Agents*.

[21] Mariani, S. and Omicini, A. (2016). Game engines to model mas: A research roadmap. In *WOA*, pages 106–111.

[22] Marks, S., Windsor, J., and Wünsche, B. (2007). Evaluation of game engines for simulated surgical training. In *Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia*, pages 273–280. ACM.

[23] Molesini, A., Omicini, A., and Viroli, M. (2009). Environment in agent-oriented software engineering methodologies. *Multiagent and Grid Systems*, 5(1):37–57.

[24] Omicini, A. (2013). Nature-inspired coordination models: Current status and future trends. *ISRN Software Engineering*, 2013.

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

[26] Omicini, A. and Zambonelli, F. (1999). Coordination for internet application development. *Autonomous Agents and Multi-agent systems*, 2(3):251–269.

[27] Omicini, A. and Zambonelli, F. (2003). Mas as complex systems: A view on the role of declarative approaches. In *International Workshop on Declarative Agent Languages and Technologies*, pages 1–16. Springer.

[28] Papadopoulos, G. A. and Arbab, F. (1998). Coordination models and languages. In *Advances in computers*, volume 46, pages 329–400. Elsevier.

[29] Ricci, A., Omicini, A., Viroli, M., Gardelli, L., and Oliva, E. (2006). Cognitive stigmergy: Towards a framework based on agents and artifacts. In *International Workshop on Environments for Multi-Agent Systems*, pages 124–140. Springer.

[30] Ricci, A., Viroli, M., Omicini, A., Mariani, S., Croatti, A., and Pianini, D. (2016). Spatial tuples: Augmenting physical reality with tuple spaces. In *International Symposium on Intelligent and Distributed Computing*, pages 121–130. Springer.

[31] Rossi, D., Cabri, G., and Denti, E. (2001). Tuple-based technologies for coordination. In *Coordination of Internet agents*, pages 83–109. Springer.

[32] Theraulaz, G. and Bonabeau, E. (1999). A brief history of stigmergy. *Artificial life*, 5(2):97–116.

[33] Trenholme, D. and Smith, S. P. (2008). Computer game engines for developing first-person virtual environments. *Virtual reality*, 12(3):181–187.

[34] van Oijen, J., Vanhée, L., and Dignum, F. (2011). Ciga: a middleware for intelligent agents in virtual environments. In *International Workshop on Agents for Educational Games and Simulations*, pages 22–37. Springer.

[35] Vilenica, A., Pokahr, A., Braubach, L., Lamersdorf, W., Sudeikat, J., and Renz, W. (2010). *Coordination in multi-agent systems: A declarative approach using coordination spaces*. na.

[36] Viroli, M. and Casadei, M. (2009). Biochemical tuple spaces for self-organising coordination. In *International Conference on Coordination Languages and Models*, pages 143–162. Springer.

[37] Viroli, M., Pianini, D., and Beal, J. (2012). Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In *International Conference on Coordination Languages and Models*, pages 212–229. Springer.

[38] Wang, S., Mao, Z., Zeng, C., Gong, H., Li, S., and Chen, B. (2010). A new method of virtual reality based on unity3d. In *Geoinformatics, 2010 18th International Conference on*, pages 1–5. IEEE.

[39] Wegner, P. (1996). Coordination as constrained interaction. In *International Conference on Coordination Languages and Models*, pages 28–33. Springer.

[40] Weyns, D., Omicini, A., and Odell, J. (2007). Environment as a first class abstraction in multiagent systems. *Autonomous agents and multi-agent systems*, 14(1):5–30.

[41] Wyckoff, P., McLaughry, S. W., Lehman, T. J., and Ford, D. A. (1998). T spaces. *IBM Systems journal*, 37(3):454–474.

[42] Zambonelli, F. and Omicini, A. (2004). Challenges and research directions in agent-oriented software engineering. *Autonomous agents and multi-agent systems*, 9(3):253–283.