ALMA MATER STUDIORUM — UNIVERSITÀ DI BOLOGNA

DEIS - Dipartimento di Elettronica, Informatica e Sistemistica
PhD Course in Electronics, Computer Science and Telecommunications

Cycle XXVII
Competitive Sector: 09/H1
Disciplinary Sector: ING-INF/05

ENGINEERING COMPLEX COMPUTATIONAL ECOSYSTEMS

Candidate

Dott. Ing. DANILO PIANINI

Supervisor

Chiar.mo Prof. Ing. MIRKO VIROLI

Tutor

Chiar.mo Prof. Ing. ANTONIO NATALI

Coordinator

Chiar.mo Prof. Ing. ALESSANDRO VANELLI CORALLI

FINAL EXAMINATION YEAR 2015

Considerate la vostra semenza:
fatti non foste a viver come bruti,
ma per seguir virtute e canoscenza.

Acknowledgements

Most of the material that got integrated as a single work in this thesis has been written in publications of which I am co-author. I am thankful to those great researchers I have collaborated with, for their suggestions, their guidance and their help. I'd like to thank first Mirko Viroli, who supervised me for the past three years and tried to make me a scientist. Sara Montagna, which shares the APICe laboratory with me, also deserves a special mention. I want to thank the work packages leaders of SAPERE, a European project that kickstarted me in the world of research: Franco Zambonelli, Giovanna di Marzo, Simon Dobson, Marco Mamei, Alois Ferscha (that also supervised me for three months in Linz) and Andrea Omicini. A special thank to Jacob "Jake" Beal, that supervised (and also hosted) me during my last three months in the United States. Finally, I desire to thank these other great researchers and professionals I have collaborated with: Bernhard Anzengruber, Jose-Luis Fernandez Marquez, Stefano Mariani, Ronaldo Menezes, Jussi Nieminen, Matteo Risoldi, Stefano Sebastio, Graeme Stevenson, Andrea Vandin, Sascia Virruso, Juan "Erica" Ye.

Danilo Pianini, 22nd December 2014

Contents

Abstract **xi**

I Background and Motivation 1

1 Pervasive computing **3**
1.1 Smart, portable devices . . . 3
1.2 Communication technologies . . . 7
1.2.1 International Mobile Telecommunications . . . 7
1.2.2 WiFi . . . 9
1.2.3 Bluetooth and Bluetooth LE . . . 9
1.2.4 NFC . . . 11
1.3 Towards a P2P pervasive continuum? . . . 12

2 Self-organisation **15**
2.1 Software ecosystems . . . 15
2.2 Nature inspiration . . . 16
2.2.1 Physical-inspiration . . . 17
2.2.2 (Bio)chemical-inspiration . . . 17
2.2.3 Stigmergy . . . 18
2.3 Spatial patterns . . . 18
2.3.1 Gossip . . . 19
2.3.2 Gradient . . . 19
2.3.3 Gradcast . . . 19
2.3.4 Voronoi partition . . . 19
2.4 Tuple-based coordination . . . 19
2.4.1 SAPERE . . . 20
2.5 Aggregate programming . . . 22
2.5.1 Proto . . . 24
2.5.2 Field Calculus . . . 25
2.6 Engineering and tools . . . 26
2.6.1 General purpose frameworks . . . 26
2.6.2 Specific simulators . . . 26

II An integrated toolchain for pervasive ecosystems 29

3 Chemical-inspired engine **31**
3.1 Gillespie's SSA as an event-driven algorithm . . . 31

3.2 Gillespie's optimised versions . . . 32
3.3 From SSA to full fledged DES . . . 33
3.3.1 Dynamic Dependency Graph . . . 34
3.3.2 Dynamic Indexed Priority Queue . . . 35

4 Meta-meta model **39**

5 Architecture **43**
5.1 Writing a simulation . . . 43
5.2 Implementation details . . . 45

6 Performance **47**

7 Collocation in literature **51**
7.1 ALCHEMIST as a DES . . . 51
7.2 ALCHEMIST as a MABS . . . 52
7.2.1 Advantages . . . 52
7.2.2 Limitations . . . 53

8 Meta-model agnostic features **55**
8.1 Distributed statistical analysis . . . 55
8.1.1 MULTIVESTA and MULTIQUATEX . . . 56
8.1.2 Integrating MULTIVESTA and ALCHEMIST . . . 57
8.1.3 Example . . . 58
8.1.4 Performance Assessment . . . 61
8.2 Real world maps . . . 63

9 Chemical meta-model **67**
9.1 Example scenario: Morphogenesis . . . 67
9.1.1 Development of Drosophila Melanogaster . . . 68
9.1.2 Simulation in ALCHEMIST . . . 71

10 SAPERE meta-model **75**

III Methods and patterns for self-organisation 77

11 A process algebra for SAPERE **79**
11.1 Model . . . 79
11.1.1 LSAs . . . 80
11.1.2 Agent primitives . . . 80
11.1.3 Eco-laws . . . 81

11.2 Formalisation . . . 82
11.2.1 Syntax . . . 83
11.2.2 Semantics . . . 84
11.2.3 Example . . . 87
11.2.4 Well-formed configurations and properties . . . 88

12 Pattern: Channel **89**
12.1 A semantic-oriented instantiation . . . 89
12.2 Realising the gradient . . . 91
12.3 From gradient to distributed channel . . . 93
12.4 Simulation in ALCHEMIST . . . 97

13 Pattern: Anticipative adaptation **103**

14 Case Study: Crowd evacuation **105**

15 Case Study: Crowd steering **107**
15.1 Reference scenario . . . 107
15.2 Steering strategy . . . 108
15.3 Simulation in ALCHEMIST . . . 110

16 Case study: Self composition **113**
16.1 Self-Composition approaches . . . 113
16.2 A comprehensive approach for pervasive ecosystems . . . 115
16.3 The self-composition issue in pervasive service ecosystems . . . 115
16.3.1 Self-Composition with Gradient service . . . 116
16.3.2 A prototype solution . . . 118
16.4 A formal model for gradients self-compositions . . . 119
16.4.1 The gradient service . . . 119
16.4.2 Rules for Gradient Composition . . . 122
16.5 Towards simulation of gradient self-compositions . . . 123

17 Case study: Semantic resource discovery **125**

18 Case study: Crowd disasters prediction **127**

19 Case study: Crowd steering at the urban scale **129**
19.1 Devices and physical configuration . . . 129
19.2 Distributed crowd steering . . . 130
19.3 Simulation in ALCHEMIST . . . 131

IV Aggregate programming languages 135

20 Tuple based aggregate programming 137
20.1 Linda in space-time . . . 138
20.1.1 Basic model . . . 138
20.1.2 The coordination language . . . 138
20.1.3 Definitions . . . 140
20.1.4 Time . . . 140
20.1.5 Space . . . 141
20.1.6 Finally . . . 142
20.2 Core Calculus . . . 142
20.2.1 Syntax . . . 142
20.2.2 Operational Semantics . . . 144
20.3 Case studies . . . 145
20.3.1 Adaptive Crowd Steering . . . 145
20.3.2 Linda in a mobile ad-hoc environment . . . 148

21 Scale independent computations 151
21.1 Space-Time Computation and Discrete Approximation . . . 152
21.1.1 Modelling Networks as Space-Time Manifolds . . . 153
21.1.2 Computations on Continuous Space and Time . . . 155
21.1.3 Space-time programs . . . 159
21.1.4 Eventually Consistent Programs . . . 159
21.2 Eventually Consistent Language . . . 160
21.2.1 From Consistency Failures to Fragility . . . 160
21.2.2 GPI-calculus . . . 162
21.2.3 GPI calculus is Eventually Consistent . . . 165
21.3 Example Applications . . . 165
21.3.1 Distance-Based Patterns . . . 166
21.3.2 Path Forecasting . . . 168
21.3.3 Context-Sensitive Distance . . . 170
21.3.4 Confirmation of Results in Simulation . . . 171

22 Higher order functions in field calculus 177
22.1 Impact on alignment . . . 177

23 Protelis: practical aggregate programming 179
23.1 Syntax . . . 179
23.2 Ordinary Language Features . . . 180
23.3 Special Field Calculus Operators . . . 182
23.4 Architecture . . . 184
23.5 Application example: Rendezvous at a Mass Event . . . 187

23.6 Application example: Network Service Management 188

V Conclusion **191**

24 Results achieved **193**
24.1 Integrated toolchain for pervasive ecosystems 193
24.2 Methods and patterns for self organisation 193
24.2.1 A process algebra for SAPERE 193
24.3 Aggregate programming languages 193
24.3.1 Scale independent computation in situated networks 193
24.3.2 Higher Order Functions in Field Calculus 194
24.3.3 Protelis 194

25 Future and ongoing work **195**
25.1 Aggregate programming 195
25.1.1 Scale independent computation in situated networks 195
25.1.2 Protelis 196
25.2 Biochemical meta model for ALCHEMIST 196

Bibliography **197**

Appendices **223**

A Proofs of Theorems **225**
A.1 Discontinuity of Discrete-Valued Fields 225
A.2 Consistency of GPI-calculus 225

Abstract

This work presents advancements of the latest three years in the engineering techniques for self-organising pervasive ecosystems of devices and services. The inherent complexity of such systems poses new challenges to those who try to dominate the complexity by applying the principles of engineering.

The recent growth in number and distribution of devices with decent computational and communicational abilities, that got suddenly accelerated with the massive diffusion of smartphones and tablets, is envisioning a world with a much higher density of devices in space. This already high device density is probably going to consistently rise if the diffusion of wearable devices gets momentum. Also, communication technologies seem to be focussing on short-range device-to-device (P2P) interactions, with technologies such as Bluetooth® Low Energy and Near-Field Communication getting more and more diffused.

Locality and situatedness become key to provide the best possible experience to users, and the classic model of a centralised, enormously powerful server gathering data and processing it is likely to get less and less efficient with device density. Accomplishing complex global tasks without a centralised controller responsible of aggregating data from devices, however, still is a challenging task. In particular, it is hard to understand which device-local programs could properly interact and guarantee a certain global service level. Such local-to-global issue makes the application of engineering principles challenging at least.

In this work, I lay the foundations of my contribution by first analysing the state of the art in coordination systems, namely in those software frameworks devoted to control and promote interactions among independent software entities. I then motivate my work, by describing the main issues of pre-existing tools and practices and identifying the improvements that would benefit the design of such complex software ecosystems. My contribution can be divided in three main branches:

1. a novel simulation tool-chain for pervasive ecosystems;
2. introduction of novel and improvements over existing self-organisation patterns;
3. creation of a new language and interpreter based on “field calculus” and its integration with the previously mentioned tool

Finally, I draw conclusions and future works.

Part I

Background and Motivation

1 Pervasive computing

It is no mystery that, with the huge progresses of miniaturisation, computational-capable devices are populating the world. The diffusion got momentum with affordable "personal computers" that made their way in a one-device-per-family world. Laptops boosted the process, offering user a personal and mobile device.

The pervasive revolution, however took place when the phones became "smart", and with the subsequent improvements in the communication technologies. In this chapter I try to briefly walk the path of success of personal mobile devices, describing also the probable newcomer, namely the wearable devices. I also focus on the current status of the communication protocols, their range and usage, and I try to foresee which world are we going to build if the current trend continues.

1.1 Smart, portable devices

When Apple in 2007 released the first iPhone (Figure 1.1), a mobile revolution started. Even though other manufacturers proposed products similar to iPhone under the point of view of communication technologies and computational capabilities in the same time frame (e.g. Nokia N810), the Apple's smartphone was the first gaining widespread adoption. It is not really relevant for this work to understand if the branding, the design, the multi-touch finger based interaction UI or the feature set was the key of its commercial success: what really matters is that, starting 2007, every person began to carry with her a personal device featuring both the abilities to communicate and compute. The reason, besides the success of iPhone in the higher segment of the phone market, is mainly to attribute to the widespread diffusion of similar but cheaper and less powerful devices in the lower segments. This kind of devices ultimately pushed the market share of the dominating today's mobile operating system, Google's Android.

This trend towards a higher diffusion got another leap forward three years later, in 2010, when a device with a feature set similar to iPhone, but with no phone abilities and a bigger screen was released: the iPad. As iPhone gave new vitality and perspectives to the phone arena, iPad revitalised a market that was languishing: the tablets. Tablet devices as they were conceived before iPad launched were nothing more than small laptops with a screen that could be rotated or detached from the keyboard, and a touch-screen normally used with the help of a pen. They

Figure 1.1: Apple iPhone is one of the first devices without any physical keyboard, and the first smartphone to gain worldwide success. *Source: Wikimedia.*

changed from devices designed for a professional niche to widespread tools, up to the point that Gartner forecasts their shipments to overtake in 2015 those of desktop PC and laptops aggregated [1].

The new frontier of pervasive is probably the wearable technology. Smartphones were precursors, they substituted mobile phones introducing new features and they have potential to substitute our wallets (see, for instance, payments through NFC technology) and keys, becoming the only object we need to carry with us in our pockets. Still there are other accessories which are hard to replace, above all watches and glasses. Yes, smartphone can easily show the current time accurately (and, to be honest, also feature phones had this feature well before 2007), but they require the user to pick them from the pocket, turn them on, and sometimes, depending on privacy settings, also unlock them. The whole operation, takes a much longer time and higher effort with respect to just rotating a wrist. This might be the reason why, despite the explosive expansion of smartphones, the wristwatches market did not declined, as an analysis from MarketWatch points out [2].

There is room for manufacturers to create new portable devices. Sony, starting 2012, has produced a series of smartwatches, such as those in Figure 1.2 that pair with a smartphone and provide quick access to some of its functionalities. The success of such solution is not huge, but despite that many other companies are interested in this market: Samsung, Motorola and

[1] `http://www.gartner.com/newsroom/id/2791017`

[2] `http://www.marketwatch.com/story/the-watchs-time-isnt-up-2013-07-01`

Figure 1.2: Smart watches. From left to right: Sony Smartwatch *(Source: Alex S.H. Lin)*, Samsung Galaxy Gear *(Source: Karlis Dambrans)*, Motorola Moto 360.

Apple presented devices meant to replace the classic wristwatch, a clear sign that this market is in expansion. At the time of writing, the main issues that slow the widespread adoption of such solutions are battery duration and dependence on a smartphone.

Another notable attempt to make a common accessory smarter is Google Glass project, depicted in Figure 1.3. Their goal is to enhance the experience of wearing glasses by attaching a device with a camera, an optical head-mounted display, and the abilities to locate itself and communicate with other devices. Google Glass, at the time of writing, are way to expensive (with the kit sold at $1500) for being able to penetrate the general public, but they are an interesting anticipation of possible future devices.

On the same line of such wearable devices, a discrete success is being achieved by the so called “fit bands”. They are bracelets equipped with low energy sensors, mainly accelerometers and gyroscopes, which are used to keep track of user’s activity. Depending on the model, they can be used to monitor some user’s health parameters, such as the number of steps walked per day or heartbeats. They normally work along with another device, a smartphone or a tablet. Such devices, due to their precise market niche and reasonably low price (the Chinese manufacturer Xiaomi recently introduced a low-end wristband at around $15) are having a notable success.

The wearable devices segment also includes less common devices such as “smart shoes” and materials that can be used to make clothing, such as e-textiles. It is a market in expansion, greatly beneficing from recent increases in performance per watt efficiency. If the trend continues, it is likely that we will more and more powerful wearable devices on sale at cheaper and cheaper prices, and a consequent widespread diffusion. The same sort may occur to other parts of our life: kitchen gear, indoor lights and many other objects are getting more and more “smart” around us. We may, literally, end up with a world where every single object embeds computational and communicational abilities.

A problem arise: how can software engineers deal with such a complexity?

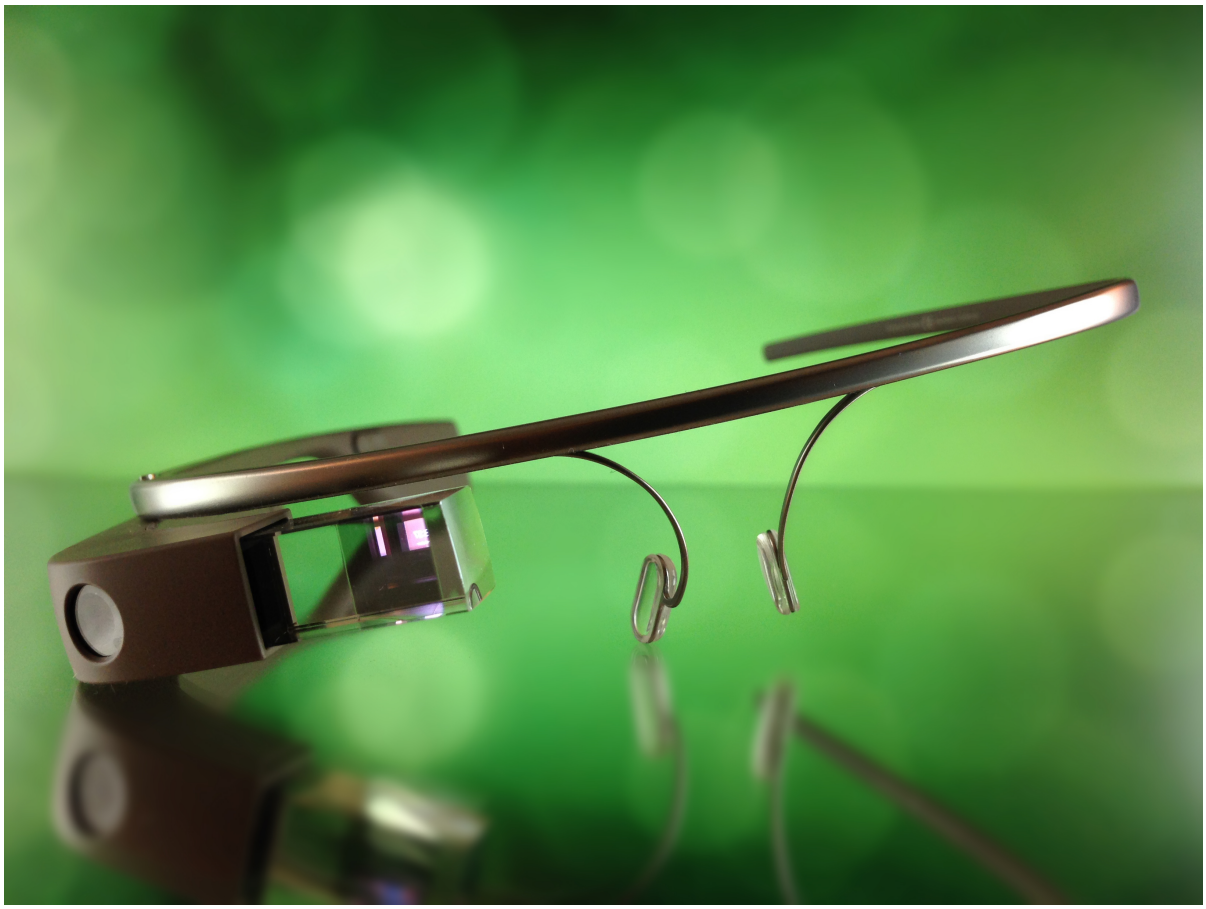


Figure 1.3: Google Glass. In this image, it is possible to see both the camera (on the left hand side) and the semi-transparent head-mounted display. Are those devices going to be part of our everyday life? *Source: Wikimedia.*

1.2 Communication technologies

Besides miniaturisation, and as a consequence the increase of computational density in space, another factor played a fundamental role in the world of pervasive computing: the ability to communicate, and in particular the ability to rely on wireless communication, which is of paramount importance when considering mobility.

In later years, many communication means arose. They largely differ in terms of range, protocols, and availability. In this section I try to resume the most diffused technologies available on today's devices, but the reader is warned: keep in mind that such technologies are evolving very quickly, and the scenario is incredibly fluid.

1.2.1 International Mobile Telecommunications

This first mean of communication is designed to allow mobile devices to access the Internet from anywhere in the world, relying on the existing mobile phone infrastructure. Such technologies are meant to be used with the standard IP protocols, and they are normally used to get access to public services, in particular to the world wide web. They are not designed for a local peer-to-peer (P2P) communication, and as a consequence they provide no mean to exploit locality. The diffusion of such communication protocols is widespread, in particular among smartphones. Due to the fact that they rely on the mobile phone network, they require a contract with a mobile telecommunications provider, and as such they are much less diffused in tablets and other portable devices.

The possibility of accessing the Internet from everywhere is probably one of the key bricks that allowed for the huge success of smartphones in today's world. As Figure 1.4 shows, the bandwidth available grew exponentially with time, to the point that in some countries (e.g. in Italy, at the time of writing) the best available mobile connections offer a higher performance than the best available home connection [3]. Such performance unlock the possibility of fully exploiting the possibilities of the world wide web, including cloud services and fruition of multimedia content.

If bandwidth is not currently an issue for international mobile telecommunications, the situation is well different when it comes to device density. Any of us probably experienced network availability issues when participating crowded events, such as concerts or sport events. The current technology, in fact, makes all the network user share the same physical resources: when the device density is too high, there is simply not enough space in the frequency spectrum to grant a decent bandwidth to everyone. Future networks (5G, and presumably those that will follow) are focussing toward this issue among others [ASS+11]. In particular, a so-called "spectrum crunch" is expected due to the expected traffic increase (thousand fold over this decade and still growing into the next), that could not be faced simply with the foreseen steady increase

[3] At the time of writing, Telecom Italia Mobile offers mobile connections on LTE with a download bandwidth up to 225Mb/s. Fastweb, the company offering the faster solutions for fiber-to-home connections, goes up to 100Mb/s.

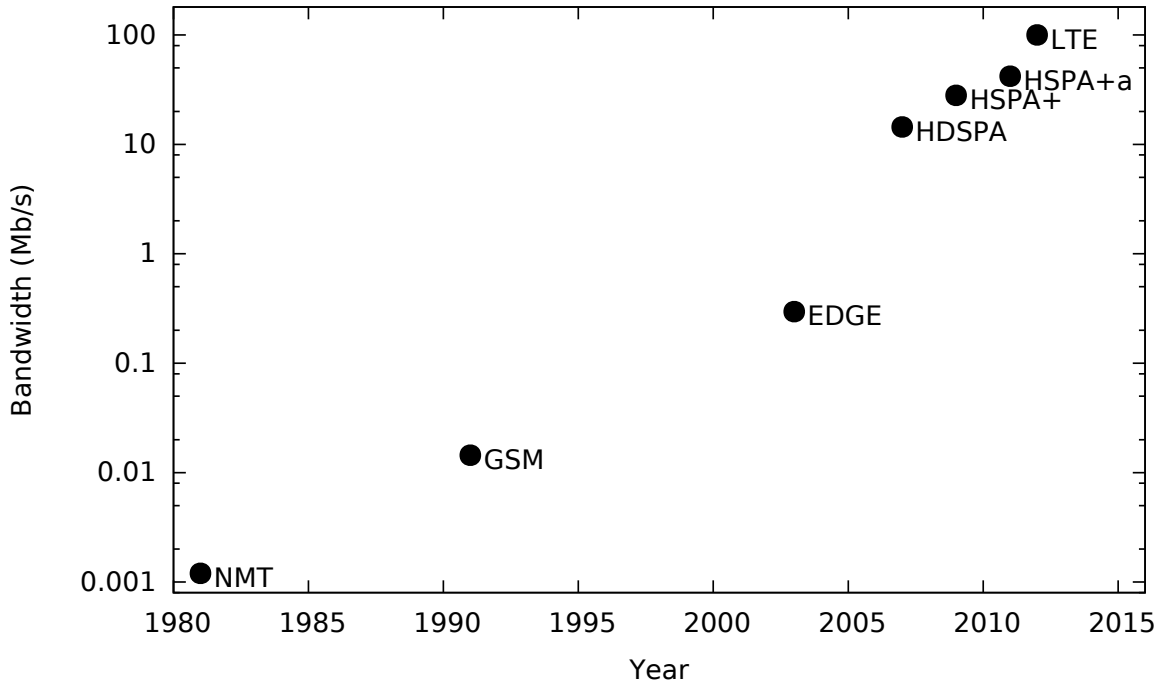


Figure 1.4: Maximum download bandwidth available for mobile devices with time. Each point is labelled with the specific communication technology name. For the most recent multiple-input-multiple-output (MIMO) technologies, such as LTE, a conservative single input channel was used.

of the spectrum allocated for mobile communication, and will require technological advances [AARC+12].

1.2.2 WiFi

WiFi technology is the most diffused technology for wireless local networking. It is widely diffused, integrated in all smartphones and tablets and also in other devices, such as printers, gaming consoles and TVs. WiFi devices communicate on a distance that ranges from 20 to 100 meters, depending on the condition of the wireless medium and on the power of the communication devices. The communication speed between two linked devices ranges from 56Mb/s to 300Mb/s.

WiFi was designed to provide wireless access to a local area network. In the most classic "infrastructure mode", wireless devices get connected to a so called access point, which is responsible to route packets among wireless devices and bridge the wireless local area network to the wired backbone. Multiple access point that share the network name (SSID) may be connected using wired network technologies, and they will appear as a single, bigger access point. It is also possible to drop the wired backbone, but specific access points are required.

Also, some WiFi devices provided "ad-hoc mode", allowing multiple devices to directly communicate without an intermediate access point. This working mode was problematic, mainly due to the fact a standard communication protocol for peer-to-peer WiFi communication was missing. This lack was filled with WiFi Direct, which provides a protocol by which one of the devices that want to communicate directly becomes the access point, allowing for direct communication. The most common usage of such a feature are direct file sharing between devices and connection to peripheral devices such as printers or scanners.

1.2.3 Bluetooth and Bluetooth LE

Bluetooth is a technology designed for building energy efficient personal area networks (PANs). Bluetooth devices are assigned a class which identifies the maximum permitted power and, consequently, the maximum operating range. For the most powerful (and power hungry) devices the communication range can go up to 100m. The communication speed ranges from 1Mb/s of the earliest 1.0 version to the 24 Mb/s of version 3.0 and later.

The most interesting features of Bluetooth are not the bandwidth nor the range (WiFi performs better on both), but rather the simple association process and the low power consumption. Thanks to those features Bluetooth found widespread diffusion as a mean to connect low consumption peripherals, such as headsets. Also, it is diffused in cars, and allows user to use the car's audio system as a speakerphone for making calls or listen to music.

A technology which is often associated with Bluetooth but that is actually a separated and not compatible protocol is Bluetooth LE. The reason why such technologies get associated is that, since the radio frequency used is the same (2.4GHz), dual mode devices can share a single radio antenna. LE stands for Low Energy, and it is the main difference between the two protocols: at

Figure 1.5: A iBeacon, compared to a 2€ coin.

the expense of some bandwidth, Bluetooth LE consistently reduce the amount of energy required. Bluetooth LE applications are particularly interesting, and range from health care to fitness to alerts to proximity sensing.

Proximity can be estimated using the received signal strength indicator (RSSI), and the very low power consumption of Bluetooth LE allowed for the realisation of electronic leashing systems, namely systems where an electronic device is paired to an object and can be used in order to compute the relative position. The applications are, for instance, finding of misplaced, out-of-sight devices (when the electronic device is paired with a movable object) and indoor localisation (if the electronic device is located on a still standing object). Relying on this technology, Apple created iBeacon, namely very small (coin sized, see Figure 1.5) electronic devices consisting basically of a battery and a Bluetooth LE device. iBeacons can be attached to objects, and they send a universally unique identifier (UUID) to enabled smartphones in range. If the smartphone can associate the UUID with a position, it can deduce its location relative to the iBeacon. The low energy feature plays a fundamental role in this kind of applications: beacons whose battery would last few hours would be of little practical use. With current technologies, a beacon device can be powered by a standard, rather cheap battery for several months, up to a couple of years. This technology is probably the prelude to precise indoor localisation.

1.2.4 NFC

Near Field Communication, or NFC, is a technology designed for low energy communication between two devices in proximity (typically few centimetres). It is designed for low energy

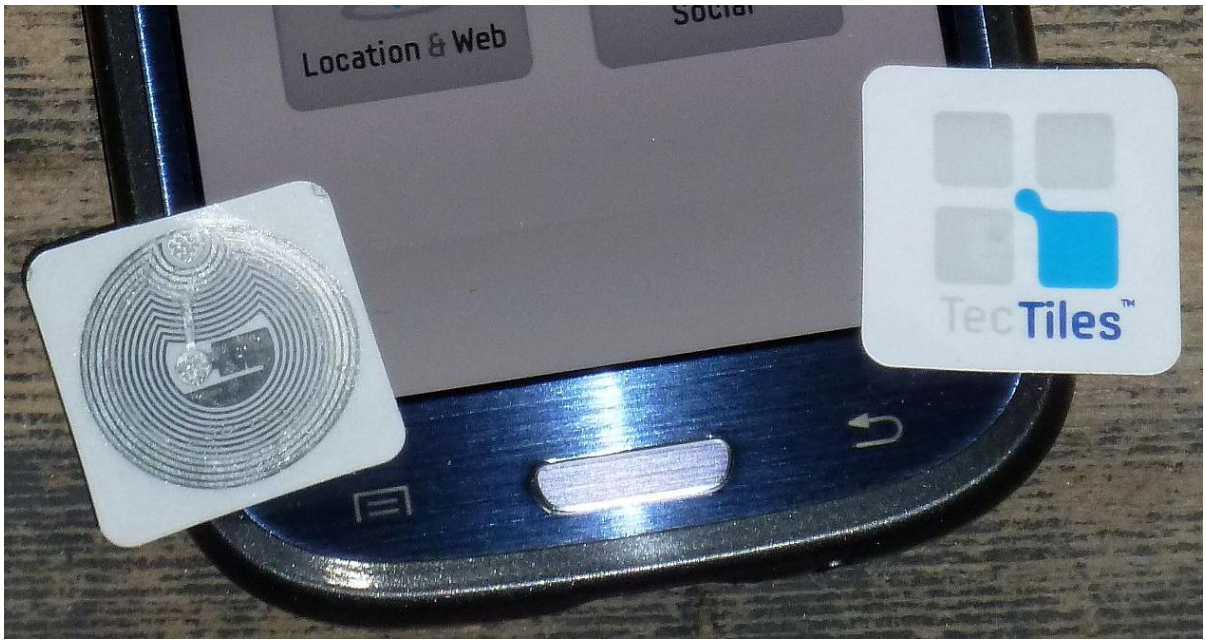


Figure 1.6: Two NFC tags, mounted on stickers. In order to understand the size, they are placed above a Samsung Galaxy S3, a 4-inches smartphone. Moreover, smaller versions of such tags exists, the models pictured here are very common-sized.

consumption rather than high bandwidth: its speed (depending on the specification) ranges from 126kb/s to 242kb/s.

One of the most interesting features of NFC is that one of the two devices (the so-called "tag") can be completely passive, and still carry a small amount of information within (currently between 96 and 4096 bytes). No battery or energy source is required, the information included can be read by active NFC devices in proximity. Having no need of battery at all, NFC tags can really be tiny, even smaller than iBeacons. Figure 1.6 shows two NFC tags: compared with iBeacons, they can be much lighter and thinner. Brought to the world of humans, it is something like creating a sticker with very small sized text: those who have a powerful enough magnifying glass and are close enough to use it properly can read what it says.

The range of applications of such technology is rather broad. The one which was probably most sponsored is contact-less payment, namely the ability to pay just tapping the phone close to the check-out counter. This is a very interesting possibility, and indeed in 2007 there were enthusiastic forecasts [OP07] about its quick diffusion, that did not happen as quick as expected. A number of studies tried to understand the reasons behind this slow adoption, and it appears that reasons are more correlated to marketing and management rather than technological maturity [PH14, Apa13, TOCH14]. Similarly, if the phone stores identity or access tokens, NFC is a very suitable technology for effectively using such tokens: in this case, its very low range is a nice feature. NFC can also be used as a technology enabler, namely as a mean to securely bootstrap

another connection, or join a local network of devices. An example of such use is the Android Beam technology, that relies on NFC in order to establish a Bluetooth connection between two Android devices, transfer a file, then close the Bluetooth connection. Technically, NFC could be used to directly transfer files, but both WiFi and Bluetooth offer much wider bandwidths and range, and as a consequence are preferred for such task. Another interesting application field is mobile device automation: it is achieved by attaching Generally speaking, NFC comes in handy when there is need of a communication mean whose range should be very limited.

1.3 Towards a P2P pervasive continuum?

We are living exciting times. In about five years from the introduction of the technology on the market, almost everybody got a personal smart device always with her. Miniaturisation and power efficiency is constantly growing at a stunning rate, allowing data, communication systems and computation to be spread around in our physical world.

In few decades, we will probably witness the diffusion on computation on everyday object. In such a scenario, the device density will be much higher if compared to the current, up to the point that the aggregation of devices participating the system could be seen as a "pervasive continuum" [ZCF+11]. This continuum is studied under a number of names, including pervasive computing, smart cities, and the Internet of Things, in order to provide a wealth of services in an un-intrusive manner [ZOA+14, CDB+12, Zam12, HDBDM12].

One of the possible strategies is to connect every single device to the Internet, aggregate its information in a remote server, do the necessary computation, then send back eventual results where they are needed. This is the strategy behind cloud computing, which is achieving great success. In particular, we argue, this strategy is interesting when the information could or should be aggregated with information from other, distant sources, or conserved for historical purposes. This path, however, gets harder and harder to follow with device density: besides the obvious increasing on the total information produced, and consequently of the information to transmit and process, there are two other problems: the saturation of the wireless medium, and the locality of information.

Who tried to use its own smartphone in a very crowded environment has probably experienced connection or network issues. The problem, as discussed in Section 1.2.1, is that current technology must share a common medium among all the devices in the same area. Increasing the maximum number of devices per area is one of the goals of the next generation of international mobile telecommunication technologies. One of the proposed approaches is to switch to a very dense array of very small cells by deploying multiple antennas at a very short distance one another, e.g. inside the public illumination poles. Devices nearby the local antenna would connect to it, and the antenna would then connect them to the rest of the network transparently, in a way somehow similar to the current WiFi "infrastructure mode". Clearly, diffusing such antennas can possibly represent a major infrastructural upgrade, and, potentially, cost.

Another observation is that not every device needs direct access to the Internet to be able

to accomplish its task, and this is increasingly true with increased density. Thinking about today's devices, let's consider the current smartwatches and fitbands: the former relies on a smartphone or tablet in order to provide Internet-based services, and the latter uses no Internet connection at all, but just sends data to the smartphone to be processed. Along the line of favouring locality, there is a second advantage which relates to privacy issues: there is no reason to send personal information away, if the system does not need data from distant points nor requires more computational power of the amount available locally. Privacy issues gain great attention recently, especially after the leaks of classified information started in 2010 on Wikileaks and continued with the more recent leaks by Edward Snowden. The content of such documents raised greater attention to privacy issues from general public.

A possible path which would help in both those directions (reduce wireless medium usage and keep data as local as possible) is the usage of local, possibly peer-to-peer interactions. This way of organising communication is already exploited by existing applications.

One notable example is Firechat [4], which got particularly spotlighted during the "Umbrella revolution", namely the sequence of protests that took place in Hong Kong in 2014. Similarly to what was done during Arab springs in 2011 [HDF+11], protesters relied on Internet services to organise and coordinate themselves. The Chinese government policy on internet is not exactly a bright example of openness and neutrality [Mac08], and services such as Facebook adn Twitter, widely exploited during Arab springs, were already effectively blocked in the land. In short time, other social network were closed (such as Instagram), in order to cut protesters' communication means. At that point, protesters had to find a communication system free of centralisation in order to prevent targeted Internet filters, and Firechat was the answer. Firechat is a messaging application for mobile phones that, when the smartphone has access to the remote Firechat cloud, works as other more famous alternatives, such as WhatsApp[5] and Telegram[6] do. When no access to Firechat servers is available, then the software tries to reach the destination by spreading the message hop-by-hop, building a de-facto mesh network.

Another interesting experiment is Serval Mesh [GSP11]. Serval Mesh accomplishes similar tasks, but it also supports calls and file transfers besides messaging. Its main goal is to provide a networking among users who are in an area where there is no Internet access at all, for instance because of a disaster event. It relies on WiFi to create a ad-hoc peer-to-peer network among devices. Due to this lower level aspect, Serval Mesh requires privileged access to the hardware and some higher skill than Firechat, which is easier to setup but requires the availability of a Internet connection. In [AGS11], Serval Mesh is used to build an alternative, purely peer-to-peer telephony network.

Despite the existence of such mesh-oriented applications, however, a general and widespread approach for easily design and program the devices that compose our pervasive continuum is still under investigation [RCKZ13]. The most consistent contribution of this thesis is devoted to the research of general, well engineered approaches to build such systems.

[4] `https://opengarden.com/firechat`
[5] `http://www.whatsapp.com/`
[6] `https://telegram.org/`

2
Self-organisation

In Chapter 1 we took a look to the world of pervasive devices, also describing their communication means and hypothesising the near future development. In this chapter we focus on the software, and in particular on the challenges of engineering the development of software that will run on an ensemble of pervasive devices. We first discuss the issue of coordination and self-organisation: how do we make all those possibly devices collaborate together in order to achieve a global goal, without a centralised decision-maker? Which software platform may we devise to ease this operation? We will see that similar problems have already been successfully solved in nature: the mechanisms underlying such natural behaviours, can, if properly mimicked, help to realise solutions in software systems. We will then run through the existing literature on the issue, analysing the existing platforms supporting pervasive computing, and the tools that can be used to test and debug applications prior to deployment. Finally, we will discuss the shortcomings of the existing technology, and pave the way for the contribution of this PhD thesis.

2.1 Software ecosystems

Regardless the name that we want to use, being it "Internet of Things" rather than "Smart cities" or "Pervasive computing", in all cases we are talking about a system that is expected to host many computations that feature, according to [ZOA+14]:

Situatedness — Pervasive services are typically time- and space-dependent, and feature physically- or socially-situated activities. Components of pervasive systems should be able to interact with the surrounding physical and social world by adapting their behaviour accordingly.

Autonomy and self-adaptivity — While individual components should be autonomous in the face of the inherent dynamics of their operational environment [GPO13], pervasive systems should also feature *system-level autonomy* to deal globally with the unpredictability of the environment, providing properties such as self-adaptation, self-management, and self-organization [MMTZ06].

Prosumption and diversity — Infrastructures for pervasive systems must promote open models of component integration, to be able to take advantage of the injection of new services and components [ZO04]. This is particularly true in the context of *socio-technical systems* [OC13], where human users and software agents act as *prosumers* – both consumers and producers – of devices, data, and services.

Eternity — As well as short-term adaptation, a pervasive systems infrastructure should allow for the long-term evolution of organizations, components, and patterns of usage, in order to accommodate technological advances as well as the mutable needs of users without requiring extensive re-engineering effort [Jaz05]. In fact, pervasive systems are better conceived as *eternal* systems, engineered for continuous, unlimited service, upgrading, and maintenance over time.

At the same time, "traditional" networks are also increasing in scale and importance for enterprises both large and small. In an increasingly information-dependent and interconnected world, the rising cost of managing and maintaining such systems is driving a search for solutions that can increase the autonomy of computing systems, enabling them to act more as collective services than individual machines [EAWS12, HQL+11], and continuing working over time. In both of these cases, and a number of other areas facing similar challenges (e.g., large-scale sensor networks, multi-UAV control), there is a growing recognition that new paradigms are needed to face the challenge of coordination, namely, engineering the space of interactions [Weg97]. The goal of such engineering paradigms is to find reliable processes that lead to self-organising aggregates that act more as a collective service than as individual machines [CLMZ03, BMP+11, LPF12, SRM+13].

2.2 Nature inspiration

Natural systems are good in dealing with the *complexity* of coordinating large-scale software ecosystems [Sha04, KR08, LT06], since they natively feature key properties such as autonomy, openness, fault tolerance, situated behaviour, self-adaptation and robustness. Many specific models have been proposed in literature, taking inspiration from natural ecosystems, social insects, biochemistry, chemistry and also physics [ZV11].

2.2.1 Physical-inspiration

To the category of physical inspired systems belong all those works that take inspiration on the way physical particles move and self-organise according to gravitational and electromagnetic forces. For instance, in [MZ06a], computational "force fields" – generated either by coordinated components or by the coordination middleware – propagate across a spatial environment, leading to distributed data structures that affect the actions and motions of the agents in that environment. A similar approach is proposed in Co-fields [MZL03], exploiting composite computational fields

to coordinate the motion of users and robots in an environment. Physical inspiration also has influenced self-assembly, for instance in [GJM12] the adoption of virtual force fields is suggested in order to control the material's shape.

A number of "bottom-up" methods implement computational fields using only the local view, including the Hood sensor network abstraction [WSBC04], Butera's "paintable computing" hardware model [But02], TOTA [MZ09], the chemical models in [VCMZ11], and Meld [ARGL+07]. The MGS language [GMCS05] takes a notable and different approach, using a method like local field computation to define and evolve the shape of the manifolds on which it executes.

Hybrid automata [Hen96] are particularly interesting for they are relying on a continuous computational space. They are based on the idea of coupling discrete, automata-like descriptions with differential equations that describe continuous evolutions of numerical values, e.g. to consider interaction with sensors/actuators in the physical world. Continuous Spatial Automata [Mac90] push the idea further, by also assuming a continuous spatial substrate over which computation can occur.

More explicit models of field computations are provided by some sensor nework programming models (e.g., Abstract Regions [WM04] and Regiment [NW04]), as well as a number of parallel computing models, most notably StarLisp [LMMD88] and systolic computing (e.g., [EC89, RL93]), which use parallel shifting of data on a structured network.

2.2.2 (Bio)chemical-inspiration

Chemical reactions can be seen as an ensemble of myriads of simple laws that generate and regulate the evolution of extraordinarily complex molecular structures. Those rules coordinate in some way the behaviours of a huge number of components, up to the point where they reductionistically drive the evolution of complex assemblies such as biological organisms and meteorological systems.

Gamma [BLM90] was the first and the most prominent example of a chemically-inspired model. In Gamma, coordination is conceived as the evolution of a space governed by chemical-like rules, globally working as a rewriting system [BFLM01]. In the CHAM (chemical abstract machine) model [Ber92a], states are interpreted as chemical solutions, where floating molecules (representing coordinated entities) interact according to some reaction rules, and where *membranes* constrain the execution of reactions.

In chemical tuple spaces [VCNO10] data, devices, and software agents are uniformly represented in the form of chemical reactants, and system behaviour is expressed by means of full-fledged chemical-like laws that are both time-dependent and stochastic rather than in form of rewriting rules. *Biochemical tuple spaces* [VCMZ11] enhance chemical tuple spaces by shaping coordination through distribution and topology. Biochemical coordination models appear very flexible in enabling the spatial formation of both localised and distributed activity patterns, and have been exploited in many special-purpose pervasive frameworks, including crowd mobility management [WBYI08] and participatory sensing [LMG+09]. The amorphous computing model can be considered an example of biochemical coordination model [AAC+00].

Beside having been conceived as general computational model, (bio)chemically-inspired computing can be an effective starting point to realize schemes of dynamic service composition [FSS10] or knowledge aggregation [MO13]. Network protocols for data distribution and aggregation according to chemical models have also been explored, as in the Fraglets approach [MYT07, MMTL13]. Several proposals exist to support service composition based on chemical models [BP09], including proposals specifically conceived for adaptive pervasive services [HAES09], or to support the adaptive organization of knowledge [MO13].

2.2.3 Stigmergy

In the computer science literature, the term "stigmergy" refers to a set of nature-inspired coordination mechanisms mediated by the environment [TB99, Bon99]. Agents deposit data into a distributed, shared, environment so as to collectively (yet implicitly) build distributed data structures that can help them navigate in such environments. The notion of *stigmergy* was introduced in [Gra59] as the fundamental coordination mechanism in termite societies.

Nowadays, the most widely-studied example of stigmergic coordination in insect societies is probably that of ant colonies [DS04]. The basic mechanism is based on *pheromones* that are released in the environment by the ants that find food on their way back to the nest, thus building pheromone trails towards food that other ants are then stimulated to follow. The pheromones act as environment markers for specific social activities, driving both the individual and the social behaviour of ants. For instance, digital pheromones [PBS02, Par06] have been fruitfully exploited as the basic mechanism for coordinating the movements of robot swarms and modular robots [MZ05], for helping people find directions in an unknown environment [MZ07], and for efficiently cluster information in networks [CVG09, PVM$^+$10]. In the area of networking, stigmergy has been exploited to realize effective routing mechanisms in dynamic networks [BDT99, DDF$^+$06].

2.3 Spatial patterns

Vedere se c'è qualcosa di utile in NACO [FMDMSM$^+$12] o in [GVO07], o in [ZV11] o in [BCD$^+$06]

2.3.1 Gossip

2.3.2 Gradient

A simple though paramount data structure that can be built upon this paradigm and that is a building block of many of the more advanced patterns is the spatial gradient, which assigns to each node a value Γ depending on its position in time and space and on its context [MZ09, BBVT08, VCMZ11]. This structure originates in one or more devices called sources. In every

source device, $\Gamma = 0$. In every other device, let N be the set of devices connected with it, and let n be the n-th neighbouring device. For this device, $\Gamma = [\min(f(n))|n \in N]$: namely, the value of the device is the minimum of a function which operates on the neighbours. For instance, if $\forall n\colon f(n) = \Gamma_n + 1$, where Γ_n is the value of the gradient in n, then the local value will reflect the minimum hop count towards the nearest source. If $f(n) = \Gamma_n + d(n)$ where $d(n)$ measures the actual distance from the device towards n, then the value of the gradient will approximate the distance from the nearest source. Finally, along with the local value, a gradient can carry more information, e.g. some strings, the position of n, or numeric values.

[BBVT08]

2.3.3 Gradcast

2.3.4 Voronoi partition

2.4 Tuple-based coordination

In a tuple-based coordination model [RCD01], software agents synchronize, co-operate, and compete based on *tuples*, which are simple data structures representing information chunks. Tuples are made available in *tuple spaces*, which are shared information spaces working as the *coordination medium*. Coordination occurs by accessing, consuming, and producing tuples in an *associative* way, relying on the actual content of tuples and not on any form of naming, addressing, or indexing. An interesting survey of the technologies and platforms [MT03] analyses the suitability of tuple-based coordination systems in supporting openness, unpredictable changes in distributed environments, and several aspects related to adaptiveness: requirements that are of primary importance in pervasive systems.

Several implementations from both academia and industry exist.

Anthill [BMM02] is a framework meant to support design and development of adaptive peer-to-peer applications, in which each node is provided with a local tuple space, agents can travel the network and interact indirectly reading, writing and retrieving tuples.

Biochemical tuple spaces [VC09]

GigaSpaces [1]

JavaSpaces [FHA99, NZ01]

Lime [MPR06]

Linda [Gel85] is the common ancestor of every tuple based coordination framework.

Molecules of knowledge [MO13]

[1] `http://www.gigaspaces.com`

SwarmLinda [TM03]

T Spaces [WMLF98]

TOTA [MZ09, MZ06b] is a tuple-based middleware explicitly conceived to support field-based coordination for adaptive context-aware and spatially-aware activities in pervasive computing scenarios. In TOTA, each tuple also carries with it a diffusion rule and a maintenance rule in addition to a content. Tota inspired other works, such as the evolving tuples model [SJ07], which adds to the tuples a form of context awareness, in form of possible evolution and adaptation to environmental changes.

TuCSoN [OZ99]

SELFMAN? [RHR+08]
MARS? [CLZ00]
Plastic? [ABI09]
CArtAgO? [RPV11]

2.4.1 SAPERE

SAPERE[2] [ZCF+11] is a nature-inspired coordination model and framework to support the design and development of composite pervasive service systems. Its reference architecture and coordination model synthesize from existing nature-inspired approaches, such as [VC09, VCO09, VCMZ11, PVM+10], and is based on an assumption of spatial, local interactions (to be realized via a network of distributed tuple spaces), which is in line with all nature-inspired approaches. Its coordination laws make it possible to express and deploy general nature-inspired distributed algorithms and coordination patterns. SAPERE abstracts a pervasive environment as a non-layered *spatial substrate* deployed upon a dense network of connected heterogeneous ICT devices (Figure 2.1). SAPERE acts as a shared coordination medium embodying the basic laws of coordination. Its core components are [VPMS12]:

LSA Because of the need to tolerate diversity, a cornerstone of pervasive ecosystems is that a uniform representation is required for the various software agents living within them (whether they run on smartphones, sensors, actuators, displays, or any other computational device). Such a representation needs to expose any information about the agent (state, interface, goal, knowledge) that is pertinent for the ecosystem as a whole or for any subpart of it. In SAPERE this description is called "Live Semantic Annotation" for it should continuously represent the state of its associated component (live), and it should be implicitly or explicitly connected to the context in which such information is produced, interpreted and manipulated (semantic)—possibly relying on standard technologies and techniques of the Semantic Web, like RDF [MM04].

[2]The model was developed as part of an EU-funded research project. See `http://www.sapere-project.eu/`.

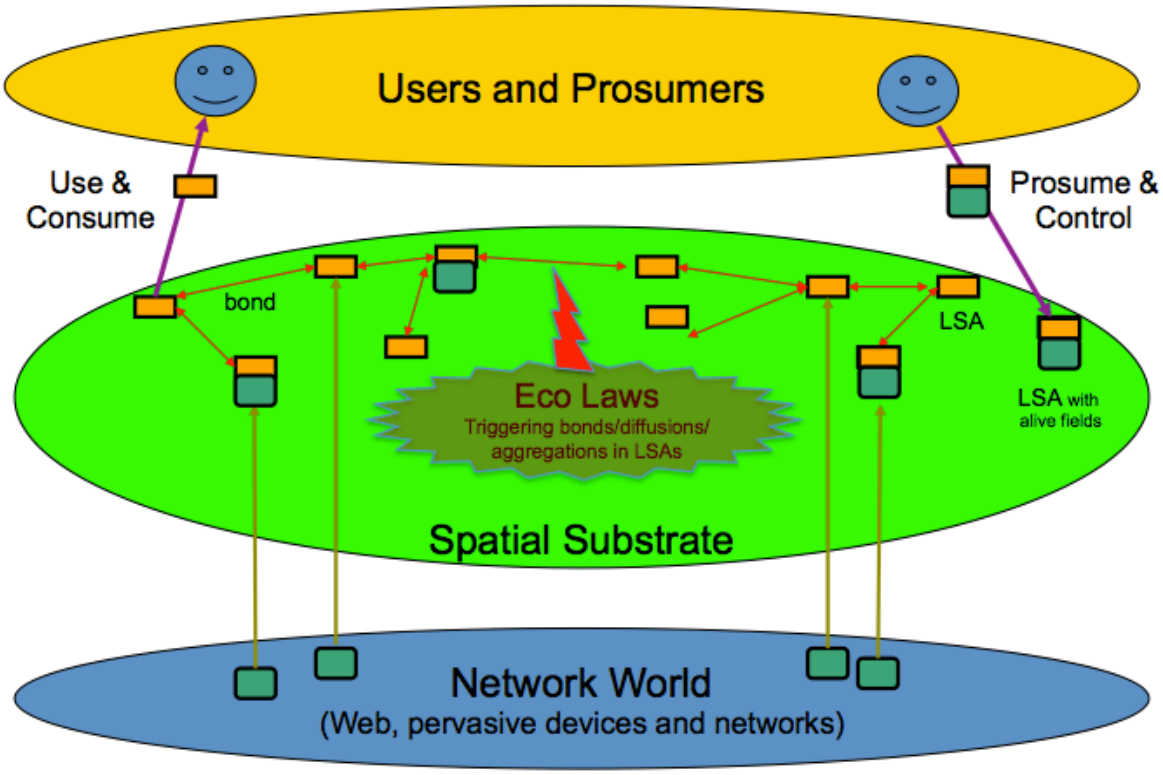


Figure 2.1: The SAPERE reference architecture.

LSA-space To handle situatedness, the behaviour of each agent strictly depends on the local context in which it runs, that is, on the state of other agents living in the same locality (intended as network neighbourhood). As such, the LSAs of each agent are reified in a distributed space (called an "LSA-space") acting as the fabric of the ecosystem, where "context" is simply defined and represented as the set of LSAs stored in a given locality.

LSA bonding Additionally, and in order to make any agent act in a meaningful way with respect to the context in which it is situated, special mechanisms are needed to provide a fine-tuned control of what to each agent is visible/modifiable and what is not. SAPERE tackles this issue by allowing an LSA to include bonds (i.e., references) to other LSAs in the same context. It is only via a bond that an agent can inspect the state/interface of another agent and act accordingly, while modifications are allowed only to the LSAs an agent injected itself.

Eco-laws Because of adaptivity, while agents enact their individual behaviour by observing their context and updating their LSAs, global behaviour (i.e., global system coordination) is enacted by self-organising manipulation rules of the LSA-space, called eco-laws. They can execute deletion/update/movement/re-bonding actions applied to a small set of LSAs in the same locality. SAPERE structures such eco-laws as chemical-resembling reactions over LSAs—similarly to other approaches like [BP09, VC09, VCMZ11].

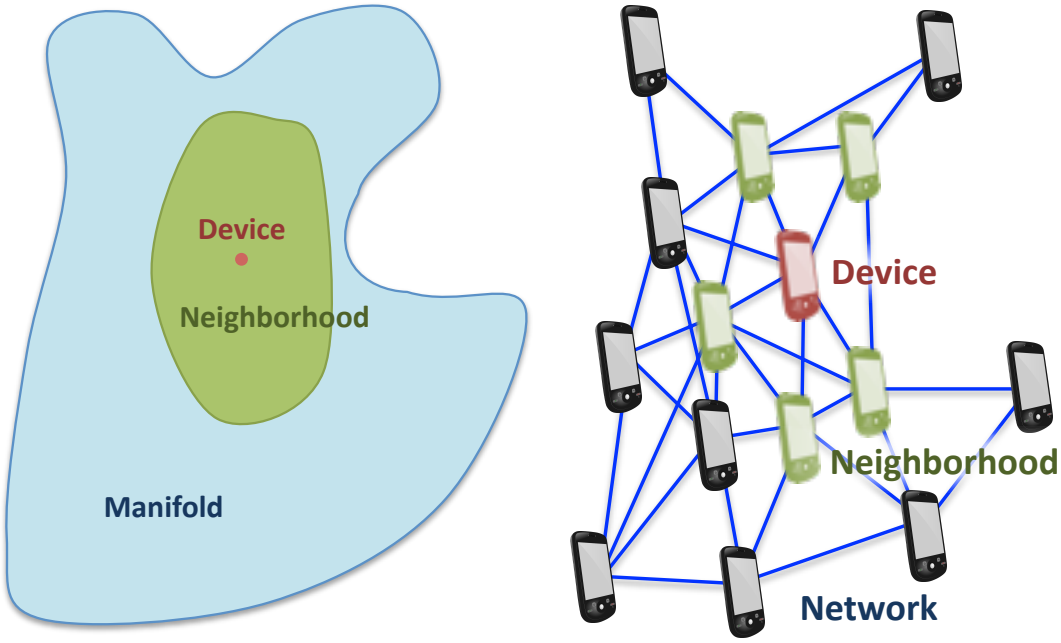
2.5 Aggregate programming

Aggregate programming is founded on the observation that in many cases the users of a system are much less concerned with individual devices than with the services provided by the collection of devices as a whole. Typical device-centric programming languages, however, force a programmer to focus on individual devices and their interactions. As a consequence, several different aspects of a distributed system typically end up entangled together: effectiveness and reliability of communications, coordination in face of changes and failures, and composition of behaviours across different devices and regions. This makes it very difficult to effectively design, debug, maintain, and compose complex distributed applications.

Aggregate programming generally attempts to address this problem by providing composable abstractions separating these aspects:

1. device-to-device communication is typically made entirely implicit, with higher-level abstractions for controlling efficiency/robustness trade-offs;

2. distributed coordination methods are encapsulated as aggregate-level operations (e.g., measuring distance from a region, spreading a value by gossip, sampling a collection of sensors at a certain resolution in space and time); and



(a) Continuous Space (b) Discrete Network

Figure 2.2: Computational field models originate from approximation of continuous space (a) with discrete networks of devices (b).

3. the overall system is specified by composing aggregate-level operations, and this specification is then transformed into a complete distributed implementation by a suitable mapping.

In short, the key idea behind aggregate programming is to provide languages and APIs that allow a distributed collection of devices to be programmed in terms of their collective behaviours, keeping outside the sight of the software designer details on how such coordination is actually implemented.

A large number of very diverse aggregate programming approaches have been proposed, including abstract graph processing (e.g., [GGG05]), declarative logic (e.g., [ARGL+07]), map-reduce (e.g., [DG08]), streaming databases (e.g., [MFHH05]), and knowledge-based ensembles (e.g., [NLPT14])—for a detailed review, see [BDU+13].

Despite the fact that all of them are based on viewing the collection of devices as an approximation of continuous space (as depicted in Figure 2.2), most aggregate programming approaches, however, have been too specialized for particular assumptions or applications to be able to address the complex challenges of these emerging environments.

2.5.1 Proto

Proto language and middleware [BB06] exploits field-based coordination to orchestrate the activities of sensor-actuator networks

Proto is based on the notion of a *computational field*—a map from devices comprising the system to (possibly structured) values, which is treated as unifying first-class abstraction to model system evolution and environment dynamics.

The programming constructs are combined together to form programs, whose semantics is defined in terms of a sequence of synchronous rounds of evaluation by a discrete network of devices called “computational rounds”. In practice, however, there is no requirement for synchrony, and each device can evaluate its own computational rounds independently.

Despite its qualities, Proto still lacks many features expected in a modern programming language and has an implementation encumbered by a number of obsolete considerations that make it difficult to maintain and extend.

[BB06]

2.5.2 Field Calculus

Field Calculus is an attempt to find a unifying model for programming *computational fields* as a generalization of a wide range of existing approaches (Proto in particular, but also [MZ09, NW04, VCMZ11, MZ06b, Nag01, Yam07, NW04]). Formalized as the computational field calculus [VDB13], this universal language appears to provide a theoretical foundation on which effective general aggregate programming platforms can be built.

Critically, although originally derived from continuous-space concepts, the calculus does not depend on them and is applicable to any network. Field calculus is expressive enough to

be a universal computing model [BVD14] but terse enough to enable a provable mapping from aggregate specifications to equivalent local implementations.

Field calculus [VDB13] hence provides a key theoretical and methodological foundation for aggregate programming. Its aim is to provide a universal model that is suitable for mathematical proofs of general properties about aggregate programming and the aggregate/local relationship, just as λ-calculus [Chu32] provides for functional programming, π-calculus for parallel programming [Mil99], or Featherweight Java [IPW01] for Java-like object-oriented programming.

In the field calculus, everything is a field: computational fields are used to model every aspect of distributed computation, including input from sensors, network structure, environment interactions, distributed computations (e.g. progressive aggregation and spreading processes), and output for actuators. In particular, field calculus is constructed using five basic constructs:

1. function definition and evaluation;
2. "built-in" operations for stateless local computation, e.g., addition, multiplication, reading a sensor;
3. a time-evolution construct which allows for stateful computation;
4. a neighbour-value construct that creates a field of values from a device's neighbours;
5. a restriction operator to select which computations to perform in various regions of space and time.

As well as in Proto, in Field Calculus the semantics of programs created by combining such constructs is defined in terms of a sequence of (possibly synchronous) "computational rounds".

The minimal syntax of field calculus has allowed its semantics, including proper coherence of device interactions, to be proven correct and consistent [VDB13]. Additionally, despite its definition in terms of discrete semantics, field calculus is also space-time universal [BVD14], meaning that it can approximate any field computation, either discrete or continuous, with arbitrary precision given a dense enough network of devices.

This, then, is the key contribution of field calculus: any coordination method with a coherent aggregate-level interpretation is guaranteed to be expressible in field calculus. Such a method can then be abstracted into a new aggregate-level operation, which can be composed with any other aggregate operation using the rules of built-in functions over fields. Moreover, it can have its space-time extent modulated and controlled by restriction, all while guaranteed that the relationship between global specification and local implementation will always be maintained.

Such a core calculus, like any other, is more a theoretical framework than a practical programming language. In practice, in fact, effective aggregate programming for real-world distributed applications is likely to require a layered approach such as the one depicted in Figure 2.3, of which field calculus represents the first block. Some of the prior approaches on which field calculus is based provide very similar semantics (most notably Proto), but they

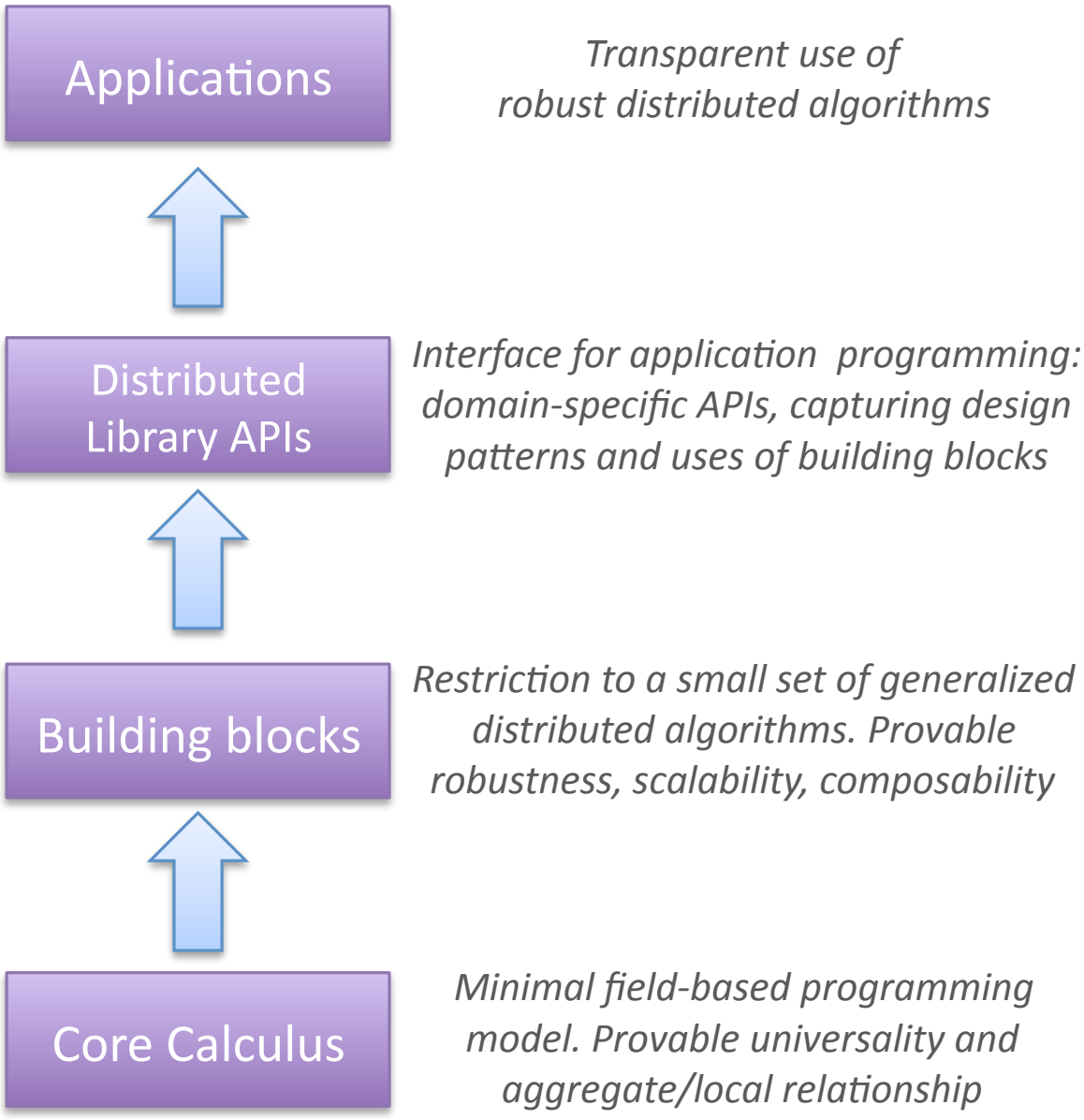


Figure 2.3: Layered approach for development of spatially-distributed systems via aggregate programming.

all suffer from some combination of design and implementation problems that render them impractical for widespread adoption. Besides Proto, which is probably the closest to a practical programming environment and whose problems have already been described in Section 2.5.1, others, such as [VPB12], have only minimal implementation.

Alignment

2.6 Engineering and tools

Despite the efforts of addressing specific aspects of self-adaptability and openness in pervasive contexts [MMTZ06, BCD+06, KR08], yet a comprehensive engineering framework is missing. The hardest challenge is due to the fact that most of the complexity of such systems does not come from the individual behaviours, but rather from interaction [Weg97, GSW06, ORV06].

Multiple methodologies have been proposed to tackle complex pervasive systems [MCOV11, BW08, BMV09a, MN10, ZCF+11], and they always include *simulation* as a key step to realise what-if analysis prior to actual development, to both assess the general validity of the designed mechanisms and to fine tune system parameters. There are many kinds of simulation tools available: they either provide programming/specification languages devoted to ease construction of the simulation process, especially targeting computing and social simulation (e.g. as in the case of multi-agent based simulation [BMV09a, SGJ07, BMV09b, NHCV07, Skl07]), or they stick to quite foundational computing languages to better tackle performance, mostly used in biology-oriented applications [Pri95, Mur89, UP05, EMRU07]. None of the existing tools, however, aims at bridging the gap between these approaches, trying to extend the basic computing model of chemical reactions – still retaining its high performance – toward ease applicability to complex situated computational systems.

2.6.1 General purpose frameworks

MASON [LCRP+05], Repast [NHCV07], NetLogo [Skl07] and Swarm

2.6.2 Specific simulators

network simulators (the one, ns2) biological simulators (ask Sara)

Specific per-use simulators, gener

Part II

An integrated toolchain for pervasive ecosystems

3
Chemical-inspired engine

In the following, we present a discrete event simulation engine derived from the popular and successful Gillespie's stochastic simulation algorithm [Gil77], and in particular from its notable and more efficient extensions developed by Gibson-Bruck [GB00] and by Slepoy [STP08]. Gillespie's SSA is a discrete and stochastic method that intrinsically owns the event-driven properties [BBL$^+$11]: introduced to model chemical systems, nowadays it represents the basis of many simulation platforms, in particular those developed for the investigation of biochemical systems [PRSS01, Kie02, CG09, VB08, MV10, GPOS13, HSG$^+$06]. With significative extensions, it was recently used to model artificial systems grounding on computational models inspired to chemical natural systems and ecology [MVFM$^+$13]. Moreover, its optimised extensions [GB00, STP08] are very efficient, thus allowing really fast simulation runs.

3.1 Gillespie's SSA as an event-driven algorithm

First of all, we summarise the idea that the Gillespie's SSA is grounded on: a chemical system is modelled as a single space filled with molecules that may interact through a number of reactions describing how they combine. The instantaneous speed of a reaction is called propensity and depends on the kinetic rate of the reaction and on the number of molecules of all the reagents involved. For a reaction i of the form:

$$R_0 + R_1 \xrightarrow{r} P_0 + P_1 + \ldots + P_n$$

the propensity a_i is defined as:

$$a_i = r \cdot [R_0] \cdot [R_1]$$

where $[M]$ is the number of molecules of species M. Given that, the algorithm relies on the idea that the system can be simulated by effectively executing the reactions one by one and changing the system status accordingly. Every algorithm follows four main steps:

1. select the next reaction μ to be executed;
2. calculate the time of occurrence of μ according to an exponential time distribution and make it the current simulation time;

3. change the environment status in order to reflect this execution;

4. update the propensities of the reactions.

Such algorithm is event-driven in that it executes only one reaction/event at a time: it changes the state of the system and consequently the event list *i.e.*. which other reactions/events can be executed from there.

3.2 Gillespie's optimised versions

This algorithm has been improved in various works in literature, particularly notable is the work in [GB00] and [STP08]. Both works optimise the base algorithm in two phases: the selection of the next reaction to execute and the update of the reaction pool – pending event list – once an event has been executed. For the latter, they both rely on the concept of "dependency graph". A dependency graph (DG) is a statically created directed graph in which nodes are all the reactions in the simulated system, and arcs connect a reaction `r` to all those that depend on it, namely, those whose triggering time should be updated as *r* is executed. For instance, if `r` changes the concentration of some molecule `m`, all those reactions that use `m` are to be properly re-scheduled as soon as `r` is fired. Even though this optimisation does not affect the execution time in the worst case, it offers great benefits in the average case, since most of the reactions are not interdependent.

The selection of the next reaction to execute is where those improved algorithms differ most. In Slepoy's work, the author divides reactions in groups based on their propensity: if p_{min} is the lowest propensity, the first group contains those reactions whose propensity ranges from p_{min} and $2p_{min}$, the second those between $2p_{min}$ and $4p_{min}$, and so on. Give those groups, an algorithm called "Composition-Rejection" (CR) is applied. In the composition phase, a group is randomly selected in logarithmic time; in the rejection phase a reaction is chosen within a group by throwing two random numbers: the first one, between 0 and the number of reactions in the group, identifies a reaction; the second one, between 0 and the maximum propensity allowed for the group, is used to decide whether or not to actually execute the selected reaction. In case the reaction is rejected, the rejection procedure is re-applied. Given the way groups are built, there is at least a 0.5 probability that the rejection procedure concludes at each attempt. On average, the CR algorithm requires five random numbers to be thrown. This algorithm requires logarithmic time to select the next reaction with respect to the total number of groups, but the author argues which, since in most biological scenarios this number is constant, then the algorithm runs effectively in constant time. In Gibson-Bruck, the same selection operation is made by computing a putative execution time for each reaction, and then using a binary heap data structure to sort them. This way, the next reaction to execute is always the root node of the tree, and the selection is performed in constant time. However, once the reaction has been executed, all the dependent reactions must be updated and re-sorted in the tree. In the worst case, this takes logarithmic time with respect to the number of reactions.

3.3 From SSA to full fledged DES

Both the Gisbon-Bruck's and Slepoy's schedulers are granted to correctly simulate a Poisson process. However, in order to build a full-fledged DES engine, we must offer the possibility to schedule also non-Markovian events. Imagine, for instance, that we want to simulate an agent that does an action every fixed time interval (e.g. a man walking). This, clearly, is not a memoryless process. We argue that Gibson-Bruck offers a more suitable base for a general purpose DES: in fact, its next reaction choosing mechanism is orthogonal to the way the putative times are computed. This intrinsic feature allows to neatly separate the generation of times (namely, the time distribution of each event) and the actual scheduling (choice of which event should run next). We chose the Next Reaction Method for ALCHEMIST, and, consequently, the main simulation algorithm follows the basic steps listed in Algorithm 1.

Algorithm 1 Simulation flow in ALCHEMIST

```
1:  cur_time = 0
2:  cur_step = 0
3:  for each node n in environment do
4:      for each reaction nr in n do
5:          generate a new putative time for nr
6:          insert nr in DIPQ
7:          generate dependencies for nr
8:  while cur_time < max_time and cur_step < max_step do
9:      r = the next reaction to execute
10:     if r's conditions are verified then
11:         execute all the actions of r
12:         for each reaction rd which depends on r do
13:             update the putative execution time
14:     generate a new putative time for r
```

A second feature that we need in order to shift the paradigm from pure chemistry towards higher expressiveness is the possibility to simulate multiple, separate, interacting and possible mobile entities. This requirement can be partly addressed by the notion of intercommunicating compartments [CG09, VB08, MV10, GPOS13], in a way that allows to also model systems characterised by a set of connected volumes and not only a unique chemical solution. The hardest challenge in simulating multiple compartments with an efficient SSA is in improving the dependency graph: reactions that could be interdependent but happen on separated compartments should not be marked as interconnected within the dependency graph. Mobility makes everything even harder, since it may lead to the creation and disruption of communication channels between compartments, and consequently to simulation-time changes in the structure of such dependency graph. Summarising, supporting dynamic environments with multiple mobile compartments require the dependency graph to become a dynamic data structure, which cannot be pre-computed

at the simulation initialisation and kept static.

3.3.1 Dynamic Dependency Graph

There are multiple ways to conceive such dynamic data structure. Ideally, it would be possible to just drop the optimisation or reuse the classic definition, in which case the triggering of a reaction in whichever compartment would cause the recalculation of the status of each potentially dependent event in every compartment. This approach would lead to a massive performance impact, since the dependency computation is the most expensive operation. As evidence, in [STP08] some charts show the difference between Gibson-Bruck and Slepoy et al. algorithms with and without a dependency graph: the result is very strongly in favour of the former.

A possibility for efficiently adapting a dependency graph to a network of compartments could be to define the input and output contexts for each reaction, namely the places where reactions respectively "read" their reactants and "write" their products. Multiple contexts could be defined, we propose to adopt three levels: `local`, `neighborhood` and `global`. In a purely chemical simulation, all the reactions have a `local` input context and may have either `neighborhood` or `local` output context, depending on whether or not they send molecules towards other compartments.

Let $dep : R^2 \longrightarrow boolean$ be the function that is used to build the static dependency graph in every SSA: given two reactions `r1` and `r2`, $dep(\texttt{r1}, \texttt{r2})$ returns `true` if the propensity of `r2` may be influenced by the execution of `r1`. In a multi-compartment scenario, `r1` influences `r2` (there is an oriented edge in the dependency graph connecting `r1` to `r2`) iff $dep(\texttt{r1}, \texttt{r2})$ is `true` and:

- `r1` and `r2` are on the same node OR
- `r1`'s output context is `global` OR
- `r2`'s input context is `global` OR
- `r1`'s output context is `neighborhood` and `r2`'s node is in `r1`'s node neighbourhood OR
- `r2`'s input context is `neighborhood` and `r1`'s node is in `r2`'s node neighbourhood OR
- `r1`'s output context and `r2`'s input context are both `neighborhood` and the neighbourhoods of their nodes have at least one common node.

The filters listed above greatly compact the number of edges of a dependency graph in most scenarios, with great benefits on the engine performance. On top of this finer-grain locality concept, if the model supports compartment mobility, the dependency graph must support the dynamic addition and removal of reactions.

Adding a new reaction implies to verify its dependencies against every reaction of the system. In case there is a dependency, it must be added to the dependency graph. Removing a

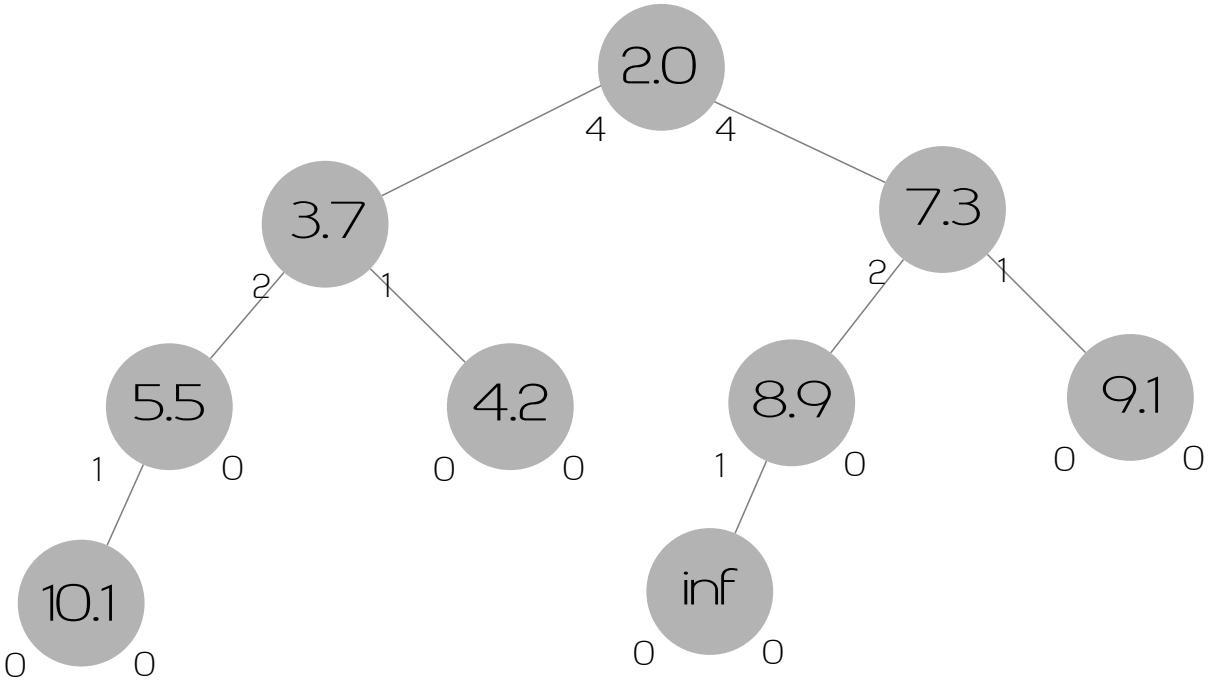


Figure 3.1: Indexed Priority Queue extended with descendant count per branch

reaction `r` requires to delete all dependencies in which `r` is involved both as influencing and influenced. Moreover, in case of a change of system topology, a dependency check among reactions belonging to nodes with modified neighbourhood is needed. It can be performed by scanning them, calculating the dependencies with the reactions belonging to new neighbours and deleting those with nodes which are no longer in the neighbourhood.

3.3.2 Dynamic Indexed Priority Queue

An issue that arises with addition and removal of nodes from the simulation is the possible unbalancing of the scheduling queue, that in the original work is realised as a binary tree of reactions, whose main property is that each node stores a reaction whose putative time of occurrence is lower than each of its sons.

Our idea is, for each node, to keep track of the number of descendant per branch, having in such way the possibility to keep the tree balanced when adding nodes. In Figure 3.1 we show how the same IPQ drawn in [GB00] would appear with our extension. Given this data structure, the procedures to add and remove a new node `n` are described respectively in Algorithm 2 and Algorithm 3, in which the procedure `UPDATE_AUX(n)` is the same described in [GB00].

Using the two procedures described above, the topology of the whole tree is constrained to remain balanced despite the dynamic addition and removal of reactions.

Algorithm 2 Procedure to add a new node `n`

```
1: if root does not exist then
2:     n is the new root
3: else
4:     c ← root
5:     while c has two descendants do
6:         if c.right < c.left then
7:             dir ← right
8:         else
9:             dir ← left
10:        add 1 to count of dir descendants
11:        c ← c.dir
12:        if c has not the left child then
13:            n becomes left child of c
14:            set count of left nodes of c to 1
15:        else
16:            n becomes right child of c
17:            set count of right nodes of c to 1
18:    UPDATE_AUX(n)
```

Algorithm 3 Procedure to remove a node `n`

```
1: c ← root
2: while c is not a leaf do
3:     if c.left > c.right then
4:         dir ← left
5:     else
6:         dir ← right
7:     subtract 1 to count of dir descendants
8:     c ← c.dir
9: if c ≠ n then
10:    swap n and c
11:    remove n
12:    UPDATE_AUX(c)
13: else
14:    remove n
```

4
Meta-meta model

The complexity of the systems we want to design is achieved by the following set of common key properties:

- situatedness – they deal with spatially- and possibly socially-situated activities of entities, and should therefore be able to interact with a limited portion of the surrounding world and contextualise their behaviour accordingly;
- adaptivity – they should inherently exhibit properties of autonomous adaptation and management to survive contingencies without external intervention, global supervision, or both;
- self-organisation – spatial and temporal patterns of behaviour should emerge out of local interactions and without a central authority that imposes pre-defined plans.

Among the many natural metaphors one can use as inspiration for modelling and developing artificial systems with the above properties [ZV11], we consider chemistry following a series of work in the field of pervasive computing [VCMZ11, VZ10, ZCF+11]. We argue that there are three main issues to be resolved in order to build a meta-model that can be sufficiently expressive for our purpose starting from a purely chemical model:

1. the concept of environment where agents are situated and can move is missing in a model that considers only intercommunicating chemical compartments;
2. the only available mean for changing the system status is the execution of a reaction;
3. the only data item that chemical reactions can manipulate are molecules' concentrations, namely numbers connected to a particular token.

In the following discussion, we will use interchangeably compartment/agent/node and reaction/events as synonyms. As first step, we introduce the environment, absent in chemistry-derived SSAs, as first class abstraction. The environment has responsibility to provide, for each compartment, a set of compartments that are its neighbours. The rule which is applied to determine whether or not a node belongs to another node's neighbourhood can be arbitrarily complicated.

Also, it is responsible of exposing possible physical boundaries, namely, to limit the possible movements of compartments situated within the environment.

The fact that reactions are the only abstraction the modeller can rely upon in order to let the simulated system progress is not a difficult problem by itself. In fact, nothing prevents to widen the generality of a reaction by defining it as: "a set of conditions that, when matched, trigger a set of actions on the environment".

With this definition in mind, a condition is a function that associates to each possible state of the environment a numeric value ranging from zero to positive infinity. If such value is zero, the event can not be scheduled; otherwise, it is up to the reaction to interpret the number: it can influence or not the time at which the reaction will be scheduled, depending on the specific reaction implementation. In case we desire to re-build the original chemical model, we would define a condition for each of the molecules on the left-hand side of the chemical reaction that return the number of molecules currently available in the local compartment. Also, we would define the reaction in such a way that it correctly interprets the number returned by the conditions as concentration of each reactant and correctly applies the rules for computing a propensity to be used to influence the reaction speed [Gil77].

In this framework, actions are arbitrary changes of the environment. In case of pure chemistry, the actions of a reaction would be one for each reactant (that must be removed from the local compartment) and one for each product (that must be added to the local compartment). In case of an extended model considering also multiple compartments, an action should be programmed to be responsible of transferring molecules from a node to a neighbouring one.

Both conditions and actions must expose the set of possible data items (molecules) that they may read or modify: this is necessary in order to allow the dependency graph to be built. Also, both conditions and actions must expose a context of the type `local`, `neighborhood` or `global`; it will be used internally to determine the input and output contexts for the reaction itself.

The reactions are responsible of computing their expected execution time. The engine may require such putative time to be updated in two cases: i) the reaction has just been executed or ii) a reaction on which this reaction depends on has been executed. In case of update required, the reaction should leverage a separately defined time distribution to compute the next putative execution time, possibly feeding the time distribution with a summary of the data gathered from conditions. In case of a Poisson process, a negative exponential time distribution initialised with $\lambda = r$ should be used, for instance. In case of a repetitive event, such as a timer, a Dirac comb may be used.

In this model the atomicity of the reactions represent a double edged sword: on the one hand, they allow for arbitrarily complex behaviours to be ordered and executed within a DES, on the other hand they make it difficult to model events that last in time, e.g. to simulate devices with limited computational power on which some complex task takes a not negligible amount of time to get completed. To support something similar, two reactions should be defined: one to trigger the start of the computation, and another to actually run it and complete.

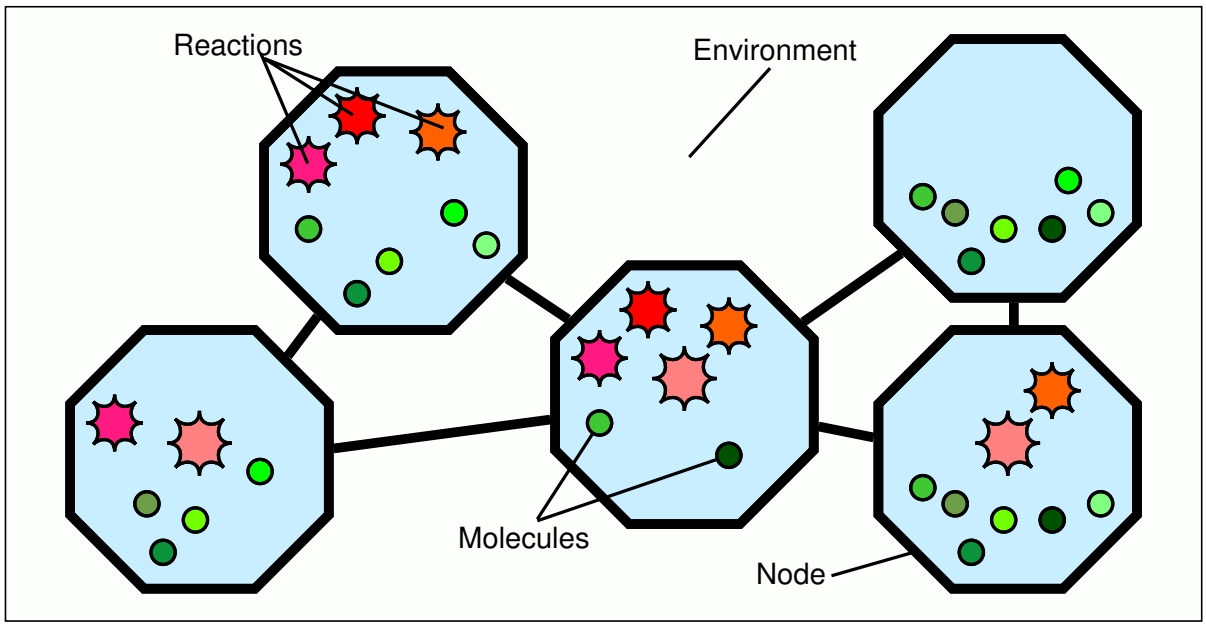


Figure 4.1: ALCHEMIST computational model: it features a possibly continuous space embedding a linking rule and containing nodes. Each node is programmed with a set of reactions and contains a set of structured molecules.

The low expressive power of the classical concentration is probably the hardest challenge to tackle when trying to extend a chemical-born meta model towards a richer world, where data items can be complex structures and not just simple numbers. We have found no trivial solution to this issue; instead, we propose to make the definition "concentration" depend on the actual meta-model: let "concentration" be "the data items agents can manipulate". Besides the trivial example of chemistry, where data items are integer numbers, let's consider a distributed tuple spaces model: in this case, the molecules would be tuples, and the concentration would be defined as "the set of tuples matching a certain tuple". Clearly, such flexibility comes with a cost: since the conditions and actions operating on different concentrations dramatically change their behaviour, for any possible class of data items the meta-model must be instanced with a proper set of conditions and actions that can act on such "concentration". We call this set of concentration-specific instances of conditions and actions an "incarnation". This sort of model inheritance justifies our double "meta" level: the model described is a meta-meta-model, an incarnation is a meta-model, and, finally, a specific scenario instancing one of such meta-models is a model.

A pictorial representation of the underlying meta-meta-model is shown in Figure 4.1. In this vision of the world, an environment is a multi dimensional space, continuous or discrete, which is able to contain nodes and which is responsible of linking them following a rule. The environment may or not allow nodes to move. Nodes are entities which can be programmed with

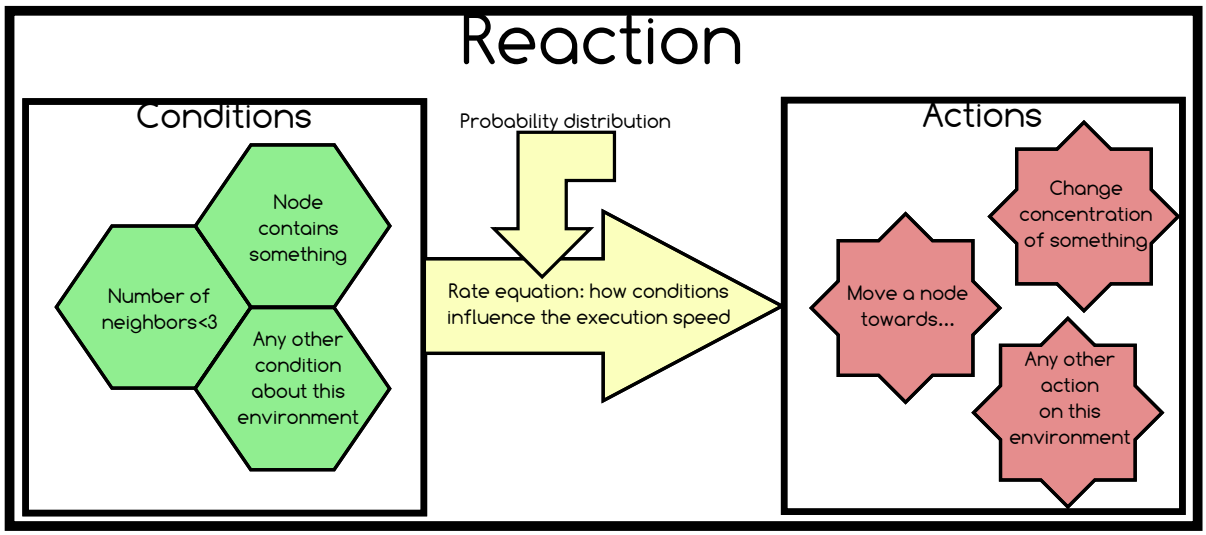


Figure 4.2: ALCHEMIST model of reaction: a set of conditions on the environment that determines whether the reaction is executable, a rate equation describing how the speed of the reaction changes in response to environment modifications, a probability distribution for the event and a set of actions, which will be the neat effect of the execution of the reaction.

a set of reactions, possibly changing over time. They also contain molecules, each one equipped with a data structure generalising on the concept of "concentration".

The concept of reaction is graphically depicted in Figure 4.2. It allows, for example, to model reactions which are faster if a node has many neighbours, or also reactions that resemble complex biological phenomena such as the diffusion of morphogenes during embryo development as described in [LIDP10]. It also allows to define which kind of time distribution to use to trigger reactions: this enables us to model and simulate systems based on Continuous Time Markov Chains (CTMCs), to add triggers, or also to rely just on classical discrete time "ticks".

5 Architecture

The whole framework has been designed to be fully modular and extensible. The whole engine or parts of it can be re-implemented without touching anything in the model, and on the other hand the model can be extended and modified without affecting the engine.

It is important to note that there is no restriction about the kind of data structure representing the concentration, which can in fact be used to model structured information: by defining a new kind of structure for the concentration, it is possible to incarnate the simulator for different specific uses. For example, by assessing that the concentration is an integer number, representing the number of molecules currently present in a node, ALCHEMIST becomes a stochastic simulator for chemistry. A more complex example can be the definition of concentration as a tuple set, and the definition of molecule as tuple template. If we adopt this vision, ALCHEMIST can be a simulator for a network of tuple spaces. Each time a new definition of concentration and molecule is made, a new "incarnation" of ALCHEMIST is automatically defined. For each incarnation, a set of specific actions, conditions, reactions and nodes can be defined, and all the entities already defined for a more generic concentration type can be reused.

5.1 Writing a simulation

In order to write a simulation, the user must have, or implement herself, an incarnation of ALCHEMIST, as described in Chapter 5.

As shown in Figure 5.1, the simulations are written in a specific XML language containing a complete description of environment and reactions. This code is interpreted in order to produce an instance of an environment: once it is created, no further interpretation is needed in order to run the simulation. This XML code is not meant to be directly exploited by users: the XML format itself is not exactly human friendly, and XML file is often considerably big (a few megabytes are considered to be normal) However, it is a very standard way of describing environments in a machine-friendly format. The idea behind this choice is that ALCHEMIST is flexible enough to be used in various contexts, each one requiring a personalised language and a different instantiation of the meta-model. It is up to the extensor to write a translation module from its personalised language to the ALCHEMIST XML. Of course, it is also possible to code the simulation behaviour directly with Java, although this way exposes the user to many more

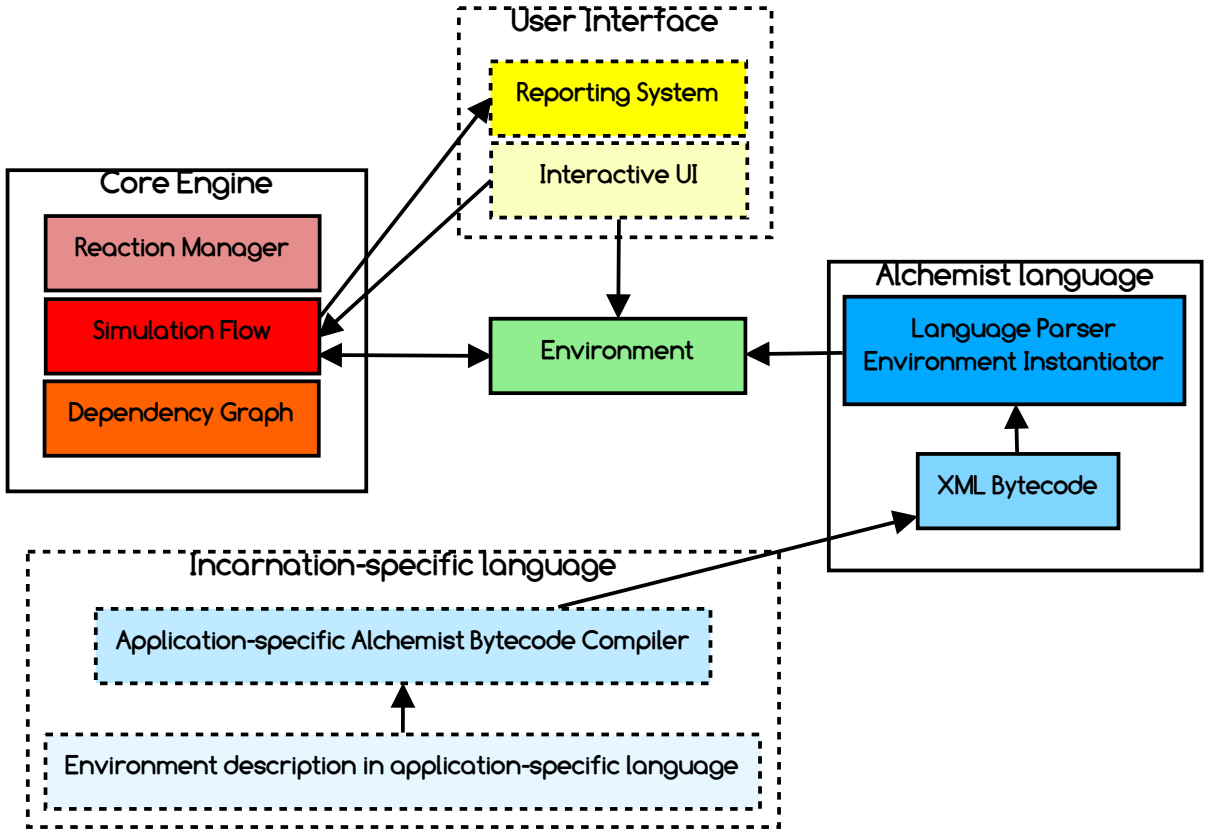


Figure 5.1: ALCHEMIST architecture. Elements drawn with continuous lines indicates components common for every scenario and already developed, those with dotted lines are extension-specific components which have to be developed with a specific incarnation in mind.

low level details.

5.2 Implementation details

The framework was developed from scratch using Java. Being performance a critical issue for simulators, we compared some common languages in order to evaluate their performance level. Surprisingly, Java performance are at same level of compiled languages such as C/C++ [BSB+03, ORAI11]. The Java language was consequently chosen because of the excellent trade off between performance, easy portability and maintainability of the code, and the high-level support for concurrent programming at the language level. The COLT Java library[1] provided us the mathematical functions we needed. In particular, it offers a fast and reliable random number generation algorithm, the so called Mersenne Twister [MN98].

ALCHEMIST is still actively developed and currently consists of about 200 classes for about 20'000 lines of code. Though still in beta version, it is released with GPL license as open source [2].

[1] http://acs.lbl.gov/software/colt/

[2] http://alchemist.apice.unibo.it.

6 Performance

It is possible to evaluate and compare the performances of ALCHEMIST with respect to some known MABS. We exemplify it considering Repast, which we used to developed an alternative simulation for the case study presented in Chapter 15[1]. There are some important facts that deserve discussion here.

First, since there is no built-in support for stochastic simulation in Repast, we choose to collapse the last five laws of Figure 15.1 into a single code path, and the same was made for ALCHEMIST by defining a new action. In that way, we were able to avoid for this very specific case the need of a full fledged dependency graph, because there will always be exactly one **source** and one **field** per sensor, and no reactions need to be enabled or disabled. This crippled most of the effectiveness of ALCHEMIST's dependency graph, which is indeed a source of optimisation not natively existing in Repast—developing a dependency graph for stochastic simulation in Repast is out of the scope of this thesis, though it would be an interesting subject for future work. On the other hand, it would have been unfair to compare our optimised version against just a plain Gillespie's algorithm built upon Repast.

The second important point is that we choose to encode all the data in both Repast and ALCHEMIST as an array of double values instead of real tuples. For ALCHEMIST, this meant that we developed a new incarnation for the precise scope of this performance test. The choice of encoding data that way made things faster (no matching required), but also less general. This was done because the SAPERE incarnation of ALCHEMIST, which allows us to write the laws as in Figure 15.1, requires a matching system that is not easily portable to Repast.

We choose the configuration of Figure 15.3 and we run multiple simulations varying the number of agents per group, in order to evaluate how the two systems scale with the problem size. We measured the running time required to our testbed to run the simulation from the time zero to the time 100. No graphical interface were attached to the simulators while running the batch, in order to evaluate only the raw performance of the engine. The system we used was an Intel Core i5-2500 equipped with 8GB RAM, with Sabayon Linux 7 as operating system and Sun Java HotSpot™ 64-Bit Server version 1.6.0_26-b03 as Java Virtual Machine. Results are shown in Figure 6.1.

[1] The source code of the simulation we developed is publicly available at `http://apice.unibo.it/xwiki/bin/view/Alchemist/JOS/`.

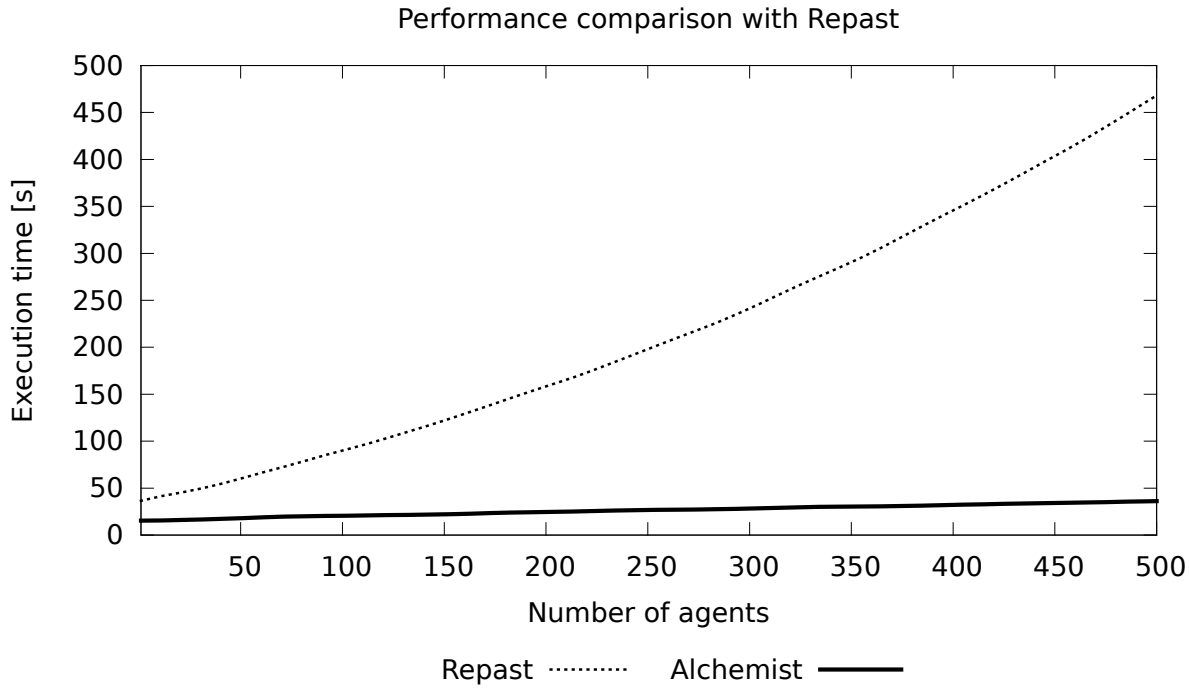


Figure 6.1: Chart showing the performance scaling of ALCHEMIST

Results show the simulator built upon the ALCHEMIST framework to be at least twice faster and to scale better than the one built on Repast. Being the dependency graph optimisation cut off as explained above, the reasons for such a difference can lay on the internal scheduler of the engine or in the optimisations in the model. For the first point, we used the default scheduler of Repast Symphony, which is a binary heap implemented through a plain Java array, while our implementation relies on the algorithm and data structures already presented in Chapter 3, so there is not a substantial efficiency difference. For the latter point, a big difference in terms of performances is due to the heavy optimizations of the neighbourhoods of the default ALCHEMIST continuous environment. Since the concept of neighbourhood was part of the computational model, it was possible to adopt caching strategies in order to ensure a fast access to the neighbourhood and a quick execution of the operations on it. This is probably the component which gives the highest impact in this case, since most interactions occur among an agent and its neighbourhood.

7 Collocation in literature

7.1 ALCHEMIST as a DES

ALCHEMIST is a discrete event simulator (DES), since it combines a continuous time base with the description of system dynamics by distinguished state changes [Zei76]. Since it allows for in-simulation modifications of the environment, it can be considered to belong to the fourth generation of DESs according to [BW08]. The work in [Pol89] surveys the classical DES approaches: according to it, ALCHEMIST belongs to the category of simulators featuring "internal clock", "next-event time advance", and adhering to the "Event-scheduling World View", namely, the simulator handles events and is concerned with their effect on system states.

Apart from the meta-model adopted, the main innovative aspect of ALCHEMIST with respect to the general DES approach is its ability of optimising the "compile current event list" stage [Pol89], which ALCHEMIST quickly executes incrementally by means of the management of dependencies that the adoption of a chemical-like model enables.

As far as the simulator model is concerned, instead, the class of DES more related to our approach are those commonly used to simulate biological-like systems. A recent overview of them is available in [EMRU07], which takes into account: DEVS [Zei84], Petri Nets [Mur89], State Charts [Har87] and stochastic π-calculus [Pri95]. Such an overview however emphasises that all such models require a considerable effort to map biological components into abstractions of the chosen formalism: this is because none of them was specifically developed with biology or bio-inspiration in mind. As described in Chapter 4, our model is meant to overcome such limitations, since all the enhancements to the basic chemical model we support can be seen as a generalisation of the abstractions of the works presented in [EMRU07], and add to them the possibly of customising with much greater flexibility a simulation to the scenarios of bio-inspired computational systems.

We should finally note that the DES approach typically contrasts the use of mathematical techniques (e.g. modelling the system by differential equations). However, the possible different choice of translating a system model to ordinary or partial different equations – which can be solved numerically in a time considerably shorter than a set of stochastic simulations – is shown to be impractical in our case. This path, explored for example by [MTUG09], can provide good results for dynamics that progress at more or less the same speed, and in which abundance of

species (data-items or agents in our case) is so high that it can be relaxed to real-valued variables [EMRU07]. This is not the case for many scenarios even in system biology (see, for example, [Cow00, UP05]), not to mention scenarios like pervasive computing where reaction rates do not change continuously, and where the effect on an even small number of agents can be key to a given system evolution.

7.2 ALCHEMIST as a MABS

Even though chemical-inspired, the meta-meta-model described in Chapter 4 holds evident relationships with multi-agent-based simulation (MABS) approaches.

According to [BMV09a], agent-based platforms for simulations can be split in three categories: general purpose frameworks with specific languages (such as NetLogo), general purpose frameworks based on general purpose programming languages (such as Repast [NHCV07]) and frameworks which provide an higher level linguistic support, often targeted to a very specific application (e.g. [WBH06]). Each approach has clearly its own advantages and weaknesses. Usually, the more general purpose is the language, the wider is the set of possible scenarios, and the wider is also the gap between the language and the simulated model. This means that an higher level language allows the user to create and tune its simulations in an easier way, on the other hand it often restricts the generality of the tool.

ALCHEMIST is meant to provide a set of meta-models, possibly each with its own domain specific language, still maintaining the possibility to extend or re-implement certain abstractions using the general purpose Java language (defining a new "incarnation", namely a new meta-model).

7.2.1 Advantages

There is a set of applications which are better tackled by ALCHEMIST. In particular, ALCHEMIST is suitable for all those simulation scenarios in which agents have relatively simple behaviour, and the notion of agent-based environment plays instead a fundamental role in organising and regulating the agents' activities [BV07] by both enabling local interactions among the proactive entities [HVUM07] and enforcing coordination rules [MOV09], allowing the modeller to shift her focus from the local behaviour of the single agent to a more objective vision of the whole MAS [SGJ07]. The idea of dealing with a strong notion of environment in multi-agent systems has been deeply developed in a series of work: other than its importance in the simulation context [HVUM07], at the infrastructure level [VHR+07] and at the methodological level [MOV09], there have been proposals of meta-models such as A&A [ORV08] and infrastructures such as TuCSoN [OZ99] and CArtAgO [RPV11]. A common viewpoint of all these works is that the behaviour of those passive and reactive components structuring the environment (e.g. *artifacts* in A&A) is well defined in terms of rules expressed as declarative conditions-imply-actions fashion. Accordingly, ALCHEMIST is particularly useful either in computing systems following

the chemical inspiration [VCMZ11, VZ10, ZCF+11], or for agent-oriented systems where the role of the environment is key, up to be a very dynamic part of the whole system—where network nodes (or, in biological terms, compartments) can move, new nodes can be spawn or be removed from the system, and links can appear or break at runtime as happen e.g. in pervasive computing scenarios [ZCF+11].

7.2.2 Limitations

Inevitably, as an attempt to build a hybrid between agent-based simulators and (bio)chemical simulators, some trade-offs had to be accepted, which ultimately makes ALCHEMIST less suited for certain classes of applications. In particular, a limitation is that ALCHEMIST's agent actions have to be mapped onto the concept of reaction. On the one hand, this makes it rather straightforward to create reactive memoryless agents [BMV09a], whose goal is just to perform rather easy computations resulting in the creation, deletion or modification of information items in the environment. In fact, since it is allowed to program different nodes with different reactions, it is easy to code reactive and context-dependent agents. On the other hand, there is neither out-of-the-box facility nor any high level abstraction useful to define intelligent agents [BMV09a]. The simulator is able to run them provided the user manually writes their whole behaviour as a single Java-written reaction—and properly specifies dependencies with other reactions. Although possible in principle, performing this task too frequently is likely breaking the ALCHEMIST abstraction, making the programmer losing the nice declarative approach that chemistry endorses, and possibly hampering the optimisation techniques that motivated ALCHEMIST.

8 Meta-model agnostic features

This chapter analyses the main features of ALCHEMIST that can be exploited regardless the specific "incarnation". They add a great value to the tool, since they can be used immediately after the creation of a novel meta-model, with no need to specifically write any new code besides some minimal glue.

8.1 Distributed statistical analysis

As we have seen in Section 2.6, simulation represents a key step in the engineering of pervasive systems. Being a simulation basically a sampling over an enormous probability space, obvious questions accompany these procedures:

- how reliable are the obtained values?
- How is the number of performed simulations chosen?
- How many simulations are required in order to state system properties with a certain degree of confidence?

Moreover, there is frequently a lack of decoupling between the model specification and the definition of the system's properties of interest: they are often embedded in the model, and their values are obtained via logging operations performed during the simulation process.

This section presents a tool meant to face these challenges, obtained chaining ALCHEMIST, with MULTIVESTA[1] [SV13], a recently proposed lightweight tool extending VESTA [SMA05] and PVESTA [AM11] which allows to enrich existing discrete event simulators with automated and distributed statistical analysis capabilities. The result is thus a statistical analysis tool tailored to chemical-inspired pervasive systems. The benefits obtained by chaining the simulator with MULTIVESTA are:

1. a language (MULTIQUATEX) to compactly and cleanly express systems properties, decoupled from the model specification;

[1] `http://code.google.com/p/multivesta/`

2. the automated estimation of the expected values of MULTIQUATEX expressions with respect to *n* independent simulations, with *n* large enough to respect a user-specified confidence interval;

3. an interactive plot as well as the generation of gnuplot input files to visualize the obtained results;

4. a client-server architecture to distribute simulations.

MULTIVESTA is independent on the model specification language: it only assumes that discrete event simulations can be performed on the input model. As described in [SV13], the tool offers a clean interface to integrate existing discrete event simulators, enriching them with a property specification language, and with efficient distributed statistical analysis capabilities.

8.1.1 MULTIVESTA and MULTIQUATEX

MULTIVESTA performs a statistical (Monte Carlo based) evaluation of MULTIQUATEX expressions, allowing to query about expected values of observations performed on simulations of probabilistic models. A MULTIQUATEX expression may regard more measures of a model, in which case the same simulations are reused to estimate them, thus improving the performance of the analysis tasks. Moreover, the tool has a client-server architecture allowing to distribute the simulations on different machines. A detailed description of MULTIQUATEX and of the procedure to estimate its expressions is out of the focus of this work, the interested reader can deepen her knowledge reading [SV13, AMS06]. Before defining a MULTIQUATEX expression, it is necessary to specify the state characteristics to be observed. This model-specific step "connects" MULTIQUATEX with the model. The state observations are offered via the `rval`(*obs*) predicate which returns a real number for each observation, specified by the string parameter *obs*.

A MULTIQUATEX expression consists of a set of definitions of *parametric recursive temporal operators*, followed by a list of `eval` *clauses*. Each `eval` clause relies on the defined temporal operators, and specifies a property whose expected value must be evaluated.

In order to evaluate a MULTIQUATEX expression, MULTIVESTA performs several simulations, obtaining from each a list of samples (real numbers). One sample for each `eval` clause is obtained, thus all the queried measures are evaluated using the same simulations, improving performance. Based on the samples obtained from simulations, MULTIVESTA estimates MULTIQUATEX expressions with respect to two user-provided parameters: α and δ. Considering the case of simple expressions with just one `eval` clause, the estimations are computed as the mean value of the *n* samples obtained from *n* simulations, with *n* large enough to grant that the size of the $(1-\alpha)*100\%$ *Confidence Interval* (CI) is bounded by δ. In other words, if a simple MULTIQUATEX expression is estimated as $\bar{x}$, then, with probability $(1-\alpha)$, its actual expected value belongs to the interval $[\bar{x}-\delta/2, \bar{x}+\delta/2]$. The case of expressions with multiple `eval` clauses is similar, with the note that the `eval` clauses may regard values of different orders of magnitude, and thus the user may provide a list of δ rather than just one. After having obtained

a sample for every `eval` clause from a simulation, these values are used to update the means of the samples obtained from previous simulations (one mean per `eval` clause). If the CIs have been reached for every `eval` clause, the evaluation of the expression is terminated, otherwise further simulations are performed. Note that each `eval` clause may require a different number of simulations to reach the required CI. Once the CI of an `eval` clause has been reached, such `eval` clause is ignored in eventual further simulations performed for other `eval` clauses.

8.1.2 Integrating MULTIVESTA and ALCHEMIST

In order to chain the tools, some steps, have been tackled once and for all, while others are model-specific. Essentially, in order to allow the interaction with MULTIVESTA, ALCHEMIST has to fulfill two requirements:

1. the ability to advance the simulation in a step-by-step manner (which is provided by the `playSingleStepAndWait` method in `ISimulation` interface);
2. the ability to analyze the model status after each simulation step, providing measures in form of real numbers about properties of interest.

Since both MULTIVESTA and ALCHEMIST are Java-based, their interaction has been easily realized by subclassing the `NewState` class of MULTIVESTA. The obtained `AlchemistState` class is sketched in Listing 8.1, where unnecessary details are omitted. The new class contains some ALCHEMIST-specific code, providing MULTIVESTA with the simulation control and proper entry points for the analysis.

In the constructor (lines 6-10), the superclass initialization is done by a simple `super()` call. The remaining code initializes ALCHEMIST-specific parameters such as the maximum time or steps to simulate. Those are in general not required, since MULTIVESTA is able to detect when the analysis requirements have been met, and consequently stop the simulation flow. The method `setSimulatorForNewSimulation()` is depicted in lines 12-20. The method is invoked by MULTIVESTA before performing a new simulation to (re)initialize the status of the simulator, providing to it a new random seed. In lines 22-24, `performOneStepOfSimulation()` is provided: resorting to the ALCHEMIST method `playSingleStepAndWait()`, it allows MULTIVESTA to order the execution of a single simulation step.

In order to inspect the simulation state, the `rval()` method defined in lines 26-32 is invoked. The argument specifies the observations of interest. This method inspects the simulation state for all aspects common to any ALCHEMIST model, e.g., in the listing are sketched the current simulated time (line 27) and the number of performed simulation steps (line 28). Clearly, each ALCHEMIST model will have its own observations of interest.

Depending on the model at hand, it may be necessary to refine the model-independent observations exposed by `AlchemistState` with a set of model-specific ones. This can be done by simply instantiating the `IStateEvaluator` interface provided by MULTIVESTA, constituted by one method only.

Listing 8.1: AlchemistState extending MULTIVESTA's NewState class

```
1  public class AlchemistState<N extends Number, D extends Number, T> extends NewState {
2    private final long maxS;
3    private final ITime maxT;
4    private ISimulation<N, D, T> sim;
5    ...
6    public AlchemistState(final ParametersForState params) throws ...{
7      super(params);
8      final StringTokenizer otherparams = new StringTokenizer(params.getOtherParameters());
9    /* Initialization of Alchemist-specific parameters
10      and execution environment resorting to otherParams */
11   }
12   ...
13   public void setSimulatorForNewSimulation(final int seed) {
14     /* Stop current simulation, create a new one. */
15     ...
16     sim.stop();
17     ...
18     env = getFreshEnvironment(seed);
19     sim = new Simulation<>(env, maxS, maxT);
20     ...
21   }
22   ...
23   public void performOneStepOfSimulation() {
24     sim.playSingleStepAndWait();
25   }
26   ...
27   public double rval(final String obs) {
28     if(obs.equals("time")) return getTime();
29     if(obs.equals("steps")) return getStep();
30     //other model-independent observations
31     ...
32     return getStateEvaluator().getVal(obs, this);
33   }
34 }
```

8.1.3 Example

This section discusses the analysis performed on a crowd steering scenario, similar to the one which will be presented in detail in Chapter 15. In such scenario, two group of people are inside an indoor environment, and are steered towards two point of interest (POI) such that their ideal trajectory intersect, creating a crowd. The steering system is implemented leveraging the SAPERE meta-model, and as such data items in this scenario are called `LSAs`. This example is not meant to provide significant scientific insights on crowd steering (thoughts about it will be provided in the remainder of this thesis), but rather to act as toy-example to demonstrate the flexibility and possibilities of the chaining of ALCHEMIST and MULTIVESTA.

The outcome of the analysis is summarized in the three charts of Figure 8.1 (obtained using the gnuplot input files provided by MULTIVESTA), showing, at the varying of the simulated time, the expected values of: the number of people which have reached their point of interest (top), the average number of connections of the devices (middle), and the number of LSAs in the system (bottom).

Listing 8.2: The evaluated parametric multi-expression (*MainMQ*)

```
1  people@POI(x) = if{s.rval("time") >= x}
2        then s.rval("bPOI")
3        else #people@POI(x)
4     fi;
5  people@RPOI(x) = // similar to people@POI
6  people@LPOI(x) = // similar to people@POI
7  avgConn(x) = if{s.rval("time") >= x}
8        then s.rval("degree")
9        else #avgConn(x)
10    fi;
11 LSAs(x) = if{s.rval("time") >= x}
12       then s.rval("LSAs")
13       else #LSAs(x)
14 fi;
15 eval parametric(E[people@POI(x)],E[people@RPOI(x)],
   E[people@LPOI(x)],E[avgConn(x)],E[LSAs(x)],x,0.0,1.0,50.0);
```

The three charts have been obtained by evaluating *MainMQ* of Listing 8.2, having 5 parametric temporal operators (lines 1-9): `people@RPOI` and `people@LPOI` regard the number of people which have reached, respectively, the POI on the right and the one on the left, while `people@POI` counts instead how many people have reached their destination. The fourth temporal operator (`avgConn`) regards the connectivity degree of the devices. Finally, `LSAs` regards the number of LSA in the system, giving insights on the amount of memory required to sustain the system. The temporal operators are analysed at the varying of the simulated time from 0 to 50 seconds, with step 1 (line 10). Thus the analysis consisted in the estimation of the expected values of the 5 temporal operators instantiated with 50 parameters, for a total of 250 expected values.

A high degree of precision has been required: α has been set to 0.01, while the δ values (the size of the CIs) have been chosen considering the orders of magnitude of the measures: 0.5 for the instances of `people@POI(x)`, `people@RPOI(x)` and `people@LPOI(x)`; 0.05 for `avgConn(x)`; and 3 for `LSAs(x)`. To reach such a level of confidence, the tool ran approximately 2500 simulations, requiring less than a hour. The discussed confidence intervals are depicted in the aforementioned charts: the two lines drawn above and below the central lines represent the obtained CIs of the expected values, thus indicating the intervals in which the actual expected values lie with probability 0.99.

The top chart regards the first 3 temporal operators: the top plot refers to the number of people in total which have reached their target POI, while the two almost over-imposed lower plots regard the number of people at the right POI and at the left POI. All the three measures

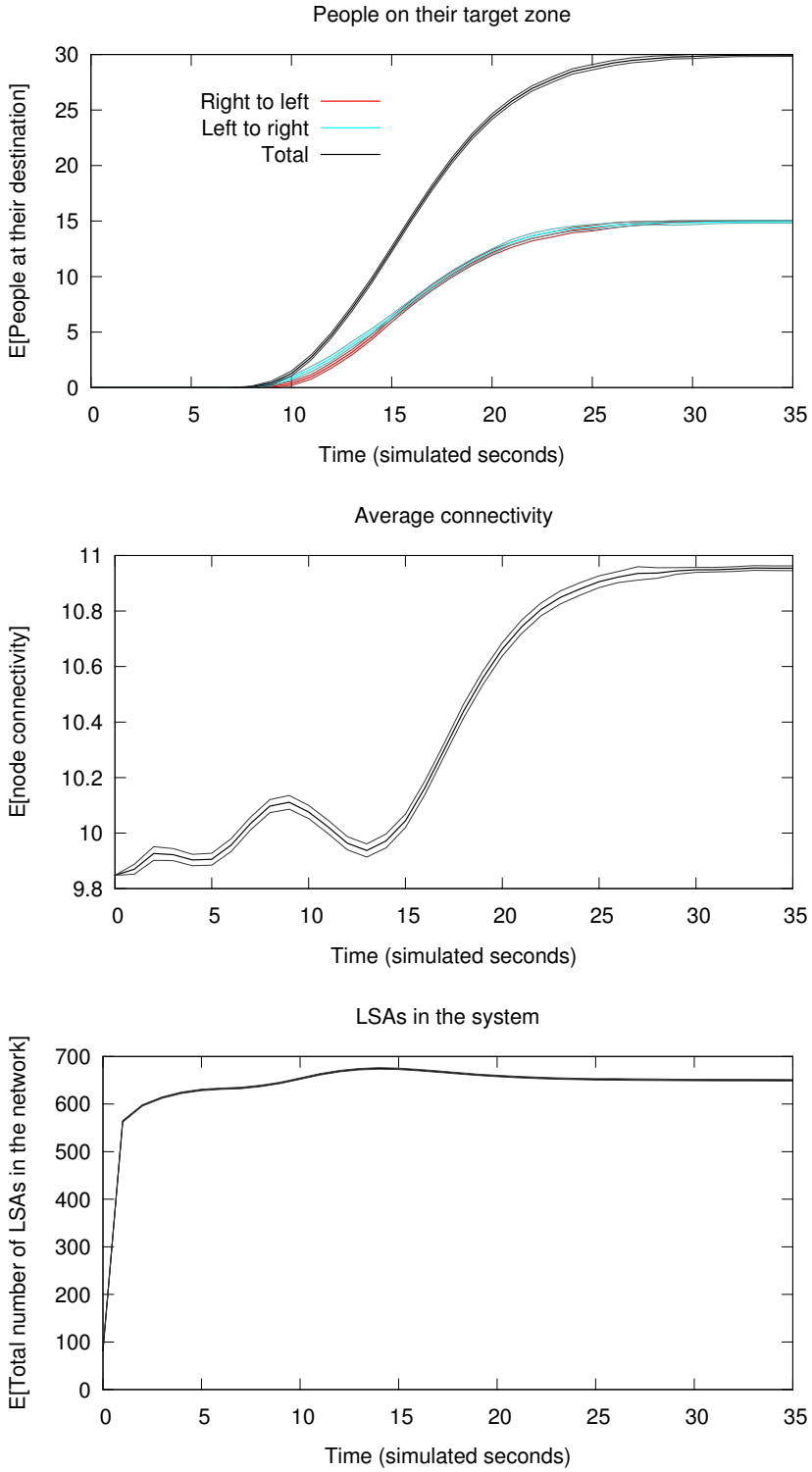


Figure 8.1: Analysis of the crowd steering scenario: (top) number of people at the POIs, (middle) average number of connections per device, (bottom) number of LSAs in the system.

under analysis produced monotonically increasing sigmoid curves. We can deduce from this behavior that there are no systemic errors in the crowd steering system, such as people with no or wrong suggested final destination, in which case we would have noticed one or more flatted zones flawing the sigmoid curve. We note also that after around 30 simulated seconds all the people have reached their target, and that it takes around 10 seconds for the fastest walkers to get to their destination. Note that, despite that the analysis has been performed for 50 simulated seconds, the charts presented in Figure 8.1 have been cut at time 35 to better show the most relevant results: in fact, the system reaches stability after this time, and no event of interest happens later. Moreover we can also notice that people going from left to right are slightly faster than the other group. This difference, due to the asymmetric positioning of people in the center of the scenario, is hardly visible and would have been impossible to spot with a lower precision analysis.

The middle chart shows the evolution in time of the average number of connections of the devices (i.e., both sensors and smartphones). Since each device is considered to be connected to all those within a range of 1.5 meters, it also gives us a hint about the crowding level. There is a noticeable peak at around 8 seconds: it is due to a high number of people approaching the central group. After that, the peak disappears when most people overtook the obstacle and are walking towards their POI. Finally, there is a growth: progressively, people reach their destination and tend to create a crowd.

The bottom chart shows the number of LSAs in the whole system, indirectly giving hints on the global memory usage. Once the system is started, there is a very quick growth, due to the gradient being spread from the sources to the whole sensors network and to the LSAs produced by the crowd-sensing. The system reaches a substantial stability after a couple of seconds. From that point on, the number of LSAs has very little variations: the system has no "memory leak", in the sense that it does not keep on producing new LSAs without properly removing old data.

8.1.4 Performance Assessment

All the experiments of the previous section have been run on a machine equipped with four Intel® Xeon® E7540 and 64GB of RAM, running Linux 2.6.32 and Java 1.7.0_04-b20 64bit.

In order to measure the performance scaling of the tool, we ran our analysis multiple times varying the number of MULTIVESTA servers deployed. Results are summarized by the dark line of Figure 8.2, showing that with only 1 server (ALCHEMIST does not have any parallel capability by its own), the analysis required almost 17 hours, while with more than 30 servers (enabled by the chaining with MULTIVESTA) it required less than an hour. For the considered scenario, by distributing simulations we have thus obtained a more than eighteen times faster analysis.

It is worth to note that such comparison of performance is affected by the statistical nature of the analysis procedure. Intuitively, when evaluating a MULTIQUATEX expression resorting to x or to $x+y$ servers, in both cases we initialize the first x servers with the same x seeds, generating thus the same x simulations. However, the remaining y servers are initialized with new seeds, and thus produce new simulations. Clearly, by having different simulations, we may obtain different

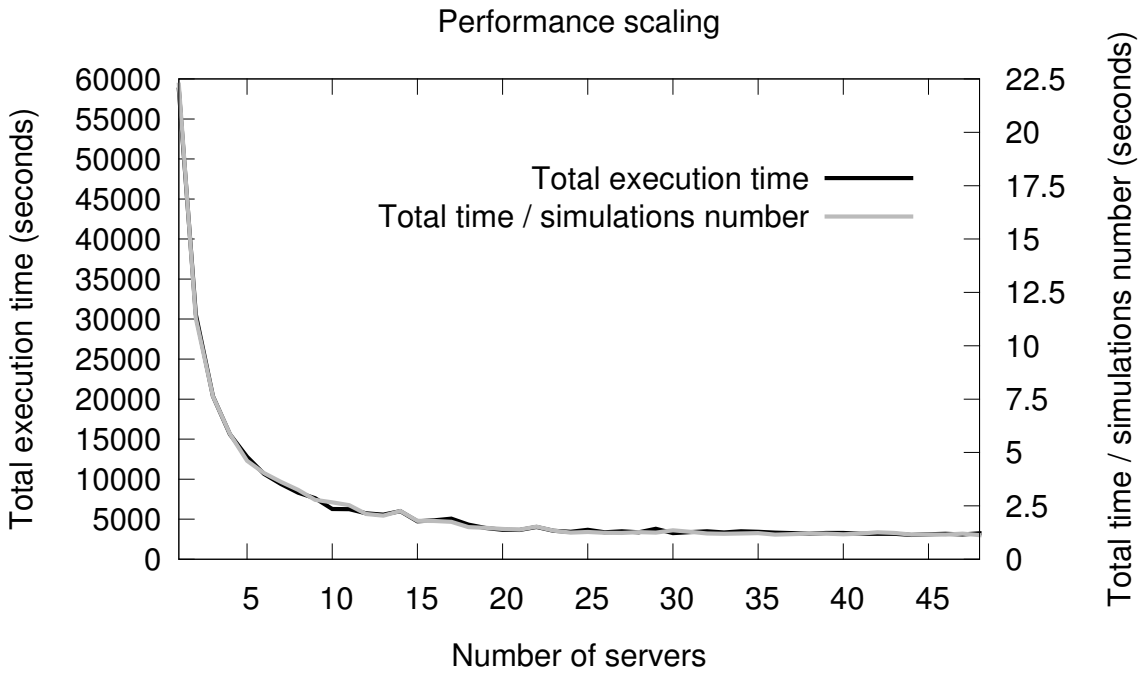


Figure 8.2: Performance scaling with of the number of running servers.

sample variances, requiring a different number of simulations. For this reason, the bright line of Figure 8.2 depicts the time analysis normalized with respect to the number of simulations (obtained dividing the overall analysis time by the number of performed simulations). The two lines of Figure 8.2 evolve accordingly, thus confirming the great performance gain brought by the distribution of simulations.

MULTIVESTA has a further important feature which dramatically reduces the analysis time: the reuse of simulations to estimate many expected values. The reusage happens on two levels:

1. expressions can be made parametric, and thus simulations can be reused for computing the same property at the varying of a parameter (e.g., at different time steps);

2. it is possible to write multi-expressions , and thus reuse simulations to evaluate multiple properties.

Moreover, by combining these two features, it is possible to reuse simulations for multiple parametric properties (e.g. *MainMQ* of Listing 8.2). As explicated by Table 8.1 (whose reported analysis have been performed resorting to 48 servers), parametric expressions (second column) allowed us to save a stunning 96% of execution time with respect to the simple expressions case without reuse of simulations (first column). Moreover, the parametric multi-expression feature (third column) allowed us to further cut down this time to a third. We can thus advocate that, by

Table 8.1: Time performance improvements reusing simulations (seconds).

	Expressions	Param. expressions	Param. multi-expressions
`people@POI`	28045.77	2567.26	n.a.
`people@RPOI`	15891.13	1114.86	n.a.
`people@LPOI`	15560.83	950.30	n.a.
`avgConn`	90111.29	1372.64	n.a.
`LSAs`	58630.45	2504.33	n.a.
Total time	208239.47	8509.39	3215.95

exploiting reuse of simulations, for the considered scenario we obtained a more than 64 times faster analysis. Clearly, the performance gains change depending on the considered scenario or properties. In fact, the more simulations are required, and the longer they take, the greater will be the advantage.

8.2 Real world maps

ALCHEMIST supports real-world environments. The base functionality lies in the possibility to load maps and simulate within them. Currently, the simulator supports OpenStreetMap[HW08], and it is able to load data in multiple OSM XML, compressed OSM XML and Protcolbuffer Binary Format (PBF). The latter is warmly recommended for both performance and map file size. Various websites provide ready-to-use extracts of the world map in PBF format for cities and whole regions, and arbitrarily sized extracts in OSM XML format can be obtained via public web API.

Once the map has been loaded, ALCHEMIST offers the possibility to enrich the simulations with the maps data, and in particular offers various ways to move nodes within the environment taking into account the characteristics of the map.

The first feature offered is about the initial node displacing. When adding a node to the environment, in fact, it is possible not to displace the node in the exact position indicated, but in the nearest street point. This comes in particularly handy in order to realise the network of static devices: the simulator can be fed with a grid of devices, and it automatically modifies the positions of each node in order to displace it on a street.

The second useful feature is the possibility to rely on the map data to compute routes, as in Figure 8.3(b). ALCHEMIST in fact ships with a module based on GraphHopper [2] which provides the simulator the ability to compute routes between two points of the map. This feature is mainly used to steer nodes correctly along the map, following the allowed ways. It is possible to use such feature also specifying different types of vehicle. Currently, pedestrians, bikes and cars are supported, and can be used together in the same simulation, and even within the same node (for

[2] `http://graphhopper.com/`

(a) GPS trace (b) Map-based navigation (c) Mixed mode

Figure 8.3: Three snapshots showing the different navigation modes available in ALCHEMIST. In Figure 8.3(a) there is a GPS trace consisting in four points. If not specified to behave otherwise, ALCHEMIST computes the average speed between two points, and moves the node along a straight line with constant speed. This, depending on the precision of the trace, may generate paths that cross over buildings. In Figure 8.3(b), the route between the start and end point is computed using the built-in navigator. This could lead the nodes towards paths different from those described by the route. In Figure 8.3(c), the routing subsystem is used to refine the quality of the GPS traces: ALCHEMIST will move the node with constant speed between the two points, using map based routing when necessary to improve precision between two consecutive trace points.

e.g. simulating a pedestrian taking her car, driving and then walking again). Another usage of this system is, for instance, the possibility to use the route distance or expected route time as data items when performing internal calculations, e.g. to compute a spatial gradient where the distance is not the classic euclidean distance but rather the distance computed by the routing subsystem.

The simulator also allows for loading GPS traces. The traces must be in a personalised binary format, fortunately, however, this format is easy to generate. In fact, it is just a Java object stream containing a List of "IGPSPoints", namely a simple structure with latitude, longitude and time. A converter from JSON to such format is also available in the simulator distribution. Just as the previously mentioned routing system, the GPS traces can be used to move nodes along the scenario, reproducing existing paths, as in Figure 8.3(a).

It is also possible to use a combo of the two techniques: often, the GPS traces could be a bit rough with respect to desired grain of the simulation. In these cases, the navigation subsystem can be used to compute the route between two GPS points, with the assumption that the user followed the paths allowed in the map. In this way, it is possible to arbitrarily refine the grain of a GPS trace without the disadvantage deriving from a simpler interpolation, namely the possible transit over physical obstacles. Such mixed navigation mode is depicted in Figure 8.3(c).

Obviously, it would be rather hard to understand what is going on on the simulation without proper rendering support. In this sense, ALCHEMIST automatically detects when a real-world environment is being used, and renders a map relying on MapsForge [3].

[3] `https://code.google.com/p/mapsforge/`

9 Chemical meta-model

9.1 Example scenario: Morphogenesis

As example to test the power of a chemical stochastic simulator enriched a more general model, we propose a case study on morphogenesis. Development of multicellular organisms begins with a single cell – the fertilised egg, or zygote. The egg cell is always asymmetric, *i.e.*, the distribution of maternal proteins inside the cell is not uniform. After fertilisation it divides mitotically to produce all the cells of the body. The resulting blob of cells starts the process of differentiation which is initially caused by the presence of *maternal factors* located in specific areas of the organism's egg. The ordered spatial organisation of this diversity is then caused by the interactions among cells. These can be direct or mediated by specific diffusing proteins called *morphogens*. After that, tissues have been created, and the formation of organs begins so as to originate the final shape of the organism [AJL$^+$02, Gil06].

The main issues of Developmental Biology, nowadays only partially solved, are:

- which are the processes and the genetic mechanisms that control cellular (or nuclear) duplication;
- which are the processes and the genetic mechanisms by which cells differentiate;
- how it is possible that cellular differentiation is spatially organised.

It is clear that the macroscopic emergent results of pattern and shape formation originates from the microscopic mechanisms of intra-cellular reactions and environmental morphogen diffusion. But how these mechanisms coordinate is still under investigation through experimental techniques and theoretical models.

A well known example of multicellular development is given by *Drosophila Melanogaster*, whose spatial organisation results in the segmented pattern of gene expression shown in Figure 9.1. To show the spatial-temporal evolution of the pattern, the organism has been object of several modelling attempts, most of which focus their investigation in gene interactions and protein diffusion mechanisms, bounding the window of analysis at a period of development that does not massively involve nuclear or cellular divisions. To cite few of them, in [RS95] the change of protein concentration is modelled as an Ordinary Differential Equation depending on the process

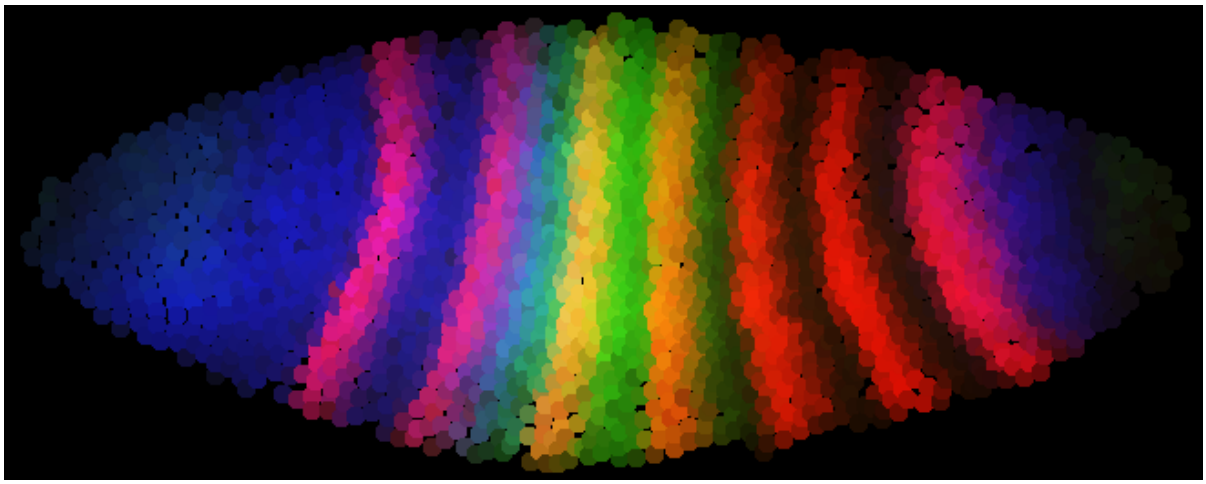


Figure 9.1: The pair-rule gene *even-skipped* (red) together with *hb* (blue) and *Kr* (green) in *Drosophila* embryo at the cleavage cycle 14A temporal class 8. Reconstructed image from [PPSR09]. Embryo name: ba3.

of gene regulation, diffusion over a discretised space and decay. Mitosis are also modelled as a synchronous atomic process that suspends the synthesis of protein and causes a doubled number of cell nuclei. In [GJK$^+$04] it is presented a continuous mathematical model based on a set of coupled non linear reaction-diffusion Partial Differential Equations that describe how protein concentrations change over time and space as a result of the mechanisms of protein synthesis and degradation, gene inhibition and activation, protein diffusion. They replace embryo's cellular structure with a continuum and do not model nuclear divisions. In [MV10] it is described a stochastic and discrete version of [GJK$^+$04] based on a set of interacting compartments inside which chemical reactions implementing the gene regulatory network can occur. It does not count for changes in the network topology due to compartment movement and mitosis. These models run over a simulation window that start from cleavage cycle 11 to cleavage cycle 14. In [LIDP10] the gradient formation of the Drosophila *bicoid* protein, mainly caused by the morphogen diffusion, is simulated with an innovative stochastic model of reaction-diffusion systems implemented into a Gillespie-like stochastic simulation algorithm. They do not model neither gene interactions nor cellular / nuclear divisions.

To capture a bigger window of embryo development we built on top of ALCHEMIST's chemical meta-model a model of *Drosophila* development so as to capture both nuclear division and molecular processes such as morphogen diffusion and gene regulatory networks.

9.1.1 Development of Drosophila Melanogaster

Drosophila is one of the best known multicellular organism. The egg of *Drosophila* is already polarised by differently localised mRNA molecules which are called *maternal effects*. The early nuclear divisions are synchronous and fast (about every 8 minutes): the first nine divisions

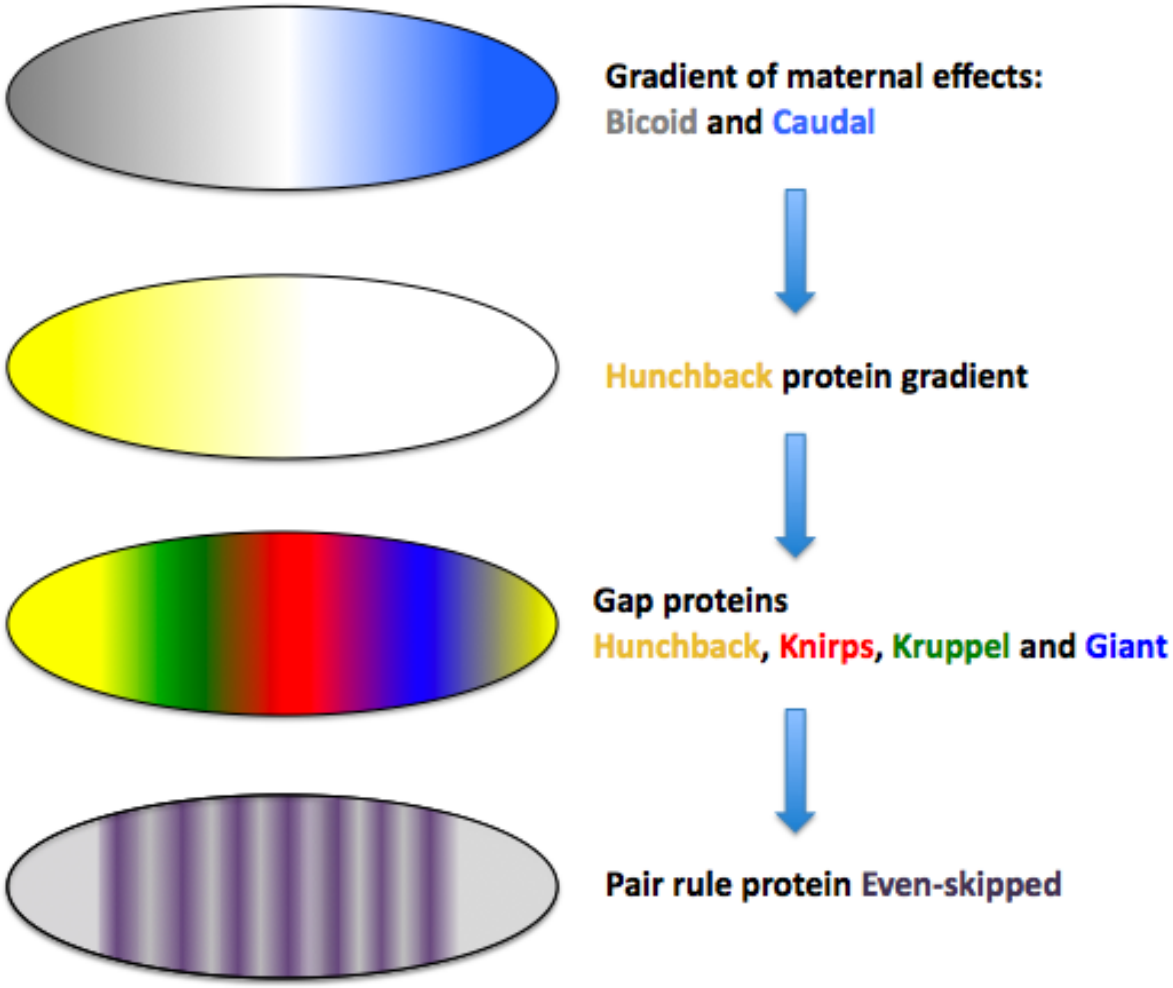


Figure 9.2: Hierarchy of genes establishing the anterior-posterior body plan

generate a set of nuclei forming the *syncytial blastoderm*. All the dividing nuclei share a common cytoplasm, and material can diffuse throughout the embryo. After other four nuclear divisions, during the fourteenth nuclear division, plasma membranes grow to enclose each nucleus, converting the syncytial blastoderm into a *cellular blastoderm* consisting of about 6000 separate cells. After the first ten divisions the time required to complete each of the next four divisions becomes progressively longer: cycle 13 takes for instance 25 minutes. During cycle 14 cells conclude their mitosis in rather different time: some cells take 75 minutes, other 175 minutes to complete this cycle. The rate of division is then constant in the first hours of development (9.05 min^{-1}), then decreases until a low value (0.2 min^{-1}). The transcription of RNA massively begins during cleavage cycle 14 so that the embryo enters in the mid-blastula transition.

The studies on the genetics at the basis of the segmentation in the anterior-posterior body plan

identified a hierarchy of genes that controls segment determination: maternal effects, gap genes, pair-rule genes and segment polarity genes. In Figure 9.2 an illustrative subset of this hierarchy is shown —in the images of this paper we hereafter refer at the anterior pole as their left side, and the posterior as their right side. Maternal effect genes are the building blocks of the anterior-posterior pattern. The most important are *bicoid* (bcd) – forming an anterior-to-posterior gradient – and *caudal* (cad)—forming a posterior-to anterior gradient. The mRNAs of such genes are placed in different regions of the egg and initiate the hierarchy of transcription, driving the expression of *gap genes*, which are the first zygotic genes to be expressed. The first is *hunchback* (hb), that appears early in the embryo development such that sometimes is classified among maternal genes. The basic others are *Krüppel* (Kr), *knirps* (kni) and *giant* (gt). These genes are expressed in specific and partially overlapping domains. Differing combinations and concentrations of the gap gene proteins then regulate the expression of downstream targets, *i.e.*, the *pair-rule genes*, which divide the embryo into a striped pattern of seven segments. The most important pair-rule genes are *even-skipped* (eve) and *fushi-tarazu* (ftz). The pair-rule gene proteins activate the transcription of the *segment polarity genes*, whose protein products specify 14 parasegments that are closely related to the final anatomical segments [AJL$^+$02].

We developed a model that reproduce the process of *Drosophila* development from the fertilised egg until the pattern formation of the *gap genes* during cleavage cycle 14. The whole embryo is modelled as a big cell where nuclei grow and move over a 2D continuous environment filled with diffusing proteins. The mechanisms we explicitly model and we reproduce with simulation are:

- nuclear divisions;
- nuclear migration;
- morphogen diffusion;
- gene interactions.

We model the process of nuclear divisions as a single chemical like reaction, whose precondition is given by the maximum number of other nuclei in the neighbourhood and whose product is a new nucleus. The new nucleus is created close to the dividing one in a casual direction and owns half of its molecular content. The rate of nuclear division is determined according to the rate observed in the real system. Since experimental data show a certain synchronisation among dividing cells, this phenomena has been modelled through a non-Markovian reaction, relying instead on a "drifting" Dirac Comb, namely on a distribution whose events happen every increasing time interval.

Movement of nuclei is based on biomechanical forces of repulsion among neighbouring nuclei. They are in fact constrained to remain within the membrane-delimited area so as to filling pretty homogeneously the available space. If two nuclei are closer than a distance given as a parameter, a new position for them is computed. The nearer two nuclei are, the stronger is the repulsion.

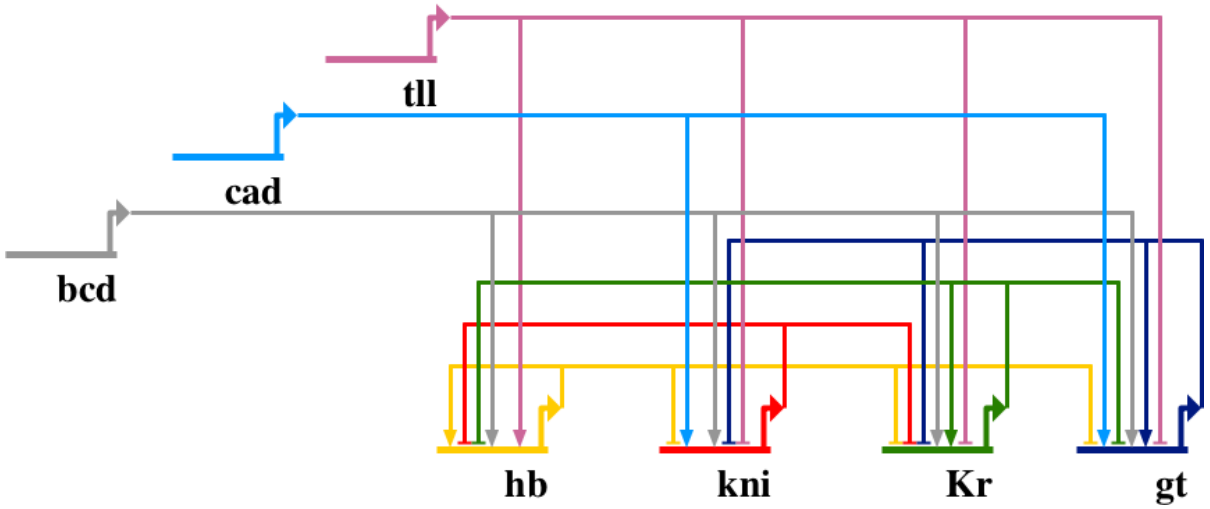


Figure 9.3: Gene regulatory relationships as in the model presented in [PJRG06, RPJ96, GJK+04]. The diagram is realised with BioTapestry [LDB09]. The type of link is pointed out by its shape: links with arrowhead are enhancers, while the repressor links have a foot.

We support diffusion from / to nuclei and inside the environment along the x and y axis. For this purpose environment is discretised into a grid of locations. Molecules move from one nucleus into a neighbour location of the environment picked up probabilistically among the whole neighbourhood, or, the other way round, from one location of the environment into a neighbouring nucleus. Diffusion among locations of the environment follows the same law, implementing a Brownian motion.

In Figure 9.3 it is shown the network of interactions among maternal effectors and gap genes we considered. As in [PJRG06], an other gap gene, *tailess* (tll), also appears as input of the network whose regulation is not clear and we do not represent in the model.

9.1.2 Simulation in ALCHEMIST

Simulations results are shown in Figure 9.4. They are evaluated observing the time evolution of the compartment number and of the gene expression pattern. The time evolution of cells number is compared with data we have from literature and described above. Simulations results for gap genes expression are evaluated according to experimental data available online in the FlyEx database [SKK+08][1] and reported in Figure 9.5. They provide only qualitative information and their evaluation is based on the expression area of the different genes.

The snapshots in Figure 9.4 show one side of the embryo along the anterior-posterior (A-P) axis where almost half of the nuclei are located. The cell number is shown in the label on top of each snapshot. The horizontal axis represents the A-P position and is labelled as % of embryo

[1] `http://urchin.spbcas.ru/flyex/` – last visited on December 2014

length.

The first snapshot shows the initial condition with only one nucleus and the egg polarised by maternal effects localised in the extreme pole: *bcd* on the left and *cad* on the right. During the first minutes of simulation only nuclear divisions occur so as to fill the whole maternal cell at the end of cleavage cycle 9 with around 250 nuclei (half of the total 500), as shown in the second snapshot. Finally in the third snapshot it is reproduced the expression pattern of the four gap genes, whose spatial organisation is compared with experimental data of Figure 9.5. Cells are coloured of yellow, red, blue and green if, in order, they express *hb*, *kni*, *gt* and *Kr*, and their size is proportional to the protein concentration. Gene *hb* is massively expressed in the anterior half of the embryo and a small segment appears in the extreme left: simulations correctly reproduce the main expression domain of *hb* while, even if its expression is observable in the posterior pole, it does not form a clear segment. Gene *kni* is mainly expressed between 60% and 80% of the embryo length, either in real embryo and in simulations, as well as gene *Kr*, which is expressed between 40% and 60% of the A-P axis. The expression of *gt* is finally observable between 10% and 30% and 80% and 90%. The qualitative results presented here are not sufficient for observing the expression in the anterior half of the embryo as soon as it totally overlaps the *hb* expression, while the posterior segment clearly appears.

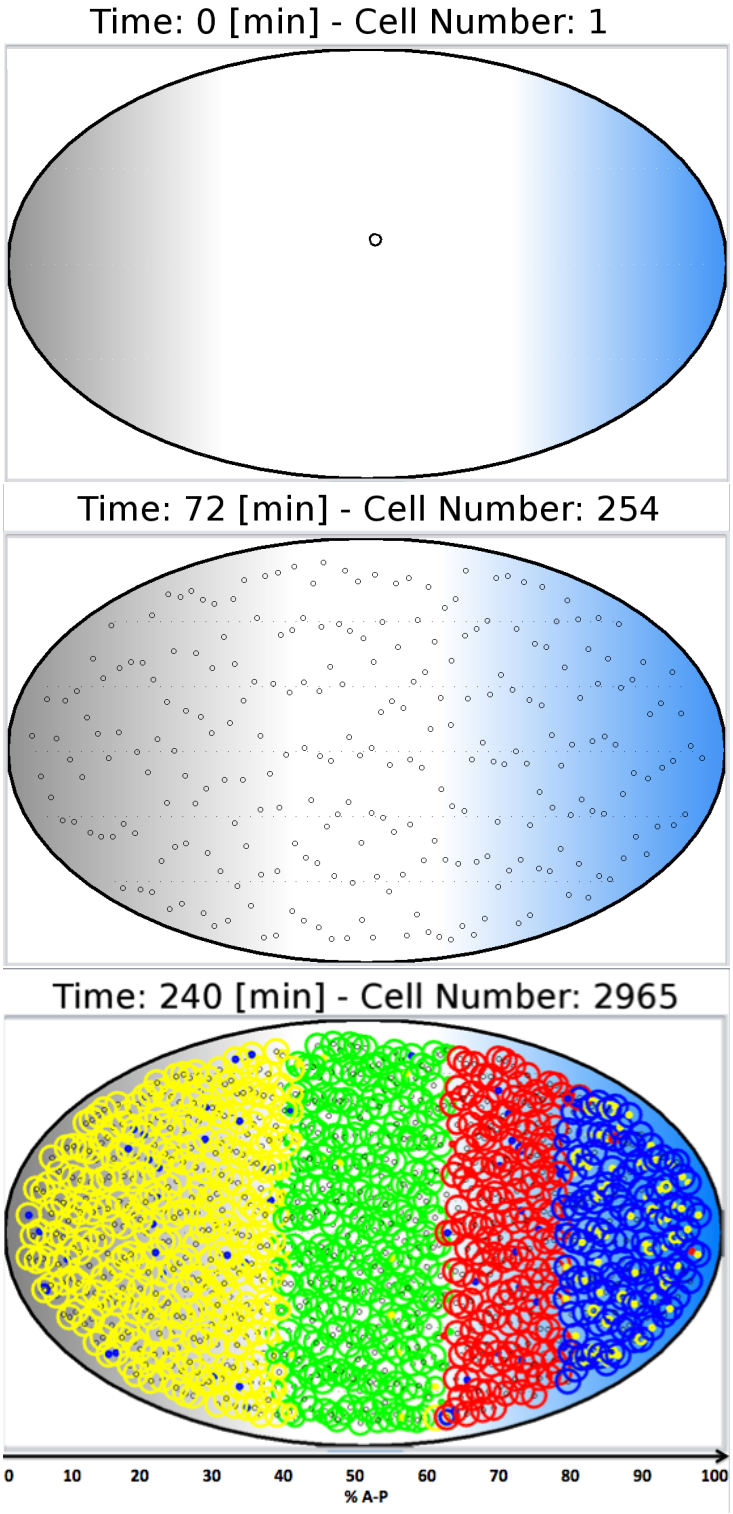


Figure 9.4: Simulation results for the four gap genes *hb* (yellow), *kni* (red), *gt* (blue), *Kr* (green) at a simulation time equivalent to the eighth time step of cleavage cycle 14A

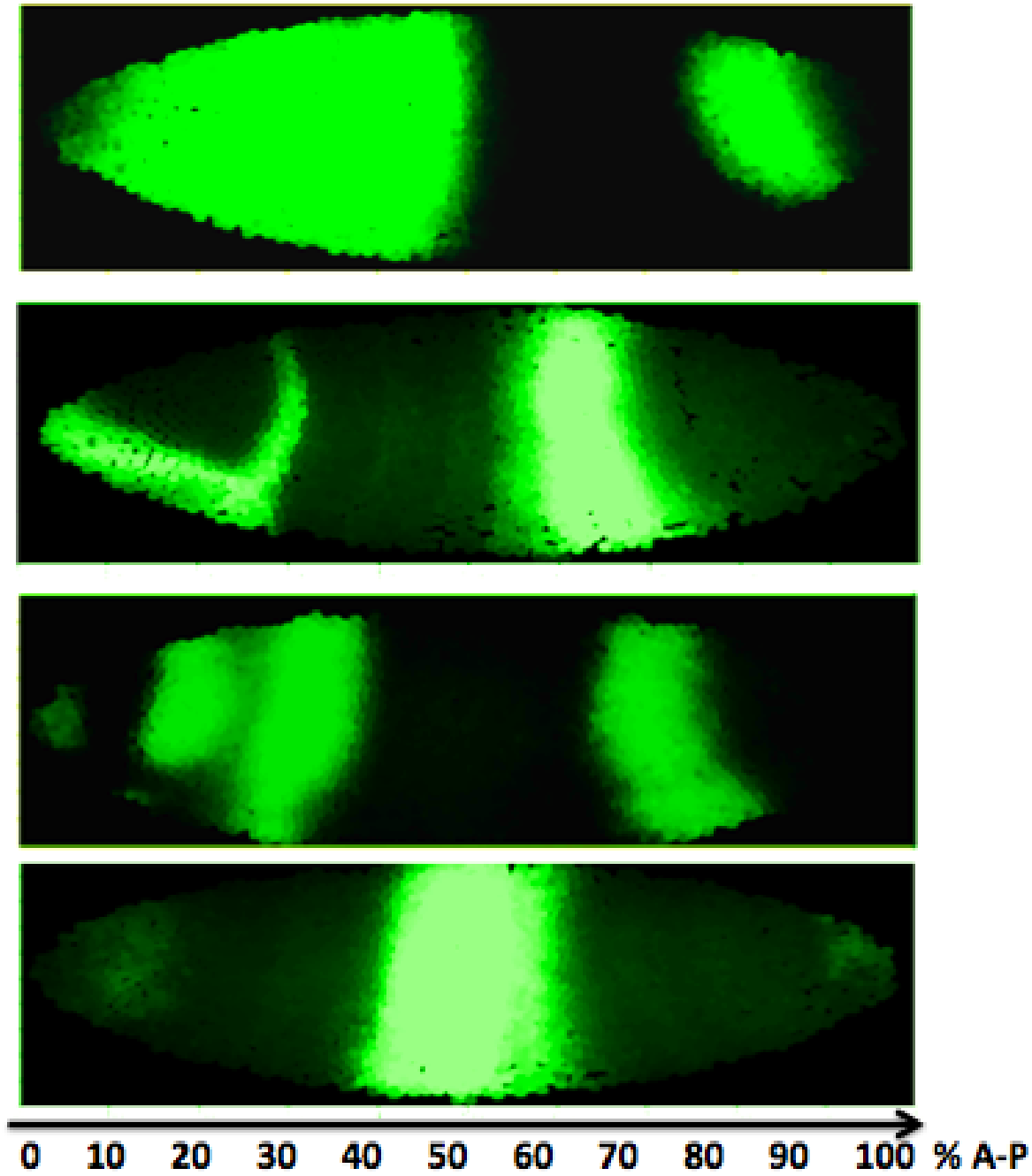


Figure 9.5: The experimental data for the expression of (from the top) *hb*, *kni*, *gt*, *Kr* at the eighth time step of cleavage cycle 14A [SKK$^+$08], ©Maria Samsonova and John Reinitz

10
SAPERE meta-model

Part III

Methods and patterns for self-organisation

11 A process algebra for SAPERE

In this chapter we present a possible formalisation of the SAPERE concepts, in terms of a model whose evolution is specifyied by means of a process algebra. Not only does this formalisation act as a non-ambiguous description, it is also an executable specification and can hence serve as ground for developing simulations of self-organisation mechanisms and for deriving formal proofs of behavioural properties, like self-stabilisation [Vir13]. Few works attempted at a process algebraic formalisation of self-organisation [VPB12, VDB13], developing on top of well-known works on coordination and spatial computing [BGZ98, BB06]. However, the one presented here is – to the best of our knowledge – the first process algebra of self-organising multiagent systems capturing the clear separation of agent behaviour and self-organised interactions, hence effectively expressing the behaviour of large-scale and situated pervasive computing scenarios.

11.1 Model

We now detail a possible coordination model of pervasive ecosystems. It is important to note upfront that the presented coordination model can be instantiated in different ways, depending on the application at hand and on technological aspects: This affects the concrete structure of data, which is an issue orthogonal to the above aspects. A similar abstraction is rather common in the description of space-based coordination models, which often represent tuples as simple atomic elements [BGZ97].

11.1.1 LSAs

LSAs have a unique, system-wide identifier (LSA-id), needed to support a notion of identity that is key both to uniquely identify the agent that injected an LSA and to properly support a bonding mechanism based on reference and not on value/copy. We refer to the content of an LSA as its *description*, which includes all the information the agent wants to manifest to the ecosystem. Several concrete approaches to the structure of an LSA description can be taken. For the sake of exemplification, following the work in [VZSD12, VPMS12, SYD+13] we take a semantic approach based on Resource Description Framework (RDF) representation [MM04], in which an LSA is a set of multi-valued properties, or equivalently, a set of triples (subject, predicate, object)

of an LSA-id, the property name, and the assigned value, expressed as literals (strings) or URIs (terms qualified by universally-accessible namespaces, to which an ontological interpretation can be given [RMCM04]). An example LSA advertising an exhibition in a smartcity, to be diffused around, would be:

```
[ app:type = app:poi, app:event = app:exhibition,
  app:artist = "caravaggio" "michelangelo", ...]
```

URI `app:artist` represents a property name, assigned to the literal values `"caravaggio"` and `"michelangelo"`. Alternatively, a value can be an URI again, like `app:exhibition` that is associated with `app:event`. Namespace `app` is used here to refer to a publicly accessible ontology supporting semantic reasoning on top of those properties and values [SYD+13].

11.1.2 Agent primitives

Standard tuple-based coordination models allow agents to insert, read, and remove any tuple (by primitives `out`, `rd` and `in`) without specific access control constraints—although some models explore their orthogonal adoption, like SecSpaces [GLZ06] and coordination contexts [ORV05, RVO04]. The model we present is profoundly different, mainly due to the chemical metaphor of bonding and the manifestation/observation approach it entails, which guarantees a better modularisation and control of accesses of agent behaviour—see a deeper discussion in Section 6. In particular, the following 4 primitives are provided:

- `new(Description):Id` — This operation relates to creating a new agent manifestation in the system. It takes a description (namely, a property-value association), and correspondingly generates a new LSA and inserts it into the local LSA-space. A new identifier, associated to that LSA, is then automatically created and returned to the agent: it will be used to perform subsequent operations on the LSA. That agent will be called the *owner* of the LSA.

- `update(Id,Update)` — This operation relates to the (continuous) manifestation of agent state. It takes the identifier of an LSA owned by the current agent, and the specification of an update to its current description (typically, property-value re-assignments), and it updates the description of such an LSA. Note that only an LSA owner can modify it.

- `observe(Id,Query):Result` — This operation relates to the observation of an agent's context, namely, of some LSA residing in the same space but possibly belonging to some other agent. It takes the identifier of an owned LSA l and a query, and searches for an LSA bonded to l that provides a valid result to the query. If one is found (nondeterministically), the query result is returned, typically in the form of a set of variable-value associations. The semantics of bonding is orthogonal to that of agent primitives: it is dictated by the structure of a specific bonding eco-law, as described in the following.

- `remove(Id)` — Finally, this operation relates to the stopping manifestation. It takes the identifier of an owned LSA, which will then be removed from the local LSA-space.

11.1.3 Eco-laws

Without any additional mechanism, the behaviour of an agent would be isolated from those of others. Following the ecosystem metaphor, we hence introduce a limited set of eco-laws (playing the role of "the laws of nature"), with the role of evolving the population of tuples over time in a way that supports agent-to-agent interaction both within the same locality, and globally by means of self-organisation. This is achieved in terms of 5 eco-laws, two dealing with local agent-to-agent interaction by the bonding (and debonding) mechanism, and the other 3 (diffusion, aggregation and decay) supporting the mechanisms needed to create self-organising distributed structures of LSAs [FMDMSM$^+$12]. As already mentioned, our description of eco-laws abstracts from the actual structure of LSAs and from the details of pattern matching, which we consider as orthogonal application- and infrastructure-dependent aspects.

- `bonding` — This is the eco-law regulating the creation of bonds. A bond is an asymmetric reference between two LSAs residing in the same space. As already described, the existence of a bond between LSA X and Y means that the owner of X can observe the structure of Y via the `observe` primitive. This eco-law makes sure that a bond between two LSAs exists as long as their description matches according to an infrastructure-specific matching function. In the typical case, a bond is created between an LSA providing a (possibly semantic) query and an LSA positively replying to the query. By denoting with "$\rightsquigarrow Y$" the existence of a bond to Y, this eco-law has the chemical-like structure[1]:

$$X + Y \rightarrow X^{\rightsquigarrow Y} + Y$$

- `debonding` — This eco-law is dual to the previous one, and regulates destruction of a bond between X and Y provided the conditions for the bond are no longer satisfied, namely, when either X or Y are removed or changed. It has the structure:

$$X^{\rightsquigarrow Y} + Y \rightarrow X + Y$$

- `diffusion` — The diffusion eco-law is used to create a clone of an LSA to be shipped to all neighbours of the current node. This eco-law applies to an LSA X that has a particular structure (typically, given property-value associations). The LSA X' accordingly created has a new identifier along with an updated value of some special properties that can keep track of the increased distance from the source (the notion of distance can be extended to consider advanced forms of context-dependency as described in [SYD$^+$13]). Note that

[1] Symbol "+" is used as separator as in chemical laws.

such a new LSA has no owner: it is typically used to create a distributed structure of LSAs representing the global outcome of a single LSA initially created by an agent. By denoting with notation $\overrightarrow{X'}$, an LSA X' to be spread to neighbours, we can express the structure of this eco-law as:

$$X \to X + \overrightarrow{X'}$$

- `aggregation` — As the diffusion eco-law keeps creating copies of an LSA that are shipped to neighbours, the aggregation eco-law has the goal of addressing the implied multiplicity—ensuring that a single version of an LSA coming from the same source is present in a node. In particular, this eco-law takes two compatible LSAs X and Y (typically, two coming from the same "source" LSA after an iterative diffusion process) and aggregates them into a single one X' exploiting a suitable order-independent aggregation function. It has the structure:

$$X + Y \to X'$$

- `decay` — In order to provide a temporal cleanup mechanism, useful to refresh information and the shape of spatial structures, a decay eco-law is also considered, which takes an LSA X and erases it as soon as a given condition on its content holds. Denoting ε as the empty set of LSAs, it has the structure:

$$X \to \varepsilon$$

11.2 Formalisation

In this section we provide a formalisation of the proposed coordination model as informally presented in previous section. We adopt the style that has now became a standard after a series of previous works—starting from the work of Zavattaro et.al. [BGZ97], as surveyed in [VO06], and more recently including, e.g., [VPB12, TNN12, BLM13, ML12, LBF13]. Namely, this is presented in the form of a process algebra, in which a system state is conceived as a "soup" (or networked set of soups) where agents and LSAs float: subjective coordination is realised by agents performing coordination primitives amounting to interactions with the local space (producing/consuming/modifying LSAs), while objective coordination is realised by eco-laws executing as rules continuously manipulating the space on top [VC09, OZ99, ROD02]. This approach has been shown to elegantly express the semantics of the various constructs in an orthogonal way, that is, one rule of operational semantics per coordination primitive (and this will indeed be the case for our formalisation as well). Additionally, using process algebras paves the way for advanced analysis similarly to what developed in related approaches, including

σ	Node identifier
n, m	LSA identifier
x	Identifier variable
d, e	LSA description
Δ	Update specification
$\overline{v}$	A meta-variable over set of elements $\{v_1, \ldots, v_n\}$

Figure 11.1: Meta-variables

mathematical proofs of behavioural properties [Vir13], simulation [VPMS12, DLM05], and observational equivalence [LBF13].

11.2.1 Syntax

We let meta-variable σ range over node (or device) identifiers, n, m over LSA identifiers, x over variables to be eventually substituted by those identifiers, d, e over LSA descriptions, and Δ over update specifications. Given any of those meta-variables, we use the overline notation to indicate a meta-variable over sets of elements (e.g., $\overline{n}$ is a meta-variable over sets of LSA identifiers)—and similarly for the others. Figure 11.1 recalls these meta-variables in tabular form for the reader's convenience. Notation $|\overline{n}|$ is used to extract the number of elements in $\overline{n}$.

The syntax of the calculus is reported in Figure 11.2. A system configuration C is a multiset composition (by operator $|$) of spaces of the form $[S]_\sigma$ (S is the content and σ the identifier), and neighbourhood relationships $neigh(\sigma, \overline{\sigma})$, meaning σ has the neighbourhood $\overline{\sigma}$. A space (content) S is itself a multiset composition of agents $(\overline{n})A$ (with behaviour A and owning LSAs with identifiers $\overline{n}$), LSAs $n\langle d \rangle$ (with identifier n and a description d), LSAs to be spread to neighbours $\star\langle d \rangle$ and bonds $n \rightsquigarrow m$ (from n to m). Agent behaviour A can be of four kinds:

1. 0 is the empty agent;
2. $a;A$ is the agent executing action a and then the continuation A;
3. $A : B$ is a try-catch construct (execute A, and when/if a failure occurs, B is executed);
4. $\texttt{iterate}^B\{A\}$ is iterative (infinite) execution of A preceded by execution of current iteration B.

Note that in the iteration construct the appendix B is not present in the surface syntax: it is initially set to 0 and will dynamically keep track of the state of current iteration.

Actions correspond to the 4 coordination primitives described in previous section: $x :=\texttt{new}(\Delta)$ creates a new LSA with description defined by specification Δ, and causing substitutiion of x with the LSA identifier; $t\,.\,\texttt{upd}(\Delta)$ updates the LSA with term identifier t by specification Δ;

$$
\begin{array}{rcll}
C & ::= & 0 \mid [S]_{\sigma} \mid neigh(\sigma,\overline{\sigma}) \mid (C \mid C) & \text{Configuration} \\
S & ::= & 0 \mid (\overline{n})A \mid n\langle d\rangle \mid \star\langle d\rangle \mid n \rightsquigarrow m \mid (S \mid S) & \text{Space} \\
A,B & ::= & 0 \mid a;A \mid A:B \mid \texttt{iterate}^{B}\{A\} & \text{Agent} \\
a & ::= & x\texttt{:=new}(\Delta) \mid t\texttt{.upd}(\Delta) \mid t\texttt{.obs}(\Delta) \mid t\texttt{.rem()} & \text{Action} \\
t & ::= & n \mid x & \text{Term}
\end{array}
$$

Figure 11.2: Syntax

$$
\begin{array}{ccc}
C \mid 0 \equiv C & C \mid C' \equiv C' \mid C & (C \mid C') \mid C'' \equiv C \mid (C' \mid C'') \\
S \mid 0 \equiv S & S \mid S' \equiv S' \mid S & (S \mid S') \mid S'' \equiv S \mid (S' \mid S'') \\
0 : A \equiv 0 & \texttt{iterate}^{A}\{0\} \equiv 0 & \texttt{iterate}^{0}\{A\} \equiv \texttt{iterate}^{A}\{A\}
\end{array}
$$

Figure 11.3: Congruence table of a SAPERE algebra

$t\texttt{.obs}(\Delta)$ observes an LSA bond to t using Δ as a query; and finally $t\texttt{.rem()}$ removes LSA with identifier t. In particular note that as soon as they are executed, the latter three constructs should have term identifier t be bound to an actual identifier n.

A congruence relation is also introduced in Figure 11.3, which equates terms that have to be considered syntactically equivalent. The first two lines state that parallel composition operator is actually a multiset one; the third line defines properties of try-catch (if the left-hand side is over, the whole agent process is over) and iteration (iterating the empty process gives the empty process, and when current iteration is over we spawn a new one).

11.2.2 Semantics

The semantics of the calculus is parametric in the functions shown in Figure 11.4, which are to be concretised to make the model fully executable—as we will develop in next section. The

$$
\begin{array}{rl}
\Delta \rhd d = d' & \text{Description filtering function} \\
\delta(d) = d' & \text{Diffusion function} \\
\alpha(d,e) = d' & \text{Aggregation function} \\
\varepsilon(d) & \text{Decay predicate} \\
\mu(d,e) & \text{Bond matching predicate}
\end{array}
$$

Figure 11.4: Abstracted functions

$$
\begin{array}{rrcll}
\texttt{[CGR-C]} & C_0 & \to & C_1 & \text{if } C_0' \equiv C_0,\ C_0' \to C_1',\ C_1' \equiv C_1 \\
\texttt{[NEST]} & C \mid [S]_\sigma & \to & C \mid [S']_\sigma & \text{if } S \to S' \\
\texttt{[SHIP]} & C \mid [\star\langle d\rangle]_\sigma & \to & C \mid \prod_i [n_i\langle d\rangle]_{\sigma_i} & \text{if } neigh(\sigma,\overline{\sigma}) \in S,\ |\overline{\sigma}| = |\overline{n}|,\ fresh(\overline{n})
\end{array}
$$

Figure 11.5: Rules modelling architectural aspects

$$
\begin{array}{rrcll}
\texttt{[CGR]} & S_0 & \to & S_1 & \text{if } S_0' \equiv S_0,\ S_0' \to S_1',\ S_1' \equiv S_1 \\
\texttt{[TRY]} & S \mid (\overline{n})(A:B) & \to & S' \mid (\overline{n}')(A':B) & \text{if } S \mid (\overline{n})A \to S' \mid (\overline{n}')A' \\
\texttt{[CATCH]} & S \mid (\overline{n})(A:B) & \to & S \mid (\overline{n})B & \text{if } S \mid (\overline{n})A \not\to,\ A \not\equiv 0 \\
\texttt{[ITER]} & S \mid (\overline{n})\texttt{iterate}^{B}\{A\} & \to & S' \mid (\overline{n}')\texttt{iterate}^{B'}\{A\} & \text{if } S \mid (\overline{n})A \to S' \mid (\overline{n}')B' \\
\texttt{[NEW]} & S \mid (\overline{n})x\texttt{:=new}(\Delta);A & \to & S \mid m\langle \Delta \rhd 0\rangle \mid (m,\overline{n})A^{\{m/x\}} & \text{if } fresh(m) \\
\texttt{[UPD]} & S \mid n\langle d\rangle \mid (n,\overline{n})n\texttt{.upd}(\Delta);A & \to & S \mid n\langle \Delta \rhd d\rangle \mid (n,\overline{n})A & \\
\texttt{[OBS]} & S \mid m\langle d\rangle \mid (n,\overline{n})n\texttt{.obs}(\Delta);A & \to & S \mid m\langle d\rangle \mid (n,\overline{n})A^{matcher(\Delta,d)} & \text{if } n \rightsquigarrow m \in S \\
\texttt{[REM]} & S \mid n\langle d\rangle \mid (n,\overline{n})n\texttt{.rem}();A & \to & S \mid (\overline{n})A &
\end{array}
$$

Figure 11.6: Rules modelling coordination primitives

$$
\begin{array}{rrcll}
\texttt{[DIFF]} & S \mid n\langle d\rangle & \to & S \mid n\langle d\rangle \mid \star\langle \delta(d)\rangle & \\
\texttt{[AGR]} & S \mid n\langle d\rangle \mid m\langle e\rangle & \to & S \mid n\langle \alpha(d,e)\rangle & \text{if } (m,\overline{n})A \notin S \\
\texttt{[DEC]} & S \mid n\langle d\rangle & \to & S & \text{if } \varepsilon(d),\ (n,\overline{n})A \notin S \\
\texttt{[BND]} & S \mid n\langle d\rangle \mid m\langle e\rangle & \to & S \mid n\langle d\rangle \mid m\langle e\rangle \mid n \rightsquigarrow m & \text{if } \mu(d,e),\ n \rightsquigarrow m \notin S \\
\texttt{[DBND]} & S \mid n \rightsquigarrow m & \to & S & \text{if } S \not\to S \mid n \rightsquigarrow m
\end{array}
$$

Figure 11.7: Rules modelling eco-laws

filtering function gives semantics to updates and queries over LSA descriptions as follows. First of all, given an update specification Δ and an LSA description d, then $\Delta \rhd d$ gives the description obtained by applying Δ to d—this function is assumed to be total. Second, Δ can be seen as a query: we say it matches a description d with result θ (a binding of variables) if $\Delta\theta \rhd d = d$, namely, when $\Delta\theta$ is seen as an update, it would not affect d—intuitively, e.g., an LSA of type a, matches with a query asking whether its type is X (and gives reply a), if updating the type of that LSA to a leaves it unchanged. As a mere example in this section assume Δ and d are both lists of single-value assignments $p = v$, and that filtering $\Delta \rhd d$ applies all assignments of Δ to d, such that $(p = 1, q = 2) \rhd (q = 3, r = 4) = (p = 1, q = 2, r = 4)$. Observe that update $(q = 3, r = X)$, where X is a variable, can be seen as a query which applied to $(q = 3, r = 4)$ yields $\theta = \{X/4\}$, in that $(q = 3, r = X)\{X/4\} \rhd (q = 3, r = 4) = (q = 3, r = 4)$ as described above.

The other four functions in Figure 11.4 work with LSA descriptions, and define the data-related aspects of eco-laws as described in the following.

The operational semantics is defined as a transition system with judgement $C \to C'$, meaning that system configuration C can move to C' by a transition. The operational semantics is split in three parts: general congruence rules (Figure 11.5), rules defining agent behaviour (Figure 11.6), and finally rules defining eco-laws (Figure 11.7), described in turn.

In Figure 11.5, rule [CGR-C] is the standard rule making congruent configurations be considered equivalent, rule [NEST] allows one to recursively enter a space, and finally rule [SHIP] defines the broadcast semantics: when an LSA-to-be-spread $\star\langle d \rangle$ occurs in a space, a copy of it is sent to all neighbours (by action $\texttt{ship}(\overline{n})$, where $\overline{n}$ is a set of externally provided freshly-generated identifiers)—note that we abuse the notation as in [IPW01], writing $\overline{n}$ for $n_1, \ldots, n_k$ and similarly for $\overline{\sigma}$.

Figure 11.6 provides the rules defining the internal behaviour of spaces, affecting agent behaviour. Rule [CGR] states congruence inside spaces similarly to what [CGR-C] does for system configurations. Rule [TRY] handles successful operations inside a try-catch construct: simply, in that case the left-hand side of operator ":" is allowed to proceed. Rule [CATCH] conversely handles a failure: when the left-hand side of operator ":" gets stuck (and is not empty), we simply allow the part on right (the handler) to carry on. Rule [ITER] handles the semantics of iteration: it simply allows the appendix to carry on—when it becomes 0 a new iteration will be spawn by the congruence relation as discussed above.

The subsequent 4 rules handle the 4 coordination primitives of the model. Rule [NEW] create a new LSA with fresh identifier n: such an LSA gets created with description $\Delta \rhd 0$ (namely, as specified in the argument of operation `new`), n is added to the set of LSAs owned by the agent, and the continuation is allowed to carry on after applying substitution of m to x. Rule [UPD] handles update operations: the LSA with identifier n should be in the space – let d be its description – which is moved to $\Delta \rhd d$ and the agent continuation is allowed to carry on. Rule [OBS] handles an observation operation of query Δ: it looks for a bond to m and accesses its description d, it then computes via function $matcher(\Delta, d)$ a minimal substitution θ such that $d = \Delta\theta \rhd d$, and if one exists it is applied to the continuation—if no such LSA is found instead, the agent gets stuck, and this failure could be caught as seen above. Rule [REM] models removal

of an LSA, which also causes removal from the set of identifiers of LSAs owned by the agent.

Finally, Figure 11.7 defines the semantics of the 5 eco-laws, which are parametric in functions δ, α, ε, and μ. The first is the diffusion eco-law: which takes an LSA whose description is in the domain of δ, and creates an LSA-to-be spread whose description is the output of δ. The aggregation eco-law takes two LSAs and applies function α to their description: the result is stored in one of the two, while the other is removed (provided it is not an owned LSA). The decay eco-law simply drops an LSA whose description makes predicate ε hold (again, provided it is not an owned LSA). Note that the side-conditions of the aggregation and decay eco-laws prevent an LSA's removal if an agent owns it. The bond eco-law creates a bond between two LSAs if they stay in the μ predicate relation (and if one such bond does not yet exist). Conversely, the debond eco-law drops any bond that the bond eco-law would not create.

11.2.3 Example

Whereas exemplifying eco-laws will better be done in next section while discussing the case study, it is here useful to provide a simple agent program to show how the operational semantics works. Assume the following agent process A_0 is, executed into an initially empty LSA-space:

$$x{:=}\texttt{new}(q=2);\texttt{iterate}^0\{x\texttt{.obs}(e=\texttt{true});x\texttt{.upd}(q=0):x\texttt{.upd}(q=1)\}$$

By rule [NEW] a new LSA is created, assume it has id n, and substitution $\{n/x\}$ is hence propagated in the continuation leading to:

$$(n)\texttt{iterate}^0\{n\texttt{.obs}(e=\texttt{true});n\texttt{.upd}(q=0):n\texttt{.upd}(q=1)\} \mid n\langle q=2\rangle$$

Let A be the specification inside iteration, because of congruence we have that current state is equivalento to:

$$(n)\texttt{iterate}^{n\texttt{.obs}(e=\texttt{true});n\texttt{.upd}(q=0):n\texttt{.upd}(q=1)}\{A\} \mid n\langle q=2\rangle$$

Now, the appendix is allowed to carry on by rule [ITER]; however since there's currently no bond from LSA n, observation is stuck, hence because of rule [CATCH] we actually move in one step to

$$(n)\texttt{iterate}^{n\texttt{.upd}(q=1)}\{A\} \mid n\langle q=2\rangle$$

and after execution of update by rule [UPD] to $(n)\texttt{iterate}^0\{A\} \mid n\langle q=1\rangle$ which is equivalent to:

$$(n)\texttt{iterate}^A\{A\} \mid n\langle q=1\rangle$$

Without bonds, observation keeps fail and the space of LSAs will not change. If instead at some point a new LSA $m\langle e=\texttt{true}\rangle$ is injected in this space, and a bond to it is created by the bonding eco-law, then observation succeeds, so at some point we would move to state

$$(n)\texttt{iterate}^{n\texttt{.upd}(q=0):n\texttt{.upd}(q=1)}\{A\} \mid n\langle q=2\rangle \mid m\langle e=\texttt{true}\rangle$$

and then after update to:

$$(n)\texttt{iterate}^A\{A\} \mid n\langle q=0\rangle \mid m\langle e=\texttt{true}\rangle$$

11.2.4 Well-formed configurations and properties

A configuration is considered well-formed if:

1. no two LSA-spaces have the same identifier;
2. for each space with identifier σ there is precisely one item $neigh(\sigma, \overline{\sigma})$ and all elements in $\overline{\sigma}$ have a corresponding space;
3. no two LSAs share the same LSA identifier;
4. iterations do not include removal of LSAs (`rem`);
5. for any agent $(\overline{n})A$ we have $\overline{n} \vdash A$.

The latter is a well-formedness judgment for agents, checking that variables over identifiers are properly managed and that no agent performs operations on LSAs it does not own. This judgment is inductively defined as:

$$\begin{array}{rl}
\bar{t} \vdash x\texttt{:=new}(\Delta);A & \text{if } x \notin \bar{t},\ (x,\bar{t}) \vdash A \\
(t,\bar{t}) \vdash t\texttt{.upd}(\Delta);A & \text{if } (t,\bar{t}) \vdash A \\
(t,\bar{t}) \vdash t\texttt{.obs}(\Delta);A & \text{if } (t,\bar{t}) \vdash A \\
(t,\bar{t}) \vdash t\texttt{.rem()};A & \text{if } \bar{t} \vdash A \\
\bar{t} \vdash (A:B) & \text{if } \bar{t} \vdash A,B \\
\bar{t} \vdash \texttt{iterate}^{A}\{B\} & \text{if } \bar{t} \vdash A,B
\end{array}$$

We now state two fundamental properties for the proposed calculus:

- **Subject reduction** — If a configuration C is well-formed, and $C \xrightarrow{l} C'$, then C' is well-formed. This property is key to guarantee that starting from a well-formed configuration, we never reach badly structured configurations.
- **Progress** — If a well-formed configuration $C \mid [(\overline{n})A]_\sigma$ allows no transition leading to $C' \mid [(\overline{n}')A']_\sigma$ (namely, the agent is stuck), then necessarily $A \equiv (n\texttt{.obs}(\Delta);A')$, and in σ there is no bond from n to m such that m's description d satisfies $\Delta\theta \rhd d = d$. This property states that in well-formed configurations operations never fail, except for the case of an `obs` that fails and has not been caught by the try-catch construct.

12
Pattern: Channel

We now present an example application of the coordination model proposed in Chapter 11, specifically targeted at emphasising its ability to tackle a number of interesting issues:

1. enact robust self-organising design patterns [FMDMSM$^+$12];
2. code relevant classes of agents (contextualizers, combinators of spatial structures, initiators of aggregation/diffusion processes);
3. support openness by an RDF-oriented instantiation of LSAs.

12.1 A semantic-oriented instantiation

We give an RDF-oriented concrete definition of data representation, which can be useful in a large number of pervasive computing applications, including the example to come—a rigorous mapping to RDF and SPARQL can be given along the lines of [VZSD12]. Namely, we provide:

- the shape of LSA descriptions (metavariable d);
- the space of update specifications (metavariable Δ);
- the description filtering function ($\Delta \rhd d$);
- the definition of eco-law-related functions $\delta, \alpha, \varepsilon, \mu$.

For the sake of conciseness, we describe them informally.

An LSA description d is a comma-separated sequence of multi-value assignments of the kind $\mathtt{p}{=}\overline{\mathtt{v}}$, where *property* `p` is a URI, $\overline{\mathtt{v}}$ is a sequence of values, and a single value `v` is either a URI, a string or a variable (written starting with a question mark as in `?Var` as in RDF/SPARQL). An update specification Δ can be a variable, or the literal `empty` (meaning the LSA is entirely empty), or a comma-separated sequence of assignments $\mathtt{p}{=}\overline{\mathtt{w}}$, where each element $\mathtt{w}_i$, called a compound value, can be a value or an expression to be evaluated (written `eval(exp)` where `exp` is a string).

Filtering function defines the update behaviour as follows. First, to compute $\Delta \triangleright d$ all expressions in Δ are evaluated, leading to filtering function d'. Second, if Δ' is literally `empty` then the result of filtering function is simply an empty description. If it is instead a list of assignments, the filtering function applies all of them to d, and returns the resulting description d'—an assignment `p=`$\overline{\mathtt{w}}$ completely rewrites any possible pre-existing one. As an example, if Δ is `[app:p = "1", app:q = "2"]` and d is `[app:p = "3" "4", app:r = "5"]`, then $\Delta \triangleright d$ gives `[app:p = "1", app:q = "2", app:r = "5"]`.

We finally need to give the definition of eco-law functions, as of Figure 11.4. First, predicate ε holds only for LSA descriptions having property `eco:decay` set to `"true"`, which are hence the descriptions of those LSAs that will be decayed.

Function δ takes the description of an LSA l with the assignments `eco:diff_function` $= f$ and `eco:diff_prop` $= p_1 \dots p_n$, where each p_i is assumed to be a property of l assigned to a single-value v_i. The output is a description obtained by the input one with two changes: *(i)* properties $p_1, \dots p_n$ are no longer assigned to their old values $v_1, \dots, v_n$, but to the result of $f(v_1, \dots, v_n)$ which is assumed to be a tuple of n values, and *(ii)* property `eco:decay` is set to `"true"` so that as soon as the LSA is diffused and processed remotely, it will then be decayed. Note that function f needs not be total, hence the diffusion process can be defined so as to eventually terminate. For instance, an LSA with description

```
[app:p = "1", app:q = "10", eco:diff_prop = app:p app:q,
 eco:diff_function = fun:inc fun:dec]
```

will be diffused and remotely become

```
[app:p = "2", app:q = "9", eco:diff_prop = app:p app:q,
 eco:diff_function = fun:inc fun:dec, eco:decay = "true"]
```

Aggregation is achieved by binary function α defined as follows. It applies to a pair of descriptions for two LSAs l_1 and l_2 having both identical copies of the three assignments: `eco:aggr_function` $= f$, `eco:aggr_prop` $= \overline{p}$ and `eco:aggr_pre` $= \overline{q}$: again, properties $\overline{p}$ and $\overline{q}$ are assumed to be single-valued. Then, function f is applied to the values assigned to $\overline{p}$ in l_1 and l_2, say they are $\overline{v}$ and $\overline{v}'$) respectively, and the result $f(\overline{v}, \overline{v}')$ (a tuple of values) will be used to replace in l_1 the assignments to $\overline{p}$ while l_2 is removed. As a precondition for the aggregation to happen, l_1 and l_2 should have the same assignment for properties $\overline{q}$. For instance, the two LSAs

```
[app:p = "1", app:q = "10", eco:aggr_pre = app:q,
 eco:aggr_prop = app:p, eco:aggr_function = fun:sum]
[app:p = "2", app:q = "10", eco:aggr_pre = app:q,
 eco:aggr_prop = app:p, eco:aggr_function = fun:sum]
```

will be aggregated and become

```
[app:p = "3", app:q = "10", eco:aggr_pre = app:q,
 eco:aggr_prop = app:p, eco:aggr_function = fun:sum]
```

Finally, function μ realising bonding applies to an LSA l_1 with assignments `eco:bond_prop` $= p_1 \dots p_n$ and `eco:bond_prop` $= v_1 \dots v_n$, and to another LSA l_2 with properties $p_1, \dots, p_n$ respectively assigned to $v_1, \dots v_n$. Accordingly, eco-law [BOND] will create a bond from l_1 to l_2, that will persist until this matching holds. For instance, the LSA

```
[eco:bond_prop = app:p app:q, eco:bond_values = "2" ?Any]
```

will bond to an LSA with description

```
[app:p = "2", app:q = "10"]
```

The above examples illustrate a convention we shall use in the following: URIs with namespace `eco` are those describing properties and values whose semantics is associated to the eco-laws engine, those with `fun` refer to a library of underlying functions, and those with `app` are relative to the application domain ontology.

12.2 Realising the gradient

Assume an adaptive display infrastructure, where a huge number of displays are deployed more or less uniformly in a wide and possibly articulated area, like a smartcity [MVFM+13]. In a given location, called the *source*, a point-of-interest (POI) is deployed: several pedestrians around may be interested in reaching that POI according to a convenient path. A steering service with the goal of guiding those people to the POI can be setup by making each display provide a dynamic sign of the direction to take, automatically computed by the opportunistic local interactions between the deployed devices. As described in previous works [MVFM+13, BBVT08, VCMZ11], this service can be provided by the so-called *gradient* self-organisation pattern [FMDMSM+12]: from the POI and by suitable diffusion and aggregation processes, a distributed data-structure is established which reifies in each node the distance (e.g., hop-by-hop) from the nearest source node. By making each display show a sign pointing towards the neighbour node with smaller distance value, one obtains the desired steering service. A solution to this problem can be obtained with the three agents described in Figure 12.1.

First of all, a *source* agent in each node located at the POI locally injects an LSA declaring the node as being a source for the POI gradient, with all additional domain descriptions of the POI.

```
% Source: injecting the source LSA at a POI
?SRC := new(app:src = true, app:type = app:grad, app:desc = ..)

% GradientAgent: from the source, prepares the diff/aggr process
?BND_LOC := new(eco:bond_prop = eco:location, eco:bond_value = "true")
?BND_SRC := new(eco:bond_prop = app:src, eco:bond_value = "true")
?GRAD := new();
iterate {
  ?BND_LOC.obs(eco:location = ?LOC);    % observes current location
  ?BND_SRC.obs(?SRC);           % observes whole source description
  ?GRAD.upd(?SRC);       % clones SRC description into the grad LSA
  ?GRAD.upd(app:src = "false",       % adds all required properties
    app:distance = "0", app:temp = "false", app:location = ?LOC,
    eco:aggr_pre = app:type app:temp,
    eco:aggr_prop = app:distance,
    eco:aggr_function = fun:minOnFirst,
    eco:diff_prop = app:distance app:temp app:location,
    eco:diff_function = fun:inc fun:falseToTrue ?LOC)
  :
  ?GRAD.upd(empty)    % if above observations fail, empties the LSA
}

% Contextualizer: if not blocked by a proper LSA, reifies the gradient
?BND_TEMP := new(eco:bond_prop = app:temp, eco:bond_value = true);
?BND_OBS := new(eco:bond_prop = app:block, eco:bond_value = ?B);
?GRAD := new();              % creates the LSA for the gradient LSA
iterate {
  ?BND_TEMP.obs(?TEMP);             % observe temporaneous grad LSA
  (                    % either consolidate or empties the grad LSA
   ?BND_OBS.obs(app:block = true); ?GRAD.upd(empty)
   :
   ?GRAD.upd(?TEMP); ?GRAD.upd(app:temp = false)
  )
  :
  ?GRAD.upd(empty)
}
```

Figure 12.1: Multi-agent system for creating a distributed gradient data structure

Then, a *gradient* agent, deployed in each node of the network (or at least on source nodes), has the goal of observing the presence of a source LSA, and accordingly start the diffusion/aggregation process ending up with a gradient data structure. By insertion of LSAs `?BND_SRC` and `?BND_LOC`, the agent seeks to bond with a source LSA and a location LSA—namely, an LSA which we assum some location agent creates and maintains to reify current node's location. It also creates an empty LSA that will hold the gradient value at this position- It then iteratively reads current location (`?LOC`) and whole content of source LSA (`?SRC`), and correspondingly updates the gradient LSA: it copies there all assignments in the source LSA, then sets distance to 0, "temporaneous" flag to `false`, location to `?LOC`, and suitable aggregation and diffusion functions. Such functions make the LSA diffuse by increasing distance by 1 at each step, updating flag from `false` to `true`, keeping the current location upon diffusion, and making multiple copies of this LSA coming from different nodes re-aggregate, always taking the one with minimum distance[1].

Finally, a *contextualiser* agent is in charge of enacting the situation recognition process by which it may be decided that the gradient has not to be locally propagated: if a blocking LSA is present (one with assignment of `app:block` to `true`), the gradient will not be propagated further here—modelling e.g. a dynamically moving obstacle to be circumvented. After suitable requests for bonding as in the previous case, and creation of an empty LSA called `?GRAD`, it iteratively checks whether there is a temporaneous gradient LSA and if there is no "blocking" LSA: if both conditions hold, it simply copies the content of gradient LSA into `?GRAD` and moves the flag to `false` (so that diffusion/aggregation will carry on), otherwise it empties `?GRAD` entirely.

The gradient structure created by this process enjoys self-stabilisation as defined in [VD14], namely, it repairs in response to *any* change in the environment, including topological changes and re-positioning of POI.

12.3 From gradient to distributed channel

On top of a gradient structure one can set up a steering service that activates all the signs located along an optimal path from an area *S*, where interested people are initially located, to the POI *D*. Such a path should also have a given "width" to take into account the variability of people movement. The resulting spatial structure can be referred to as a (distributed) *channel*, and should be created by self-organisation in a way that it can, again, automatically adapt to the presence of obstacles and to changes in the environment—see a preview in Figure 12.2.

A solution to this problem can be conceived following the general approach of "spatial computing" [BDU+13, Vir13, BB06], namely, functionally composing simpler gradient-based data structures. It can be coded in our process algebraic language as shown in Figure 12.3, with the behaviour of the following 3 kinds of agent.

[1] Function `fun:minOnFirst` yields the first argument if smaller than the second, while function `fun:falseToTrue` applies only if the flag is `false` and turns it to `true`

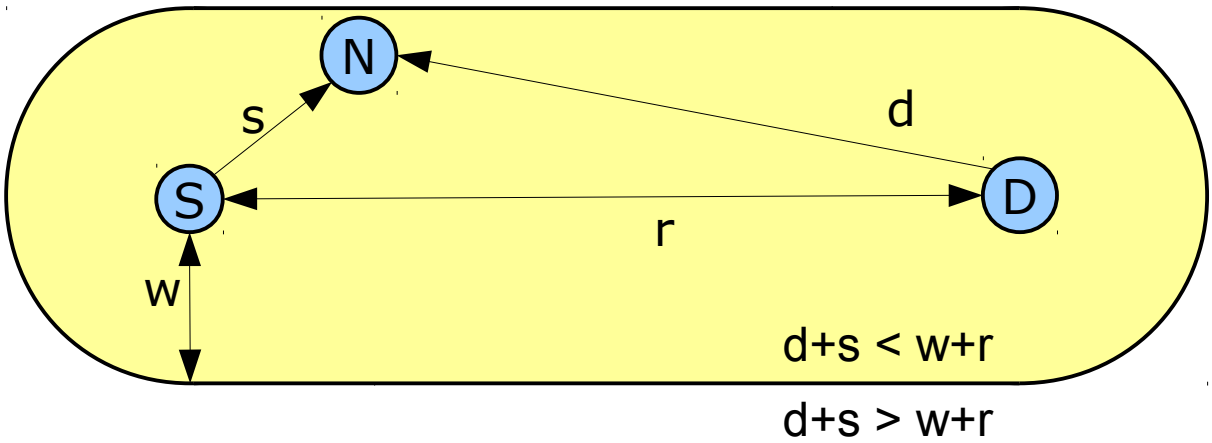


Figure 12.2: Construction of a distributed channel, based on distance between the two ends S and D (r), width (w), distance from S (s) and distance from D (d).

First a *channel source* agent located at position S is in charge of bonding with the gradient spread from position D, and accordingly creates a source LSA `?SRC` which will be continuously updated to reflect the current distance `?R` from S and D, and to carry also the width value `?W`. The gradient agent defined in Figure 12.1 will be able to create a new gradient out of this source LSA, which will propagate carrying both `?R` and `?W`, in addition to the distance from D associated with the `app:distance` property.

Then, a *channel service* agent is able to finally create the channel LSA: it bonds and observes both gradients, and the sets a flag into a newly created (and continuously updated) LSA depending on whether the node is inside or outside the channel area, as described in Figure 12.2.

Finally, a *steering service* agent has the goal of reifying a sign towards the direction of area D only in nodes inside the channel area: this is achieved by retrieving the current location (`eco:location`) and the location of the neighbouring node with smallest distance value from D (held in `app:location`) and performing a subtraction operation—the semantics of this subtraction depends on the actual shape of coordinates used for identifying locations. The LSA it produces and maintains is representative of the direction of the sign to be shown on pervasive displays, and is intended to be observed by a display agent.

12.4 Simulation in ALCHEMIST

We provide a brief qualitative and quantitative account of the behaviour of the channel distributed data structure, with the goal of validating the design of agents as defined above.

We first discuss some qualitative aspects, with help of the snapshot in Figure 12.4, showing the result of a simulation of a channel structure in a high-density network, created by running an equivalent model in the Proto simulator [BB06]. First, one can note that the resulting channel data structure is highly independent of local aspects of network topology: provided that nodes

```
% ChannelSource: initiates the channel propagation
?BND_GRAD := new(eco:bond_prop = app:type, eco:bond_value = app:grad);
?SRC := new();  % the source LSA initing channel propagation
iterate {
  ?BND_GRAD.obs(app:distance = ?R);         % perceiving POI's gradient
  ?SRC.upd(app:src = true, app:type = app:rng,
           app:range = ?R, app:width = ?W)            % updating source
  :
  ?SRC.upd(clear);
}

%ChannelService: computes the final channel data structure in each node
?BND_DST := new(eco:bond_prop = app:type , eco:bond_val = app:grad);
?BND_RNG := new(eco:bond_prop = app:type , eco:bond_val = app:rng);
?CHN := new(app:type = app:chn, app:active = "false");
iterate {  % observes ?D,?S,?R,?W and computes channel activation flag
    ?BND_DST.obs(app:distance = ?D);
    ?BND_RNG.obs(app:distance = ?S, app:range = ?R, app:width = ?W);
    ?CHN.upd(app:active = eval("?S + ?D < ?R + ?W"));
    :
    ?CHN.upd(app:active = "false");
}

% SteeringService
?BND_LOC := new(eco:bond_prop = eco:location, eco:bond_value = "true")
?BND_CHN := new(eco:bond_prop = app:type app:active)
?BND_FIE := new(eco:bond_prop = app:temp, eco:bond_value = "false");
?DIR := new();
iterate {
  ?BND_CHN.obs(app:active = "true"); ?BND_LOC.obs(eco:location = ?MYLOC);
  ?BND_FIE.obs(app:location = ?DIRLOC);
  ?DIR.upd(app:direction = eval("?MYLOC - ?DIRLOC"))
  :
  ?DIR.upd(empty);
}
```

Figure 12.3: Multi-agent system for the distributed channel

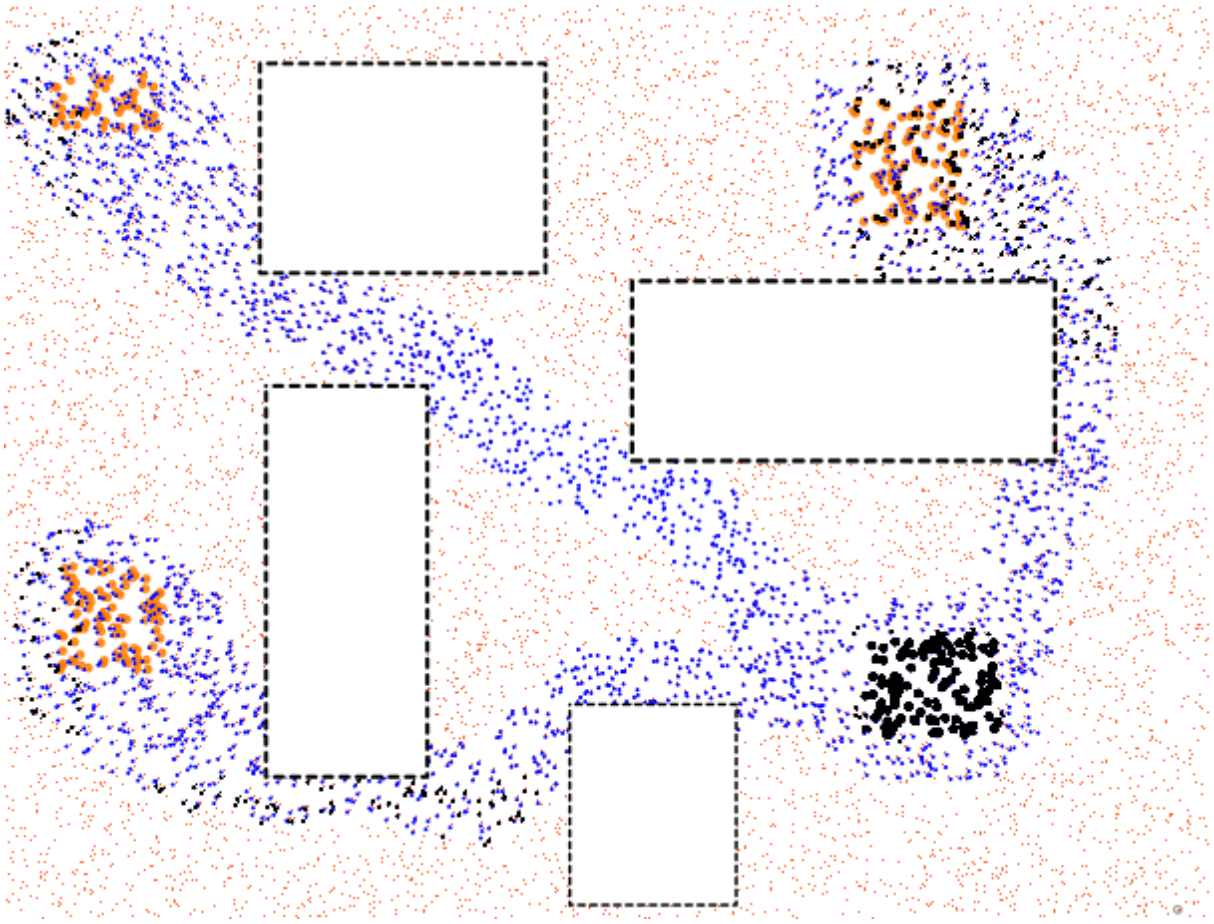


Figure 12.4: Channels (blue) route between source (orange) and destination (black), deployed in a high-density environment of 10'000 nodes (right) with blank areas acting as obstacles.

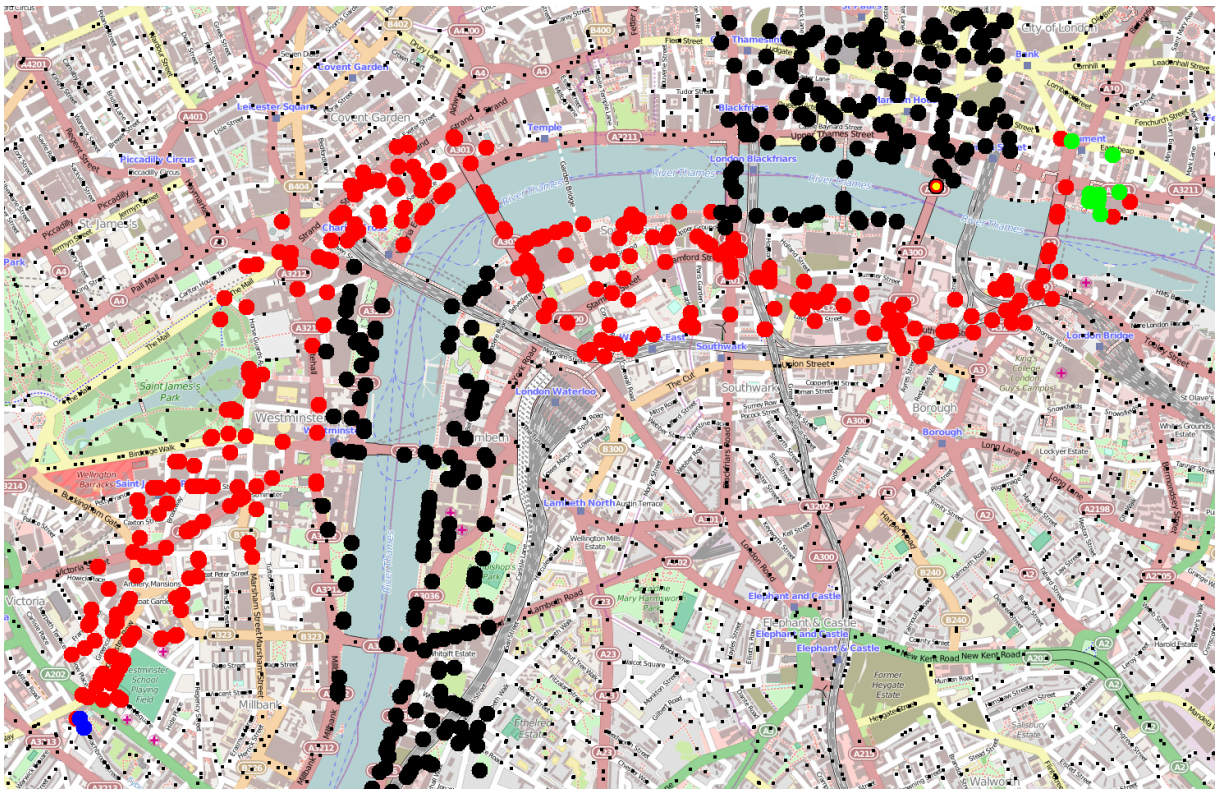


Figure 12.5: The distributed channels deployed in a smartcity (London) for pedestrian steering. Red nodes represent devices inside the channel area, while black nodes correspond to “obstacle” areas.

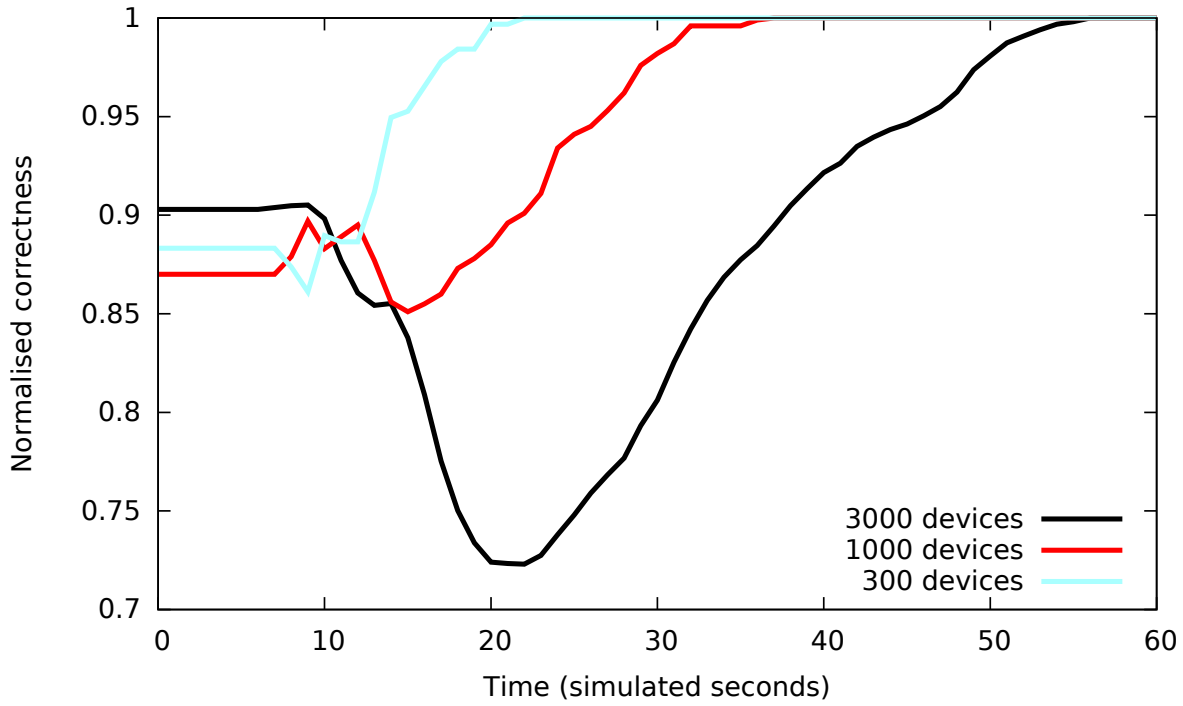


Figure 12.6: Percentage of devices correctly showing channel information over time, using topologies with different densities.

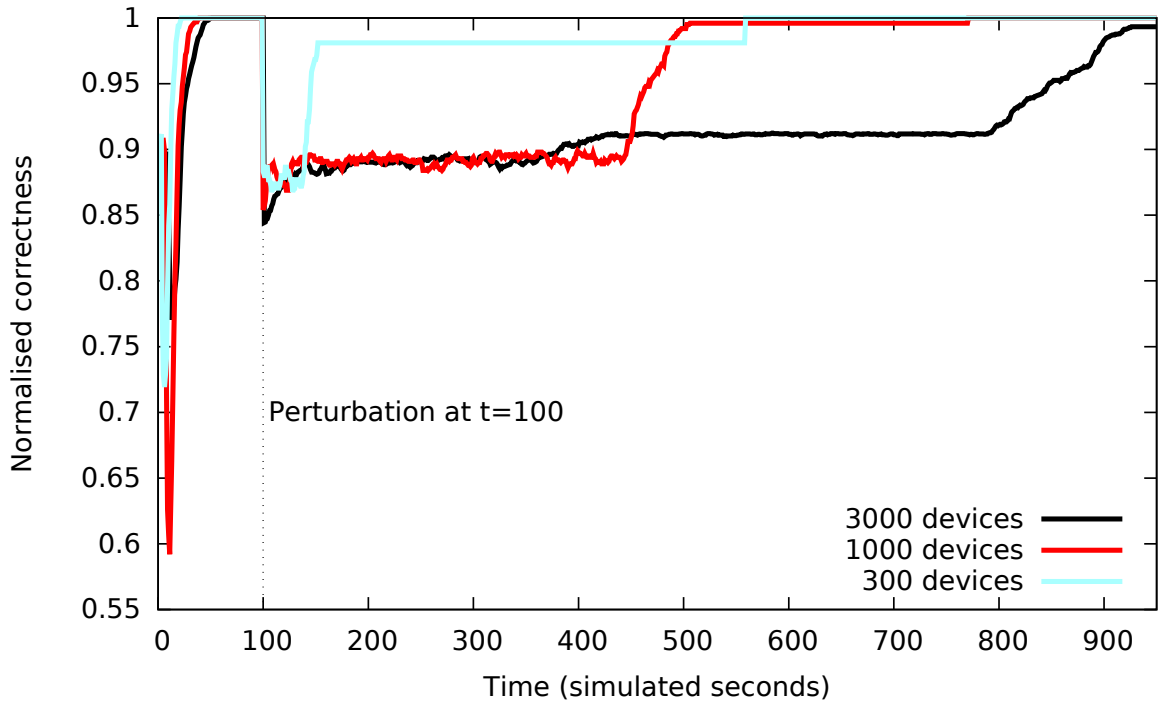


Figure 12.7: Percentage of devices correctly showing channel information in the presence of addition of an obstacle.

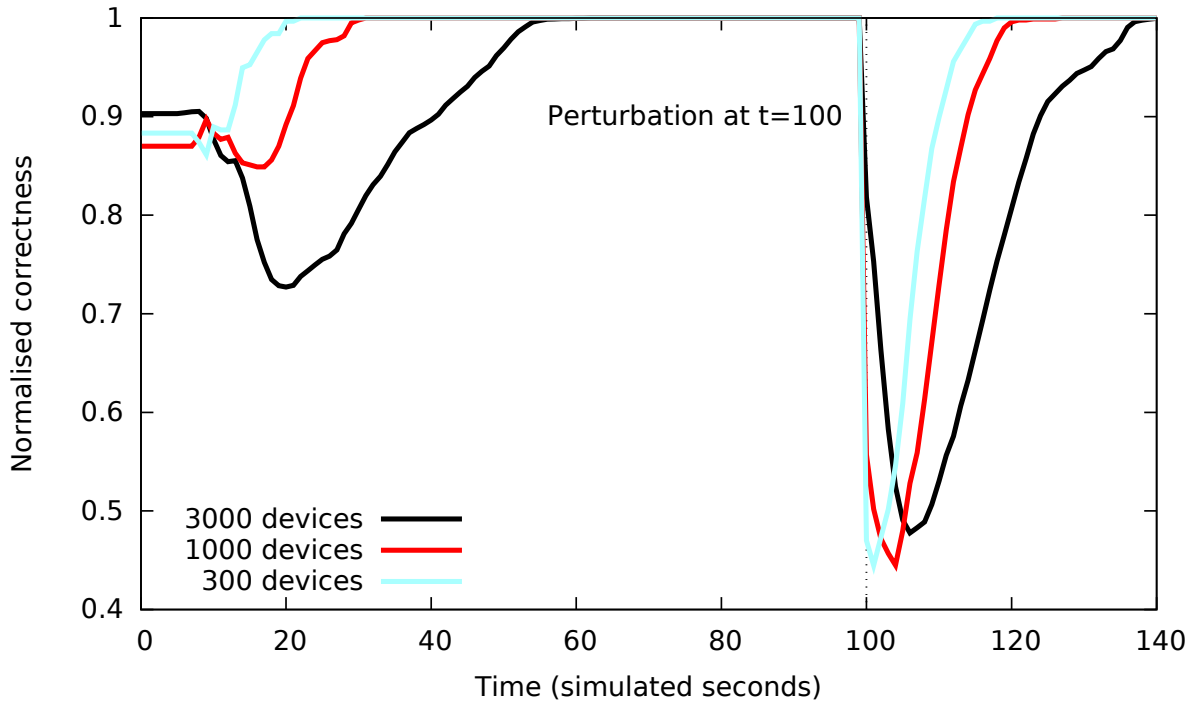


Figure 12.8: Percentage of devices correctly showing channel information in the presence of removal of an obstacle.

are spread more or less randomly, only density affects the final shape, and in particular, the "width" of the channel[2]. Second, the way the channel is created allows one to use distributed concepts of "source" and "destination" of a channel, and even to consider the case of a channel starting from multiple sources—e.g., to guide different groups of interested people. This is because self-organisation is achieved here by the functional combination of well-engineered spatial structures, as advocated in works such as [BDU+13]. Third, this structure automatically adapts to the shape of the network, properly bypassing obstacle areas. This happens since it can be shown that the diffusion-aggregation process underlying gradients enjoys self-stabilisation as defined in [VD14] (diffusion always increments distance, and aggregation is performed based on minimum values).

The presented technique can hence be usefully employed in complex environments, where it is difficult to predict the contingent situations. We consider, as an example scenario, pedestrian steering in smartcities. Figure 12.5 shows a snapshot of a simulation corresponding to a deployment over a London map. Devices were displaced with different density for each experiment, totalling 300, 1000 and 3000 devices respectively. Devices are located on streets available to pedestrians, and device connections are based on proximity. We considered as the POI Monument Street, and created a channel starting from the Royal Horticultural Society, where interested people may be initially located. Dark nodes are those in which some form of unfavourable condition is assumed to be detected, requiring dynamic re-route along different paths.

From a qualitative point of view, Figure 12.5 confirms the self-adaptive character of the distributed channel data structure, as we discussed above. From a quantitative point of view, we measured the time for this data structure to establish under different conditions. Starting from the basic scenario depicted in Figure 12.5, in Figure 12.6 we investigated the time needed to reach a full stabilisation of the correct channel data structure. It charts the percentage of nodes which reached the correct final state over time. The result is that stabilisation is always eventually reached, in an average time that grows linearly with network diameter. The speed of information diffusion is approximatively computed as the distance of nodes multiplied by the frequency at which they run the computation round of eco-laws. Figure 12.7 and Figure 12.8 study self-stabilisation in response to changes in the environment, corresponding to the addition/removal of all obstacles shown in Figure 12.5, which requires a complete (spontaneous) re-calculation of the path. These charts confirm the estimation of stabilisation time studied in [BBVT08]. Namely, the repair time is much faster with removal of obstacles, since in this case gradient values quickly fall down to their new values (i.e., distances quickly decrease). On the other hand, with obstacles added instead, values more slowly grow to the new stable state. This happens because of the issue of slow "raising values" pointed out in [BBVT08].

[2]This is because we used a hop-count-gradient to propagate distances: full independence of density can be achieved by suitably reifying estimated distances between nodes as proper LSAs, and using such information when re-aggregating LSAs—which we did not develop for the sake of simplicity.

13
Pattern: Anticipative adaptation

14
Case Study: Crowd evacuation

15 Case Study: Crowd steering

In this chapter we show a crowd steering case study in which a middleware has the goal of leading people in the desired location within a complex environment in short time, avoiding obstacles such as crowded regions and without global supervisioning.

15.1 Reference scenario

Consider a museum with a set of rooms, whose floor is covered with a network of computational devices (infrastructure nodes). These devices can exchange information with each other based on proximity, sense the presence of visitors, and hold information about expositions currently active in the museum. Each room has four exits and they are connected via external corridors. Visitors wandering the museum are equipped with a hand-held device that holds the visitor's preferences. By interaction with infrastructure nodes, a visitor can be guided towards rooms with a target matching their interest, thanks to signs dynamically appearing on his smartphone or on public displays. This is done using techniques suggested in the field of spatial computing [VCMZ11]—namely, computational gradients injected in a source and diffusing around such that each node holds the minimum distance value from the source.

The environment is a continuous bidimensional space with walls. Smartphones (or public displays) are agents dynamically linked with the nearest infrastructure node – the neighbours are the sensors inside a certain radius r, parameter of the model – from which they can retrieve data in order to suggest visitors where to go. Visitors are agents which follow the advices of their hand-held device (or public displays). It is defined a minimum possible distance between them, so as to model the physical limit and the fact that two visitors can't be in the same place at the same time. Visitors can move of fixed size steps inside the environment. If an obstacle is on their path, their movement is shortened to the allowed position nearest to the desired place.

15.2 Steering strategy

All the information exchanged is in form of annotations, simply modelled as tuples $\langle v_1, \ldots, v_n \rangle$ (ordered sequence) of typed values, which could be for example numbers, strings, structured

types, or function names. There are three forms of annotations used in this scenario:

$$\langle \texttt{source}, id, type, N_{max} \rangle$$
$$\langle \texttt{field}, id, type, value, tstamp \rangle$$
$$\langle \texttt{info}, id, crowd, M, t' \rangle$$

A **source** annotation is used as a source with the goal of generating a field: *id* labels the source so as to distinguish sources of the same type; *type* indicates the type of fields in order to distinguish different expositions; N_{max} is the field's maximum value. A **field** annotation is used for individual values in a gradient: *value* indicates the individual value; the *tstamp* reflects the time of creation of the annotation; the other parameters are like in the source annotation. An **info** annotation is supposed to be created and kept up to date by each sensor. M represents the number of smartphones the sensor is perceiving as neighbours.

The rules are expressed in form of chemical-resembling laws, working over patterns of annotations. One such pattern P is basically an annotation which may have some variable in place of one or more arguments of a tuple, and an annotation L is matched to the pattern P if there exists a substitution of variables which applied to P gives L. A law is hence of the kind $P_1, \ldots, P_n \overset{r}{\mapsto} P'_1, \ldots, P'_m$, where: *(i)* the left-hand side (reagents) specifies patterns that should match annotations $L_1, \ldots, L_n$ to be extracted from the local annotation space; *(ii)* the right-hand side (products) specifies patterns of annotations which are accordingly to be inserted back in the space (after applying substitutions found when extracting reagents, as in standard logic-based rule approaches); and *(iii)* rate r is a numerical positive value indicating the average frequency at which the law is to be fired—namely, we model execution of the law as a CTMC transition with Markovian rate (average frequency) r. If no rate is given the reaction is meant to be executed "as soon as possible", which means that the rate that associated with the reaction tends to infinite. To allow interaction between different nodes (hence, annotation spaces), we introduce the concept of *remote pattern*, written $+P$, which is a pattern that will be matched with an annotation occurring in a neighbouring space.

As sources annotations are injected in nodes, gradients are built by the first three rules in Figure 15.1. The first one, given a source, initiates the field with its possible maximum value. The second one, when a node contains a field annotation, spreads a copy of it to a neighbouring node picked up randomly with a new value computed considering the crowding around the sensor. The parameter k allows to tune how much crowding should influence the field, while $\#D$ is the measure of the distance between the two involved sensors. As a consequence of these laws, each node will carry a field annotation indicating the topological distance from the source. The closest is the field value to N_{max}, the nearest is the field source. When the spread values reach the minimum value 0, the gradient has to become a plateau.

To address the dynamism of the scenario where people move, targets being possibly shifted, and crowds forming and dissolving, we introduced the following mechanism. We expect that if a gradient source moves the diffused value has to change according to the new position. This is the purpose of the *tstamp* parameter which is used in the fourth law, continuously updating old

$$\langle \texttt{source}, id, type, N_{max} \rangle \xmapsto{r_{init}} \langle \texttt{source}, id, type, N_{max} \rangle, \langle \texttt{field}, id, type, N_{max}, \#T \rangle$$

$$\langle \texttt{field}, id, type, N, t \rangle \xmapsto{r_{diff}} \langle \texttt{field}, id, type, N, t \rangle, +\langle \texttt{pre\_field}, id, type, N - \#D, t \rangle$$

$$\langle \texttt{pre\_field}, id, type, N, t \rangle, \langle \texttt{info}, id, crowd, M, t' \rangle \mapsto \langle \texttt{field}, id, type, N - k * C, t \rangle, \langle \texttt{info}, id, crowd, M, t' \rangle$$

$$\langle \texttt{field}, id, type, N, t \rangle, \langle \texttt{field}, id, type, M, t' + t \rangle \mapsto \langle \texttt{field}, id, type, M, t' + t \rangle$$

$$\langle \texttt{field}, id, type, N, t \rangle, \langle \texttt{field}, id, type, M, t \rangle \mapsto \langle \texttt{field}, id, type, max(M, N), t \rangle$$

$$\langle \texttt{field}, id, type, N, t \rangle, \langle \texttt{field}, id', type, N + M, t' \rangle \mapsto \langle \texttt{field}, id', type, N + M, t' \rangle$$

Figure 15.1: Laws describing the museum application.

values by more recent ones (*youngest* law). In this way we ensure that the system is able to adapt to changes of the source states. Finally, the spreading law above will produce duplicate values in locations, due to multiple sources of the same type (indicated by different ids), multiple paths to a source, or even diffusion of multiple annotations over time. For this reason we introduced the last two laws. They retain only the maximum value, i.e. the minimum distance, the former when there are two identical annotations with only a different value, the latter when the id is different (*shortest* laws). The proposed solution is intrinsically able to dynamically adapt to unexpected events (like node failures, network isolation, crowd formation, and so on) while maintaining its functionality.

People are modelled as nodes programmed with a single reaction with no conditions and having, as action, the ability to follow the highest field value among neighbouring sensors. For the crowding annotation, we may assume that sensors are calibrated so as to locally inject and periodically update it by setting the current level of crowding, *i.e.* the number of persons. The reaction rates are identified by hand performing different simulations with different parameters. The results reported are obtained with $r_{init} = 1$ and $r_{diff} = 50$. The other laws show no rate because it is assumed to be infinite (as-soon-as-possible semantics).

15.3 Simulation in ALCHEMIST

We here present simulations conducted over an exposition, where nine rooms are connected via corridors. People can express different preferences represented by their shape.

Three snapshots of a first simulation run are reported in Figure 15.2. We here consider four different targets that are located in the four rooms near environment angles. People are initially

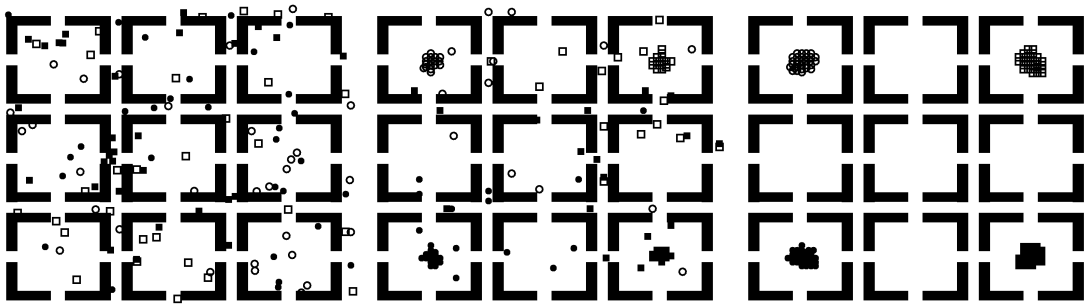


Figure 15.2: A simulation run of the reference exposition: three snapshots of ALCHEMIST's graphic reporting module with this simulation.

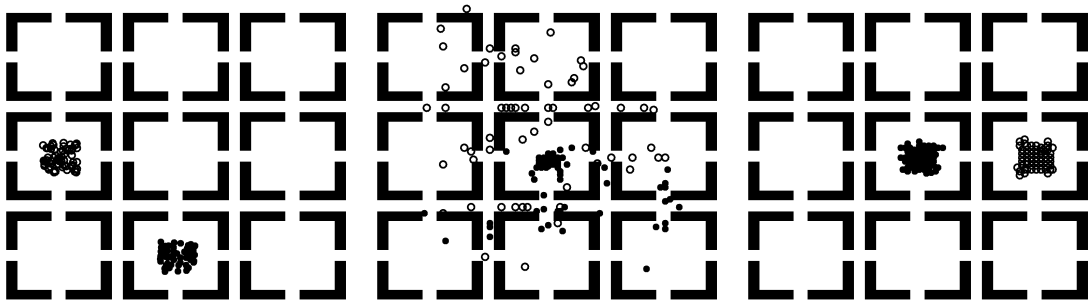


Figure 15.3: A run showing the effect of crowding: dark visitors occupy a central room, making other visitors moving left to right by a longer, less crowded path

spread randomly in the museum, as shown in the first snapshot, and they eventually reach the room in which the desired target is hosted, as shown in the last snapshot.

Figure 15.3 shows a simulation experimenting with the effect of crowding in the movement of people. Two groups of people – denoted with empty and full circles – with common interests are initially located in two different rooms, as shown in the first snapshot. The target for the dark visitors is located in the central room of the second row, while the others' is in the right room of the second row. In the simulation, dark visitors reach their target soon because it is nearer, thus forming a crowded area intersecting the shortest path towards the target for the other visitors. Due to this jam the latter visitors choose a different path that is longer but less crowded.

Both tests show qualitative effectiveness of the proposed laws, and suggest that our simulation approach can be used for additional experiments focussing on tuning system parameters (factor k) or alternative strategies (e.g., diffusing crowd information) to optimise paths to destinations. For instance, in the context of the second case, Figure 15.4 shows how factor k can influence the time for (sub)groups of (light) people to reach the destination, by which we can see that even small values of k lead to a significant improvement—which slowly decreases as k grows.

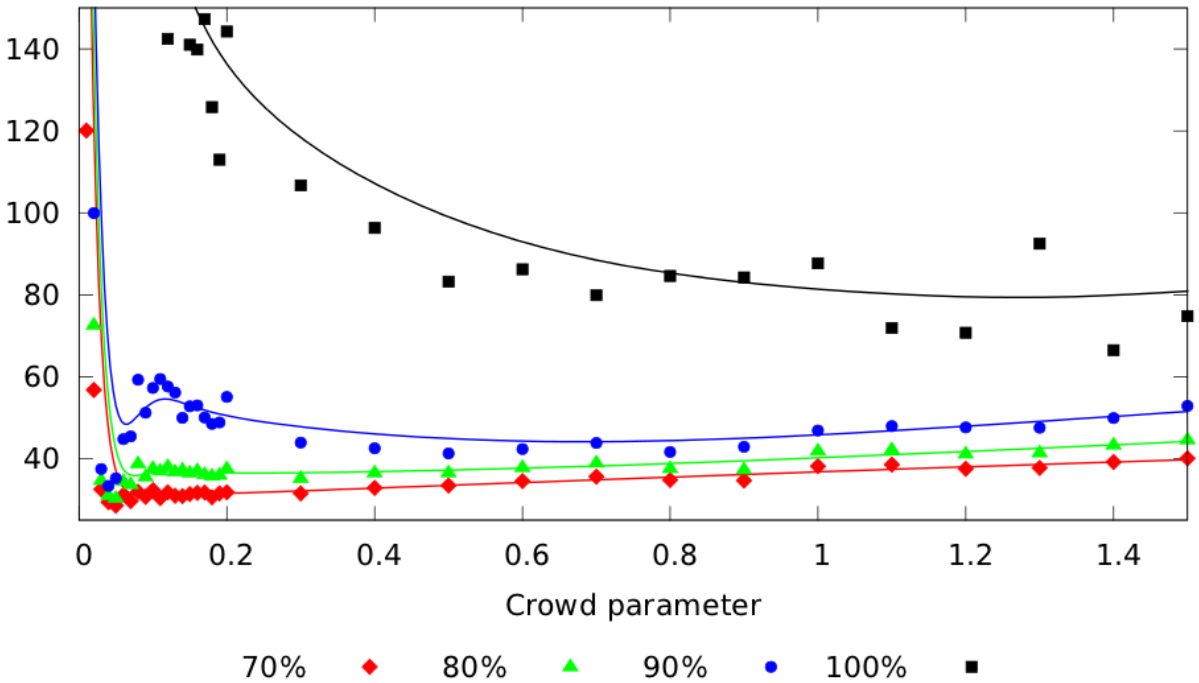


Figure 15.4: Time units of convergence time with different values of crowd parameter and different percentages of people

16 Case study: Self composition

Services are one of the main actors in pervasive and ubiquitous computing. Composing services is a big issue in these contexts, providing new applications or high level services. Self-composition is even more challenging.

In literature some surveys and possible solutions have been proposed, but only few of them give concrete proposals for the design, implementation and evaluation of service composition. Moreover the composition of services is solved in a centralised way, solution which hardly deal with the large-scale distributed property of pervasive systems, or the composition is specified by the application developers, preventing the system to use new services that appear at runtime.

In this chapter we frame/situate the problem of self-composition in the framework of pervasive ecosystems, defining what a service is and what composing services means. We then propose an approach to perform self-composition of services and demonstrate our ideas showing how gradients can be composed with other services. In particular our discussion focuses on the composition of gradient with services providing local levels of crowd, at the purpose of avoiding path crossing overcrowded areas. This example is modelled in terms of LSAs and eco-laws, that are the basic components of a pervasive ecosystem [VPMS12], and preliminary simulations, aimed at validating the approach, are run on top of ALCHEMIST.

16.1 Self-Composition approaches

A software service represents a functionality that can be provided on demand in order to create higher level software artefacts, such as applications or higher level services. The notion of service allows applications to be easily developed and to adapt by changing the different services, which the application is composed by, at run-time. Self-composition of services involves the automatic discovery of new services by composing available ones, the selection of services, the plan to execute them (i.e. services execution order), and the adaptation of the composed service when new requirements appear or a service disappears from the system.

Service Composition in SOA

The static character of traditional composition approaches such as orchestration and choreography has been recently challenged by so-called dynamic service composition approaches, involving semantic relations [BBG06], and/or AI planning techniques to generate process automatically based on the specification of a problem [WRG+07]. Their basic goal is to analyse a description of the services to compose and compute a composition satisfying structural/behaviour/ontological compliance of the result of composition as can be deduced from information about the composites. One of the main challenges of these approaches is their limited scalability and the strong requirements they pose on the detail of service description.

Evolutionary techniques

Evolutionary approaches such as those based on Genetics Algorithms (GA) have been proposed as well for service composition, such as in [CDPEV05], motivated by the need of determining the services participating in a composition that satisfies certain QoS constraints, which is known to be a NP-Hard problem. More generally, evolutionary approaches have been used in the field of autonomic computing to establish the set of norms, policies or rules that drive the system to the desired emergent behaviour [SRAA10]. We observe that this approach has a potential for service composition: in the same way as the genetic programming determine the proper execution of functions and their parameters, it could be used to find which services, their order of execution and the parameters that its service should receive.

Competition-based approaches

In order to tackle the potentially high number of different compositions that can arise in an open system, in [Vir11] a mechanism of *coordination reactions* for networked tuple spaces is proposed, showing how it can be applied to support competition and composition in a fully distributed setting—services opportunistically compose in those regions of the network where this results in a more competitive service. While all compositions can possibly take place, thanks to a positive/negative feedback only successful ones are meant to "survive" in the system, the others fading until becoming never actually selected. This is achieved by matching services with requests through a self-regulating dynamics inspired by prey-predator model of population dynamics [Ber92b], and by diffusing service tuples using a computational-field approach similar to the one presented in [MZ05, BB06].

16.2 A comprehensive approach for pervasive ecosystems

As previously emphasised, the issue of service composition is becoming wider and wider, and is now encompassing new research areas. Among them – pervasive computing, autonomic computing, robotics – we here focus on pervasive computing, and in particular, on the framework

of pervasive ecosystems [ZV11]. There, we understand and design a very large and dense pervasive computing system as a sort of substrate in which humans, institutions, software systems and devices, inject services of various kinds without an a-priori knowledge of the structure and behaviour of those already available and those that will be injected later. Such services diffuse in the network, and are situated and context-aware in the sense that they contextualise to (i.e., they will affect and be affected by) the situation in each *niche* of it. In turn, while in standard SOA a service is a centralised point of functionality, interacting with its clients by stateful protocols of message-passing actions, a service is here a more generalised concept: it is any activity, triggered by the intention of some localised software agent, that flows in the system updating the spatial and temporal configuration of other services, and generating a set of events ultimately perceived by the other agents in the system. Examples of such services include: the materialisation of data produced by all sensors available around, routing services able to interconnect devices based on their physic al, logical or social proximity, local/global advertisers of the availability of some content, situation recommenders capturing relevant information about a context, and so on. Typical application scenarios include pervasive display ecosystems, augmented social reality, and traffic control—the reader is deferred to [ZV11] for a more deeper description of them.

16.3 The self-composition issue in pervasive service ecosystems

Pervasive ecosystems naturally call for a radical shift in the way software is designed and developed. The concept of "software system" loses its standard meaning of a monolitically designed/implemented/deployed artefact. It is rather a mash-up of services, and the development of a "new software system" simply translates into the development of additional services to be immersed in a scenario of existing ones, with the goal of composing with them and evolving their functionalities, thus making software live in a sort of "eternal beta" state [KC09].

The natural questions we have to answer in order to support this vision are hence: what can make services compose if they were not explicitly designed to do so? How can we decide "whether" and "how" two or more services should compose? How can such decisions be continuously reconsidered depending on the actual (spatial/temporal context) in which the composition is deployed? Can we make multi-level composition seemingly work as well?

We refer to this problem as the self-composition problem for service ecosystems, where by "self-" we mean that the underlying middleware makes sure that services tend to compose in order to form meaningful new services, in a way that users of the ecosystems simply perceive only atomic and composite services that are actually useful in practice.

We argue that this vision cannot be fulfilled by the above-mentioned approaches taken in isolation, but instead require them to work altogether. Advanced (semantic) matching of services is key to decide whether two services deployed by third parties can be composed together. Evolutionary approaches seem the only solution to the problem of finely tuning the parameters used in the composition, preselecting, for instance, a subset of more promising compositions. And

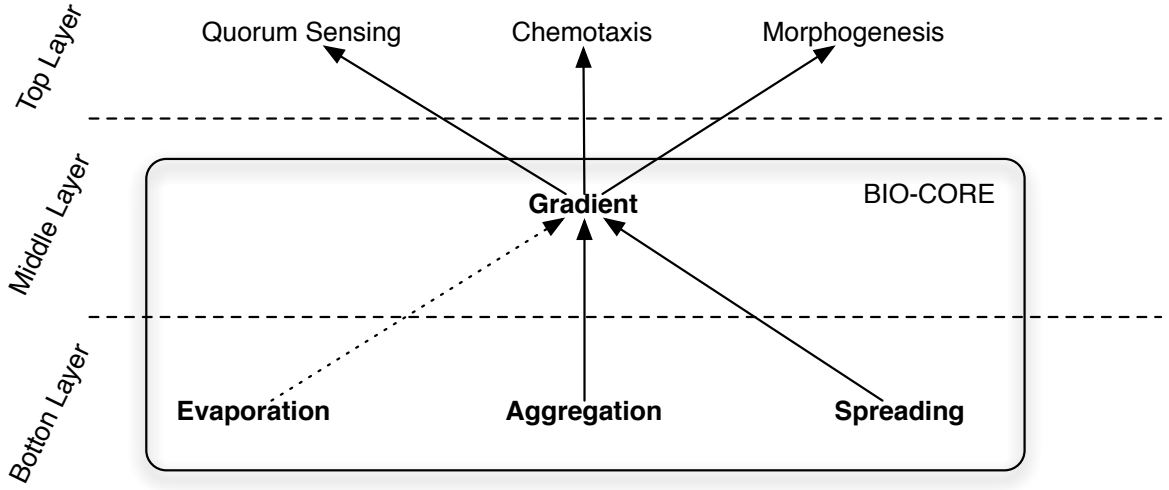


Figure 16.1: Self-organising patterns and their relationships

finally, competition of different compositions based on their actual exploitation appears instead as the only viable approach to measure the quality of a composition, hence promoting successful ones. On the other hand, understanding how they can work together in a comprehensive framework is very difficult, and generally depends on the specific platform and service model one adopts.

16.3.1 Self-Composition with Gradient service

To ground our discussion of requirements and possible solutions, we here outline a paradigmatic example of composition in service ecosystems. We consider a crowd steering scenario, based on the idea of guiding people towards locations hosting events of interest in a complex and dynamic environment (using semantic matching with people's interests) without any supervision, namely, in a self-organised way. In particular, we consider a museum with a set of rooms connected by corridors, whose floor is covered with a network of computational devices (called sensor nodes). These devices exchange information with each other based on proximity, have sensors of various kinds, and hold information as LSAs (e.g., about exhibits currently active in the museum). Visitors exploring the museum are equipped with a smartphone device that holds their preferences. By interaction with sensor nodes, a visitor can be guided towards rooms with a target matching his/her interest, thanks to signs dynamically appearing on the smartphone or on public displays. This system is based on a gradient service, a key brick of self-organisation mechanisms (see Figure 16.1) that is typically designed to provide optimal paths to roam a distributed system even in the context of very articulated environments (a building with rooms and corridors, or a traffic scenario) and by dynamically adapting (namely, self-healing) to unpredicted situations

such as sudden road interruption [FMDMSM$^+$12, FMSM12]. Gradient starts from a source LSA located at an exhibition and spreads clones of it to all the nodes of the network by eco-laws of spreading and aggregation, basic building patterns as shown in Figure 16.1. In turn, as the gradient is stabilised, in each node the minimum distance value is kept, so that the shortest path to the source is automatically (and dynamically) defined [PMV11]. In this way, a person willing to reach a point of interest simply "binds" to a gradient LSA, and public/private displays show signs to guide her to the source.

However, other services exist in the system, such as those provided by the many sensors available around. Among them we have the crowd detection service, enacted by the injection in each node on the floor of a crowd LSA reporting the crowd level sensed there (0 is no crowd, 1 is maximum crowd). An other service is the one that, given visitor preferences and profile, defines a set of other possible points (exhibitions, toilets, refresh points) the user can found interesting or useful. This service can be an external agent that is able to process visitor information, for instance defining related exhibitions, so as to provide, in the form of an LSA, a list of best points the user can pass by. Then we can have sensors for temperature, sensors for pollution, accelerometers etc. More generally, being a pervasive system an open system, new services may appear in the system, and autonomously be composed with existing ones, without a predefined schema. Only those compositions that create useful services will then survive. To exemplify the idea, in the following we describe two self-composition problem we envisioned.

Crowd composition

This self-composition problem focuses on making the gradient service automatically compose to the crowd detection service so that the estimated distance to the source gets "penalised" in case of a high crowd value, which implies that the path computed towards a point of interest intrinsically takes into account situations of crowded areas dynamically emerging in the environment, properly circumventing them. In particular what we expect to the middleware to automatically do is: decide to compose the gradient with crowd sensing instead of with other (irrelevant sensors), and optimally tune how much crowd information should affect the computation of optimal path.

Ad-hoc path composition

If the gradient service is tagged with the point of interests crossed, it can be composed with the list of "best points" defined for each user, so that, in case of intersection, such gradient gets "reinforced" for instance diminishing its value, which then implies that the direction it comes from has more chance to be chosen, and paths which cross points interesting or useful for the user are likely to be followed. In particular what we expect to the middleware to automatically do is: decide to compose the gradient with the LSA representing the "best points", and optimally tune to which extent information on preferences should affect the computation of optimal path.

16.3.2 A prototype solution

Starting from a number of services available in those nodes where sources of gradient services are located – all reified by an LSA that can be injected in different parts of the system but via chemotaxis reach gradient sources – the solution we propose is based on the following mechanisms:

Composition

Based on one or more "composition recommender" agents available in source nodes, all the available compositions are computed using techniques of advanced matching of interface/behaviour/ontology. These are ranked – according for instance to feedbacks from the users on the composite service, as discussed later – and the description of a limited number of them is reified in the space in the form of a set of LSAs. In the example of crowd steering proposed above, these compositions should result in a set of LSAs representing the source of different gradients, each one created composing the basic gradient with one or more services. We assume that the composition actuated, *i.e.* which services are composed and which parameters are used in the composition, is stored into a property of the gradient LSA, so that the same composition can be performed in all the locations of the system.

Contextualisation

From the source, composite gradients and the basic one, are diffused. In each location they are contextualised with the local value of those services they are composed by. Contextualisations involving gradients normally result in a modification of the local value of distance according to a function δ specified in the service or defined by the composition agent. If services are not locally available, contextualisation does not happen and the distance value does not change. For instance, contextualise gradient with the crowd level perceived in each location at the purpose of discouraging paths crossing crowded areas. It means that distance should be increased and function δ looks like $\delta = k * c$ where k is a function parameter (see later how to define its values), and c is the local crowd level.

Choice

The resulting effect is that the gradient service is available in the system in different forms (the basic one and the composite ones), and users, once entering in the system, can probabilistically choose among them, grounding such decision mainly on the distance and satisfaction values. They will then follow the same composition for the whole path.

Feedback

Each gradient owns a property called "satisfaction", representing the feedback of users on the quality of the service itself. It can for instance evaluate the length of the path, the interest and so

on: the higher the value is, the higher is the user satisfaction. People, once they reach the source, should then provide their feedbacks in the form of LSAs with predefined properties —such as length, comfort, beauty, interest. Marks, for each property, might be defined in a predefined range. From there one agent can then be in charge to compute the satisfaction value as a function of all the marks given – for instance, the average. The LSA of the correspondent gradient source is then modified by adding to the previous value of the satisfaction property the last computed. This updated value is then spread in the whole system autonomously as a result of gradient diffusion.

Evaporation

The satisfaction value can also be interpreted as the relevance value of the Evaporation pattern [FMDMSM$^+$12]. This parameter gets decreased until fading if not augmented by user positive feedbacks. Once it is equal to zero the composite service can be removed. This mechanism ensures that no satisfactory compositions are decayed.

Evolution

If parameters are to be chosen for the composition – for instance parameter k in $\delta = k * c$, representing how much the crowd level has to influence the distance value of the field –, they can be tuned by agents implementing evolutionary techniques. They are in charge to define the new population of parameters according to the fitness function that can be computed from the satisfaction value of the gradient itself.
The emergent idea is that, after a transition period, system makes available only those services that better fit user preferences. For instance, for those users interested in quickly reach the sources, only those gradients that ensure the lowest arrival time should survive.

16.4 A formal model for gradients self-compositions

To describe how self-composition of gradients can be built in the SAPERE framework, we first introduce the implementation schema for gradients which we shall rely upon to formalise the self-composition task in the specific example of crowd service exploitation.

16.4.1 The gradient service

Our strategy for modelling gradient's features:

1. a continuous broadcast of information in each node as in [Bea09], with latter information always overwriting previous one;
2. propagation of estimated distance, computed by summing the contributions given by property `mid:distance` in neighbour annotation;

3. propagation with no horizon (it is easy to impose a distance bound to the propagation of information);

4. after been sent to a neighbour, an annotation is subject to a contextualisation phase (in which e.g. aggregations are executed) before being spread again.

Such rules are reported in Figure 16.2 and are described in turn. Note that they refer to namespace `mid` for URIs related to middleware activities, and to `sos` for those concerning self-organisation aspects.

Rule [PUMP] is a transformation rule which takes a source annotation and creates a new annotation with distance set to 0, used to start the spreading process. The source annotation should feature property `sos:type` assigned to value `sos:source` (we shall say the annotation is of type `source` for brevity), and it should also have some value assigned to properties `sos:aggr_prop` (the name of the property holding the value used to aggregate other annotations as described below), `sos:step` (an incremental value used to refresh information), `sos:r_diff` (diffusion rate) and `sos:r_ctx` (contextualisation rate). This rules fires at diffusion rate `?R`: according to CTMC semantics, this means that as soon as a matching set of reactant annotations is found, actual application of the rule is delayed of a time t randomly drawn according to a negative exponential distribution of probability with average value $1/$`?R`—namely, it is fired at frequency `?R`. The effect of firing this rule is twofold: it increases the value of property `sos:step` of the source annotation, and creates (by cloning the source) a new annotation `?GRAD` of type `sos:diff` and `sos:aggr` (original type `sos:source` is removed), and declaring distance 0 and orientation `here` with respect to the source—these properties will be update as the annotation is spread around. Note that the right-hand side of a rule mentions only changes in annotations, without any need of repeating information provided in the left-hand side which has not changed: this allows a simpler structuring of rules with respect to more syntactic approaches like the one in [PMV11].

Rule [DIFF] is used to diffuse a cloned version `?GRAD1` of a gradient annotation `?GRAD` in one neighbour. The gradient annotation should be of type `diff`, declare distance `?D` from the source and have diffusion rate `?R`. This rule has as further reactant a neighbour annotation, from which information about a neighbour `?L` with orientation `?O` and distance `?D2` can be extracted. This rule leaves the reactants unchanged, but creates a clone of `?GRAD` with type `ctx` instead of `diff`, to be relocated at `?L` and indicating orientation (with respect to originating location) `?O` and distance `?D+?D2`. Note that continuous application of this rule at rate `?R` makes copies of the gradient annotation to be created and diffused in all neighbours.

An annotation is relocated with type `ctx`, which forbids further application of rule [DIFF]. This is because we first need to aggregate all incoming annotations before a new diffusion is scheduled. To do so, rule [CTX] defers change of type from `ctx` to `diff`, which happens at rate `?RC` (contextualisation rate).

One form of aggregation is due to rule [YOUNGEST], used to refresh gradients with new information. It takes two annotations such that the values associated to the aggregation property `?P` are the same, and retains the one with bigger `sos:step`. The idea is that `sos:aggr_prop`

Transition Rules for Gradients

```
# [PUMP] An annotation of type source continuously injects the initial
# gradient annotation
?SOURCE sos:type sos:source; sos:aggr_prop ?P; sos:step ?T;
     sos:r_diff ?R; sos:r_ctx ?RC
--?R-->
?SOURCE sos:step =(?T+1) +
?GRAD(?GRAD clones ?SOURCE) sos:type -sos:source sos:diff
     sos:aggr; sos:dist "0"; sos:orientation "here"

# [DIFF] A gradient annotation is cloned in a neighbour, with distance
# increased and updated orientation
?GRAD sos:type sos:diff; sos:dist ?D; sos:r_diff ?R +
?NEIGH mid:type mid:#neigh; mid:remote ?L; mid:orientation ?O;
     mid:distance ?D2
--?R-->
?GRAD + ?NEIGH +
?GRAD1(?GRAD1 clones ?GRAD)  sos:type -sos:diff sos:ctx;
     sos:dist=(?D+?D2); sos:orientation =?O; mid:#loc ?L

# [CTX] A contextualising annotation is transformed back into an annotation
# to be diffused
?GRAD sos:type sos:ctx; sos:r_ctx ?RC
--?RC->
?GRAD sos:type sos:-ctx sos:diff;

# [YOUNGEST] Of two annotations to be aggregated based on property ?P, the
# one with newest information is kept
?ANN1 sos:type sos:aggr; sos:aggr_prop ?P; ?P =[?C]; sos:step ?T +
?ANN2 sos:type sos:aggr; sos:aggr_prop ?P2; ?P2 =[?C]; sos:step ?T2(?T2<?T)
--->
?ANN1

# [SHORTEST] Of two annotations to be aggregated based on property ?P,
# the one with shortest distance from source is  kept
?ANN1 sos:type sos:aggr; sos:aggr_prop ?P; ?P =[?C]; sos:dist ?D1;
     sos:step ?T  +
?ANN2 sos:type sos:aggr; sos:aggr_prop ?P2; ?P2 =[?C];
     sos:dist ?D2(?D2>=?D1); sos:step ?T
--->
?ANN1

[DECAY] An annotation decays
?GRA sos:type sos:diff; sos:r_dec ?RD
--?RD->
```

Figure 16.2: Rules for gradients

holds the name of a property `?P` which is expected to contain some value(s) that should be identical in two annotations for them to be aggregated. For instance, it could be the unique id of a gradient source, so that we do not aggregate annotations coming from difference sources—but more involved situations can be programmed as exemplified in next section.

Rule [SHORTEST] is similar: it takes two annotations with same step (hence coming from the same gradient annotation as generated by rule [PUMP]), and retains the one with shortest distance only if, again, they have same content of aggregation property.

Finally, rule [DECAY] is used to remove an annotation at a given decay rate `?RD`. This is useful as a cleanup mechanism in the case a gradient source is removed from the system—though it plays no crucial role in this paper and will be neglected in next discussions.

For the above rules to properly work we need to synthesise a well-structured source annotation, featuring all required properties and proper values for rates—namely, diffusion rate is to be chosen to keep the system sufficiently reactive to changes without bloating the system with undesired messages, and contextualisation rate should be significantly higher than diffusion rate (at least one order of magnitude). For instance we could use the following source annotation:

```
:id314 mid:#loc :loc117;  sos:type sos:source;
       sos:step "0";      sos:sourceid "341AB2"
       sos:aggr_prop sos:sourceid;
       sos:r_diff "10";   sos:r_ctx "100"
```

16.4.2 Rules for Gradient Composition

Rule [COMPOSITION] in Figure 16.3, is in charge of creating the source of the composite gradient. The new source owns the same set of properties of the basic one, as soon as it refers to the same PoI, but it also defines which properties are to be modified as an effect of the composition – `sos:dist` –, and which are composition parameters – `scm:parameters` – , *i.e.* function – `scm:crowd_op` – to be used for modifying the distance value, and coefficient —`scm:crowd_factor`. In this preliminary example we adopt a simple linear function of the crowd level perceived by sensors in each location. It increases the distance of those paths crossing crowded area, if the coefficient has an opportune value. Rule [CONTEXTUALISATION] applies the composition specified in `scm:parameters`, in each location of the system. Rule [FEEDBACK] is used to modify the satisfaction value of gradient, taking into account the velocity of the path chosen. It can be computed by the system itself, or provided by the user. The value of `scm:satisfaction` gets increased by `scm:velocity` value.

Rules [EVAPORATION] and [DECAY] are finally used to evaporate the `scm:satisfaction` value according to an evaporation factor `scm:factor_ev ?FE` in the range of [0,1] and to remove the gradient composition in case of a `scm:satisfaction` value equal to zero, or negative, ensuring that only those compositions that compute fast paths will survive.

Eco-laws for gradient contextualisation

```
# [COMPOSITION] The gradient source is composed with the crowd service
?SOURCE sos:type sos:source; scm:satisfaction ?S +
?CROWD scm:type crowd; crowd:level ?CL
--->
?SOURCE +
?CSOURCE(?CSOURCE clones ?SOURCE) scm:property sos:dist;
     scm:parameters scm:crowd_op  ?CF;  scm:crowd_op  ?CF*?CL

# [CONTEXTUALISATION] If sensors perceive crowd, the gradient distance is
# augmented
?GRAD sos:type sos ctx; sos:dist ?D;  scm:property sos:dist;
     scm:parameters scm:crowd_op scm:crowd_factor; scm:crowd_factor ?CF;
     scm:crowd_op  ?CF*?CL +
?CROWD scm:type crowd;  crowd:level ?CL
--->
?CROWD + ?GRAD  sos:type -sos:ctx sos:diff; sos:dist =(?D+?CF*?CL)

# [FEEDBACK] Feedbacks are used to update the satisfaction values
?FEEDBACK scm:parameters scm:crowd_op; scm:feedback  scm:velocity;
scm:velocity ?V +
?GRAD scm:satisfaction ?S; scm:parameters scm:crowd_op
--->
?GRAD scm:satisfaction =(?S+?V)

# [EVAPORATION] The gradient satisfaction value gets decreased
?GRAD scm:satisfaction ?S; scm:factor_ev ?FE; scm:r_ev ?RE
--?RE->
?GRAD scm:satisfaction =(?FE*?S)

# [DECAY] If the gradient satisfaction value becomes zero that composition
# is removed
?GRAD scm:satisfaction "0";
--->
```

Figure 16.3: Additional rules for anticipative gradients.

16.5 Towards simulation of gradient self-compositions

We checked the correctness of the proposed self-composition solution by simulation, conducted using ALCHEMIST.

Results presented show early experiments on gradient composition with crowd level, where people dynamically enter in the system and begin to move towards the target ascending one of the different compositions available. Movement velocity is constant in the system, except for crowded areas, where it decreases according to the crowd level perceived: higher crowd, slower velocity. In these early experiments, parameter k in function δ is not computed dynamically

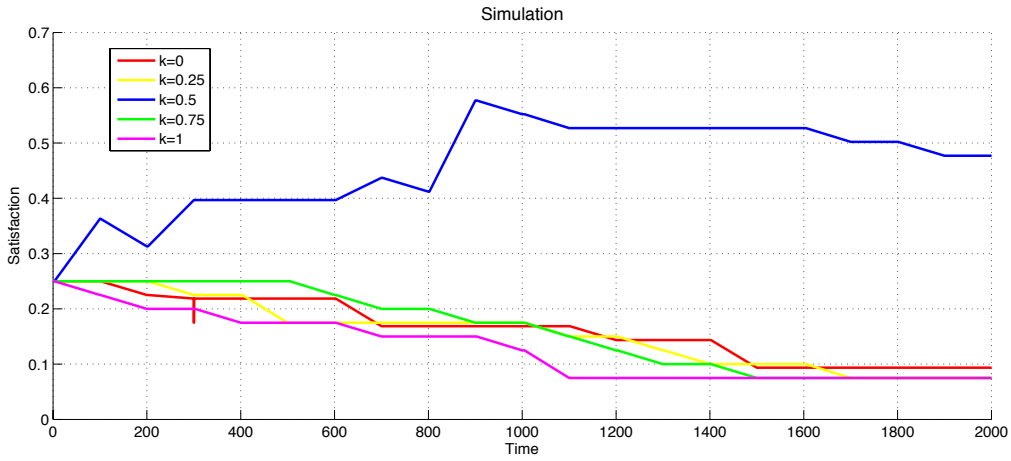


Figure 16.4: Satisfaction values for different compositions changing over time.

through an evolutionary algorithm but composition agents define a set of possible values. A new composition corresponds at each value. Users chose one gradients randomly but considering also the associated satisfaction value that measures the velocity of the best path each gradient proposes. This feedback is computed by each user once reached the target, given the length of the path and the distance walked.

The goal is to observe that, satisfaction values, initially the same for each composition, dynamically change according to feedbacks from the users, and only best compositions survive. In Figure 16.4 such process is shown for different values of parameter k: the system reaches a stable state with $k = 0.5$ ensuring the faster path while the other compositions decay.

17

Case study: Semantic resource discovery

18

Case study: Crowd disasters prediction

19 Case study: Crowd steering at the urban scale

In section, we focus on the problem of crowd steering at the urban scale. The problem is noticeably different from the classic per-user navigation: we want to consider the current position of people to make the user avoid congested areas, thus reducing the overall trip time and improving the security level. Also, this differs from the experiment in Chapter 15 for the complexity of the environment and the number of devices involved. Crowds of people vary with time, and consequently we need a system able to dynamically adapt to such changes. As per our line of research, we obviously want the computation to happen in a completely distributed fashion, with no centralised computing system involved.

19.1 Devices and physical configuration

We suppose users to be equipped with smart devices, also able to communicate with other devices within a certain range with a wireless technology, and must be able to be aware of their current position.

We also suppose the organisers to have spread around the city some "static" network nodes, for instance on public illumination poles, traffic lights, or signals. Such nodes must be equipped with network capabilities similar to those of a Wi-Fi access point. In particular, they must allow nearby mobile devices to connect to all the static nodes within their communication range. There is no upper or lower number or density (devices per square meter) limit to the number of static devices to deploy. The only strict requirement involves both the distance among devices and their communication range: the devices must be placed at a distance and have a communication range large enough such as there is no network segmentation. Besides strict requirements, some other guidelines apply:

- it is preferable to displace static nodes in points where there are people, e.g. on streets and crossings;
- the wider the number of static nodes that can communicate, the more precise will be the results;

- the higher the density of static devices, the more precise will be the results.

The last assumption we make is that devices are able to estimate the time needed for walking towards any of the connected device in normal conditions. This last assumption is again perfectly acceptable considering the current technology, where smart devices are always connected to the Internet and can access public navigation services.

19.2 Distributed crowd steering

Given the structure described above and the mobile smart devices able to communicate with the static nodes, we now outline a possible software solution. The device wanting to be steered publishes a gradient including the information about the wanted destination. The information about destination can be either expressed as physical location (e.g. latitude and longitude) or as a description: the system works in a fashion totally orthogonal to this choice. One or more static devices located near a potential point of interest (POI) receive such gradient, and react becoming sources of a gradient which measures the distances between nodes. This response gradient diffuses on static nodes until it reaches the source, carrying within itself information about the chain of static devices forming the shortest path. The requester will receive a gradient pointing towards the nearest POI, and can navigate step-by-step by reaching in order all the nodes of the path. Although it is not part of this very experiment, the reply could be tuned to be different depending on the request and on the characteristics of the point of interest. For instance, a value representing the "level of matching" between the request and the response could be embedded in the gradient, and used to modify its spatial structure in such a way that POIs more affine with the request are preferred even if farther. Some example of similar usage of gradients is available in [SYD+13].

Now that a basic form of steering is in place, we can improve it by adding contextual information. We assume that static nodes are able to detect the number of people in their surroundings. Again, this is not critical even in a real deployment, and can be achieved in numerous ways with a different level of precision, which spans from just keeping track of which mobile nodes are connected (and, as a consequence, within the communication range) to the usage of dedicated sensors and techniques (e.g. cameras and computer vision). Let's call C the perceived number of mobile devices surrounding a static node. We can alter the spatial shape of the gradient by acting on the function that outputs the actual distance between devices. For instance, adopting the notation in Section 2.3.2, we can use $f(n) = \Gamma_n + d(n) + K \cdot C$ where K is a system parameter. What happens? Basically, areas densely populated with mobile devices will appear as more distant, and consequently the requester will be steered towards alternative routes. The whole system works in a totally emergent and distributed way, with no global knowledge of what's going on, no central control involved, and complete distribution of the computation among its components.

A further refinement of these steering mechanisms involve the ability to react to events that will happen in future (given that there is information about that). The patterns required to do

so have already been presented in Chapter 13, and the interested reader can deepen reading [MPV12].

19.3 Simulation in ALCHEMIST

We decided to focus on the case of mass urban sports events, and in particular on the Vienna City Marathon 2013, an event that every year involves about 40.000 actives and 300.000 spectators. During such event, a smartphone application based on SAPERE [ZCF+11] concepts was deployed, and gathered 1503 high quality GPS traces [APNF13]. We relied on such data set to build simulations demonstrating our concepts of crowd steering at a urban level.

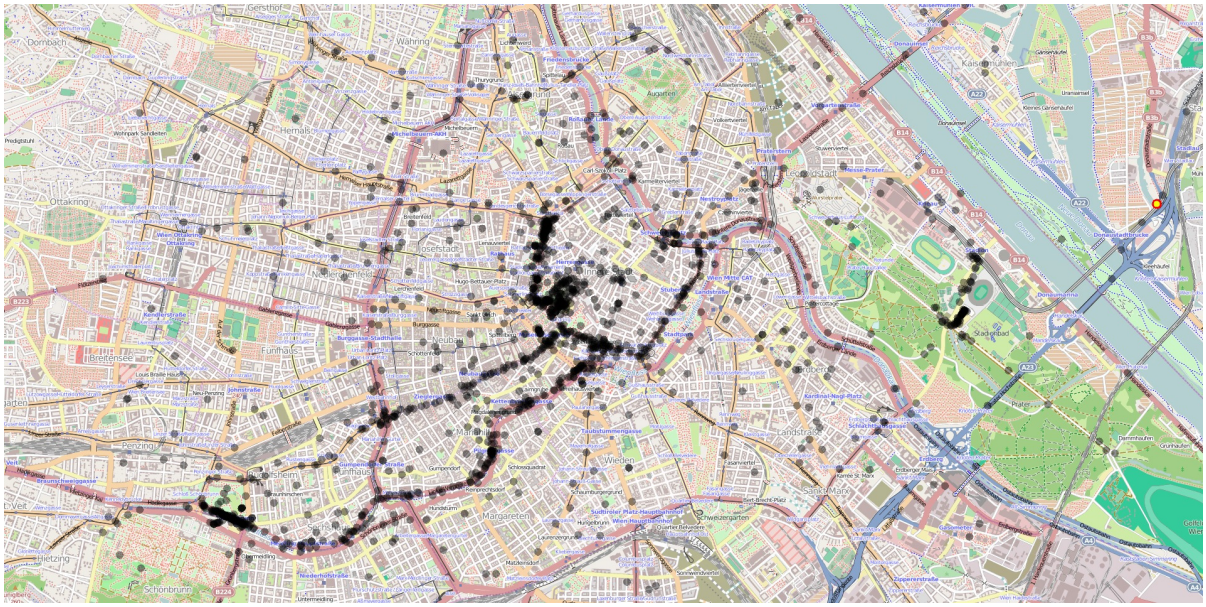
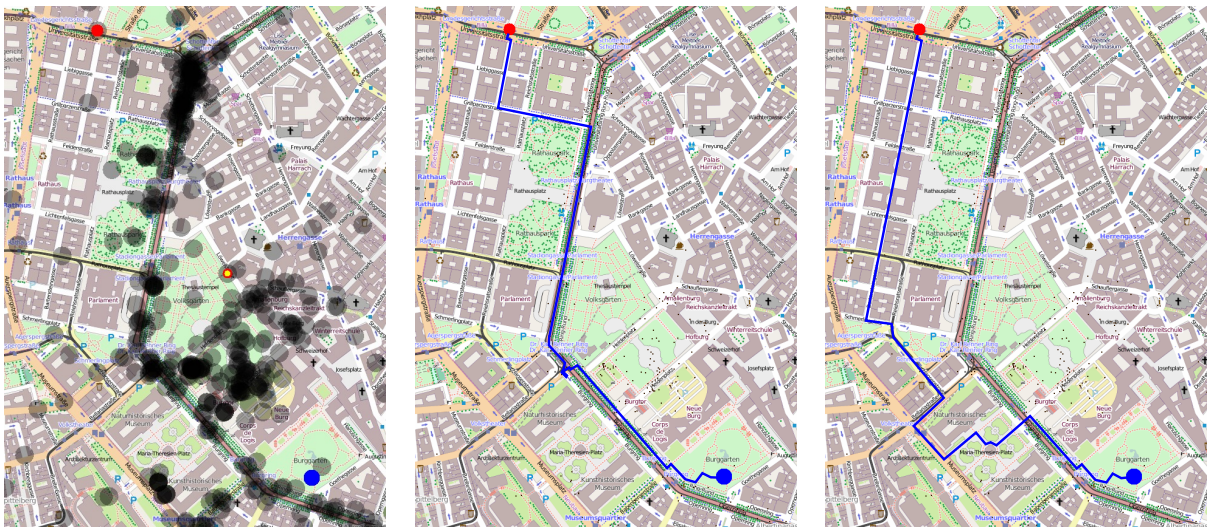


Figure 19.1: A snapshot of the whole city of Vienna as simulated in ALCHEMIST. This snapshot is taken while simulating the city at 10am, each black point corresponds to a GPS trace. The more an area is crowded, the blacker it appears in the image.

As ALCHEMIST environment, we obviously used the map of Vienna. Another rather straightforward choice was to map each device to a node, supposing the users to carry one and one only device which participates to the system. The network of static nodes was built by creating a grid of 572 devices spread around the city and enabling the positioning in the nearest street point. After that, we positioned a node at Burggarten (48.203926,16.365765), representing our POI, and a node at (48.21441,16.35825) representing our navigation system user. Finally, we positioned the nodes of the users which follow the traces. As result, we had in total 2077 simulated nodes, of which 1504 mobile and moving during the actual simulation, 1503 by following the traces using the mixed mode (see Figure 8.3(c)) and one following the steering system.



(a) Level of crowding in the interested area

(b) Classic navigation

(c) Crowd-sensitive navigation

Figure 19.2: In these three snapshots a qualitative evaluation of the crowd-sensitive steering system benefits is offered. The requester is painted in red, while the nearest POI is in blue. In Figure 19.2(a), the density of people in the area is shown (at the time at which the red node starts): a more intense black mean a more crowded area. In Figure 19.2(b) and Figure 19.2(c) is possible to see the suggested routes of, respectively, the classic navigation and the crowd-sensitive navigation. The second suggests a longer but much less jammed path.

As linking rule we decided to let the infrastructure nodes connect to every other node within a communication range of 150 meters. We used the same set of reactions for each node in the system, but on the mobile nodes, in which also the reactions needed to program the mobility were included. In Figure 19.1 a graphical output of the simulator is proposed. We relied on the SAPERE meta-model. This way, we were able to simulate a network of programmable tuple spaces.

A complete evaluation of such an emergent environment is all but trivial. The best way to measure the effectiveness of a crowd-sensitive user steering system would probably be the measure of the average walking time on routes generated probabilistically according to the most walked paths. This kind of evaluation has two strong requirements to prove itself scientifically relevant: first, it requires to identify how popular are the routes; second, and most important, it would require a realistic model of pedestrian, including the physical interaction with other people. At the time of writing, we do not have such model, although we are working to find a decent approximation.

As proof of concept, we chose to monitor the path of a single user steered from Universitätstrasse to Burggarten. We chose this path because the shortest path connecting the user to her destination includes a walk in between one of the most crowded areas during the sport event,

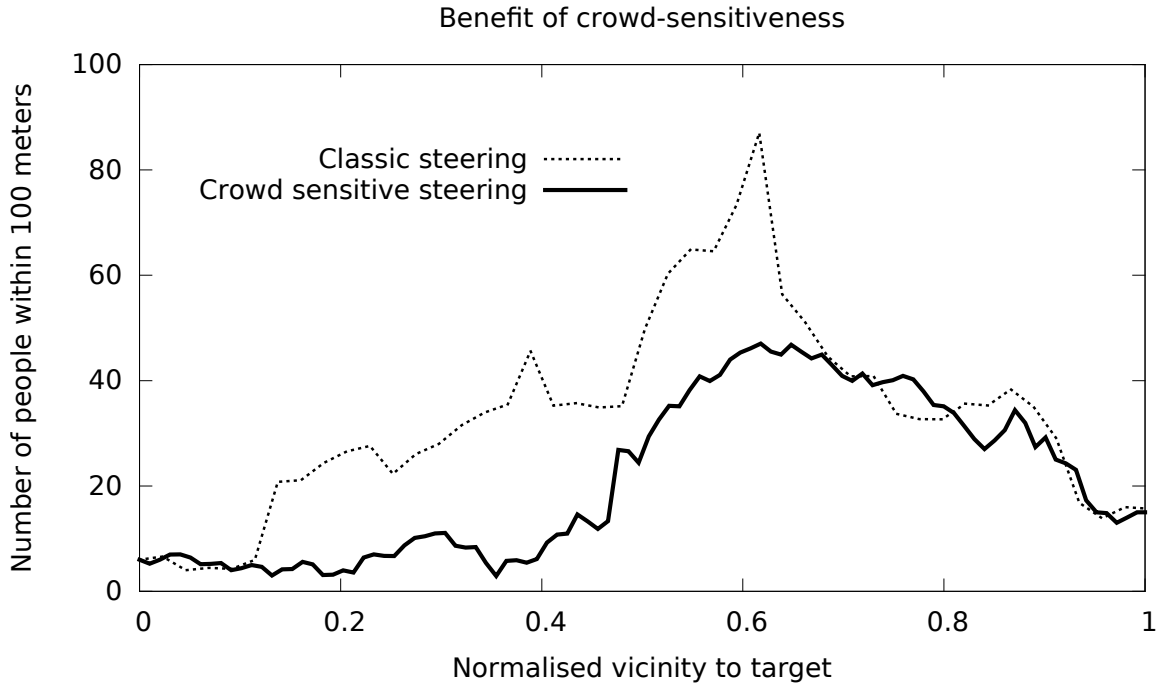


Figure 19.3: This chart shows how the number of users surrounding the user (within 100 meters from her) vary with the proximity to the target. With "normalised distance", we mean that we divided the distance the user still has to walk to reach the target by the total length of the suggested path.

and as such is the perfect test bed for an approach whose goal is to improve the trip time in such conditions.

In Figure 19.2 a qualitative impact is given. It is immediately clear that the suggested path in Figure 19.2(a) is longer but successfully reduces the space walked within crowded areas, with respect to the one in Figure 19.2(b).

The chart in Figure 19.3 confirms the qualitative evaluation. Towards the beginning and the end, the two algorithms have a similar performance, due to the fact that these parts of the walk (as the initial) are common. In the central part of the walk, however, the path suggested by the crowd-sensitive algorithm tends to avoid a considerable amount of jammed areas. We claim that, assuming the traces to be a representative sampling of the actual population of the area and the more crowded areas to be slower to walk (both assumptions sound rather straightforward to the authors), then the crowd sensitive algorithm guarantees an average lower time to destination.

Part IV

Aggregate programming languages

20 Tuple based aggregate programming

A common viewpoint in developing coordination models on top of Linda [Gel85] is that tuples are a mechanism to reify and exchange events/data/knowledge that are important to system coordination, and to synchronise the activities of coordinated components in a parallel and/or distributed system. A tuple is, however, a very point-wise abstraction, and applications often need to express relationships articulated across the physical environment through which the computational system is distributed. There is thus a need for coordination models and languages that raise the level of abstraction from the single tuple to a spatial structure of tuples, without forgetting the possibility that such a structure, as well as being distributed, may also be highly dynamic and mobile.

Based on the works such as [VCMZ11, MZ09, NFP98, PCBB07], and on an existing trend of studies of concurrency "in space" [CG00, CG10, BMS11, BB06], we introduce a new coordination model and language aiming at further bridging the gap between coordination and the continuum ideas of spatial computing. In our model, we make situated agents interact by injecting into the coordination substrate so-called *space-time activities*, namely, processes that manipulate the space-time configuration of tuples in the network. Such activities are expressed in terms of a process-algebra like language, composing atomic coordination primitives in the style of Linda with additional Proto-derived constructs that deal with spatial and temporal aspects:

1. spreading of an activity in a node's neighbourhood depending on its relative orientation;
2. scheduling an activity at the next computation round of a node;
3. accessing space-time contextual information to link the configuration of tuples with the actual physical space.

20.1 Linda in space-time

20.1.1 Basic model

Our coordination infrastructure runs on a (possibly very dense and mobile) set of situated computational devices, e.g., located in specific points (i.e. nodes) of the physical environment.

Each node hosts software agents (the coordinated components) and a tuple space, and has ability of interaction with nodes in the neighbourhood—where proximity can be seen as a physical or virtual property. Differently from Linda, in which agents interact by atomic actions (inserting, removing and reading tuples in the local tuple space), in $\sigma\tau$-Linda, agents interact by injecting *space-time activities* (activities for short). These activities are processes composing atomic Linda-like actions with additional constructs allowing the activity to diffuse in space (i.e. to other nodes) and in time (i.e. to be delayed). The net effect of an activity is hence to evolve the population of tuples in the network, thereby affecting the distributed structures of tuples that agents use to for global coordination.

In our model, each node undergoes the following *computation round*:

1. it sleeps, remaining frozen;
2. it wakes up, gathers all incoming activities (contained in messages received either from neighbour nodes or from local agents) and executes them;
3. it executes the continuation of the activity executed in previous computation round;
4. it spreads asynchronous messages to neighbourhood;
5. it schedules an activity continuation for next round;
6. then it sleeps again, returning to the beginning of the cycle.

The duration of the computation round is dictated by the underlying infrastructure, and could possibly change over time or from device to device (as in [BBF07]). We only assume that it is long enough for executing steps (2-3-4) above. This particular computational model for a tuple space has many similarities with the platform assumptions of the Proto language [BB06, VBC11].

20.1.2 The coordination language

A key role in the proposed coordination model is played by the concept of space-time activities. We here incrementally describe their features by presenting a surface language for their specification.

Primitive actions

We being with three basic Linda actions for manipulating the tuple space: "`out` *tuple*", "`in` *tuple*", "`rd` *tuple*". These respectively insert, remove and read a tuple from the local tuple space. A tuple is a ground first-order term in our model—similarly to [OZ99]. Read and removal specify a template (a term with variables, denoted as literals starting with an upper-case letter) that should be syntactically matched with the retrieved tuple. Read and removal are predicative: they are non-blocking and yield a negative result if no matching tuple is found. To these, we add a fourth primitive, "`eval` *pred*", which evaluates the predicate expression *pred*.

When such actions are defined for an activity injected by an agent, and are executed in the tuple space where the agent is situated in, then a notification result is shipped to the agent—although, following the spirit of [BGZ97], we shall not discuss internal aspects of agent interactions in this paper.

Protocols

Primitive actions can be sequentially composed in a protocol-like manner. Other standard operators of parallel composition and choice could be orthogonally added, but are not discussed in this paper for brevity. Additionally, `in`, `rd` and `eval` define branches leading to two different continuations, one for positive and one for negative outcome of the predicative action. Three examples of activities are:

```
out t(1,2,3); out t(a,1+2,b)
in r(X,2,3) ? out r(X,2,3) : out r(0,2,3)
(in r(X,2,3) ? (eval X=1 ? out r(X,2,3) : 0) : 0); out ok
```

The first expression inserts tuple `t(1,2,3)` and then `t(a,3,b)`. Note that tuples are evaluated before being used in actions: evaluation amounts to computing the result of (mathematical) expressions used in a tuple's arguments.

The second expression removes any tuple matching `r(X,2,3)` (variable `X` is bound to the value of first argument, and this substitution propagates through the remainder of the activity). If it succeeds (`?` branch) the tuple is inserted back, otherwise (`:` branch) a new tuple `r(0,2,3)` is inserted.

The third example attempts to remove any tuple matching `r(X,2,3)`. If it succeeds and $X = 1$ then it inserts it back, otherwise it does nothing (`0`). Independently of the outcome of such a removal, tuple `ok` is then inserted. We may also omit the denotation of a ":" branch when it leads to the execution of empty process `0`, writing e.g.

```
(in r(X,2,3) ? eval X=1 ? out r(X,2,3)); out ok
```

in place of the third example above.

20.1.3 Definitions

When desired, one can equip the specification of an activity with *definitions* (which can possibly be recursive), in the style of agent definition in π-calculus. These have the form "N `(`x_1`,...,`x_n`)` `is` *activity*", which define *activity* as having name N and arguments $x_1,\ldots,x_n$. For instance, after declaration

```
in-out(T) is (in T; out T)
```

we have for instance that activity

```
in-out(r(1,2,3))
```

behaves just like

```
in r(1,2,3); out r(1,2,3)
```

namely, the tuple is added if it is not already there. Note that since no branches are used for the removal operation, this in turn is equivalent to

```
(in r(1,2,3)?0:0); out r(1,2,3)
```

or, similarly, to

```
in r(1,2,3)?out r(1,2,3):out r(1,2,3)
```

20.1.4 Time

The language provided so far is still point-wise in space and time. We now expand it, beginning by adding construct `next` to situate activities in time. Executing action "`next` *P*" (where *P* is the protocol – also called process – defining an activity), amounts to scheduling *P* for execution at the next computation round of the current node. Special variable `$delay` can be used in *P* and evaluates to the amount of time passed in between the current computation round and the previous one. Similarly, variable `$this` can be used to denote the identifier of the node on which it is evaluated. Useful examples of definitions (along with a brief descriptions of them) are then the following ones:

```
% When a tuple matching T is found,
% it is removed and replaced with T2
chg(T,T2) is (in T ? out T2 : next chg(T,T2))

% Inserts a tuple time(T,X), updated as time X passes
rep(T)  is (out time(T,0); next rep2(T))
rep2(T) is (in time(T,Y); out time(T,Y+$delay); next rep2(T))

% Inserts tuple T that is removed after X time units elapse
outt(T,X) is (in-out(T); eval X<=0
                         ? in T
                         : next outt(T,X-$delay))
```

Note that by the use of `next` in conjunction with a recursive definition, `chg` actually declares an activity with a duration in time, which will be stopped only when a tuple matching `T` is eventually found. Concerning the use of `$delay`, we then observe that – in the spirit of a Proto-style space-time computing model – as the average duration of computation rounds tends to zero, activity `rep` tends to define a continuous update of tuple `time(T,X)` as time `X` passes, and similarly, `outt(T,X)` tends to remove tuple `T` precisely as time `X` passed.

20.1.5 Space

To situate activities in space we introduce construct `neigh`. Executing action "`neigh` *P*" amounts to sending a broadcast message containing *P* to all neighbours, which will then execute *P* at their next computation round. Special variable `$distance` is also introduced, which evaluates to the estimated distance between the node that sent the message and the one that received it. Similarly, variable `$orientation` can be used to denote the relative direction from the receiver to the sender (e.g., as a vector of coordinates [CG10]). Some examples are as follows:

```
% Broadcasts tuple T in the neighbourhood
bcast(T) is (neigh out T)

% Broadcasts tuple T in the neighbourood but only within range R
bcastr(T,R) is (neigh (eval $distance<R ? out T))

% Gossips tuple T in the whole network within range R
goss(T,R) is (eval R>=0 ? (in-out(T); neigh goss(T,R-$distance)))
```

Of particular interest is the last definition, which spreads one copy of `T` to all the nodes whose hop-by-hop distance from the source is smaller than `R`. As in the case of time, as devices become increasingly dense, and their distance tends to zero, the set of devices holding tuple `T` will actually form a continuous sphere with radius `R` around the origin of `goss`.

Note that it is an easy exercise to define processes dealing with both space and time. For instance, one can define a process `gosst` that adds temporal aspects to the `goss` example, such as to make the sphere of tuples created by `goss` all disappear following a timeout. In the sense of spatial computing interpretation [BB06], the definition of `gosst(T,R,TO)` would be the definition of a geometric space-time activity called "sphere of tuple `T` with radius `R` and timeout `TO`"—useful to limit the spatial and temporal extent of some advertised information.

20.1.6 Finally

We conclude by introducing a construct named `finally`, used to simplify the task of structuring the activities executed at a given round. Executing action "`finally` *P*" makes activity *P* executed in the current round, but only when all the others actually completed. A typical use of this construct is to start an aggregation activity for incoming messages only when all of them have been processed, as in the following equivalent specification of gossiping:

```
gossf(T,R) is (eval R>=0 ? (out T; neigh gossf(T,R-$distance);
                            finally clean(T)))
% Cleans multiple copies of T, leaving just one of them
clean(T) is (in T ? (in-out(T); clean(T)))
```

Messages spread by gossiping cause the receiver to execute the `gossf` activity, which inserts tuple `T`, further spreads messages, and finally schedules the `clean(T)` process for a later time. Only when all such messages have been processed in a round (and there are typically more than one) will the set of all `clean` activities be executed. The result of their execution is that only one tuple `T` will remain in the tuple space. The `finally` construct can thus, e.g., serve a similar aggregation and simplification role to the `*-hood` constructs in Proto.

20.2 Core Calculus

In this section we introduce a formalisation of the proposed framework similar in spirit to those of [BGZ97, NFP98, VC09], namely, by a core calculus taking the shape of a process algebra.

20.2.1 Syntax

Let meta-variable σ range over tuple space (or node) identifiers, x over logic variables, τ over real numbers used to model continuous time, and f over function names (each with a given arity, and used either in infix of prefix notation)—as usual we refer to functions with arity 0 as constants. Meta-variable t ranges over terms built applying functions to variables, numbers, identifiers, and constants, and will be written in typetext font. For simplicity, we shorten special variable `$orientation` to ω, and neglect `$distance` since it can be "compiled away" to term $length(\omega)$ where *length* is a function. We let ε range over evaluations (functions) for terms, write t^{ε} for application of ε to term t, and denote $\varepsilon(\sigma,\tau)$ the evaluation that (other than computing mathematical functions) maps `$this` to σ and `$delay` to τ. For instance, we have $\texttt{a(\$this,1+\$delay)}^{\varepsilon(\texttt{id23},5.1)} = \texttt{a(id23,6.1)}$. A substitution θ of variables $x_1,\ldots,x_n$ to terms $t_1,\ldots,t_n$ is expressed by notation $\{t_1/x_1,\ldots,t_n/x_n\}$, and is applied to a term t by syntax $t\theta$, e.g., $a(x,1)\{x/2\}$ means $a(2,1)$. We write $mgs(t,t')$ for the most general substitution θ such that $t'\theta = t$—such a notation makes no sense (as in partial functions) if $mgs(t,t') = \bot$, i.e., when t is not an instance of t'.

Given these premises, the core syntax of the model is expressed by the grammar in Figure 20.1 (a). P defines the syntax of a process (or activity): it includes empty process 0, action prefix, predicative actions with branches, and call of a definition. Note we skipped from this syntax the composition operator ";", which can be basically compiled away once we have action prefix "." and branching "? :"—by straightforward equivalences like $0;P \equiv P$, $(\pi?P:Q);R \equiv \pi?(P;R):(Q;R)$ and $(\alpha.P);R \equiv \alpha.(P;R)$. A space S is a composition, by operator " | ", of processes and tuple sets. The topology of a network is modelled by a composition L of connections of kind $\sigma \overset{t}{\rightsquigarrow} \sigma'$, representing proximity of node σ' to σ with orientation vector t—e.g., expressed as term `coord(x,y,z)` or the like. Finally, a system configuration C is a composition, by operator $\otimes$, of nodes $[S]_{\sigma}^{\tau,\tau'}$ (with id σ, space S, current round at time τ and previous one at τ'), topology L, and messages $P \triangleright \sigma$ (with content P and recipient σ).

t	$::=$	$x \mid \sigma \mid \tau \mid f \mid f(t_1,\ldots,t_n)$	Terms
P,Q,R	$::=$	$0 \mid \alpha.P \mid \pi?P:Q \mid D(t_1,\ldots,t_n)$	Process
α	$::=$	$out\ t \mid \Box P$	Action
$\Box$	$::=$	$next \mid neigh \mid finally$	Scheduling operator
π	$::=$	$rd\ t \mid in\ t \mid eval\ t$	Predicative action
T	$::=$	$0 \mid t \mid (T \mid T)$	Tuple set
S	$::=$	$0 \mid T \mid P \mid (S \mid S)$	Space
L	$::=$	$0 \mid \sigma \overset{t}{\rightsquigarrow} \sigma \mid (L \mid L)$	Topology
C,D	$::=$	$0 \mid [S]_\sigma^{\tau,\tau'} \mid P \triangleright \sigma \mid L \mid (C \otimes C)$	Configuration
λ	$::=$	$\cdot \mid \sigma!P \mid \sigma\tau?P \mid P \triangleright \sigma \mid L:L$	Labels

" $\mid$ " and "$\otimes$" are commutative, associative, and absorb 0

$$(\Box P).Q \equiv Q \mid \Box P \qquad \Box(P \mid Q) \equiv (\Box P) \mid (\Box Q) \qquad \Box 0 \equiv 0$$

$$finally\ P \mid next\ Q \equiv finally\ (P \mid next\ Q)$$

(STR) $$\frac{C \equiv C' \qquad C' \xrightarrow{\lambda} D' \qquad D' \equiv D}{C \xrightarrow{\lambda} D}$$

(SND) $$\frac{C \xrightarrow{\sigma!P} C'}{(\sigma \overset{t}{\rightsquigarrow} \sigma') \otimes C \xrightarrow{\sigma!P} C' \otimes (P\{t/\omega\} \triangleright \sigma') \otimes (\sigma \overset{t}{\rightsquigarrow} \sigma')}$$

(BRO) $$\frac{(\sigma \overset{t}{\rightsquigarrow} \sigma') \notin C \qquad P \not\equiv 0}{[S|neigh\ P]_\sigma^{\tau,\tau'} \otimes C \xrightarrow{\sigma!P} C \otimes [S]_\sigma^{\tau,\tau'}}$$

(REC) $$\frac{C \xrightarrow{\sigma\tau?P|Q} C'}{(P \triangleright \sigma) \otimes C \xrightarrow{\sigma\tau?Q} C'}$$

(NEW) $$\frac{P \triangleright \sigma \notin C \qquad \tau_2 > \tau_1}{[T|next\ Q]_\sigma^{\tau_1,\tau_0} \otimes C \xrightarrow{\sigma\tau_2?P} C \otimes [T|P|finally\ Q]_\sigma^{\tau_2,\tau_1}}$$

(FIN) $$\frac{-}{[T|finally\ P]_\sigma^{\tau,\tau'} \otimes C \dot{\rightarrow} C \otimes [T|P]_\sigma^{\tau,\tau'}}$$

(RUN) $$\frac{S\langle P\rangle \xrightarrow{\varepsilon(\sigma,\tau-\tau')} S'\langle P'\rangle}{[S|P]_\sigma^{\tau,\tau'} \otimes C \dot{\rightarrow} C \otimes [S'|P']_\sigma^{\tau,\tau'}}$$

(MOV) $$\frac{-}{L \otimes C \xrightarrow{L:L'} C \otimes L'}$$ (AGN) $$\frac{-}{C \xrightarrow{P\triangleright\sigma} C \otimes (P \triangleright \sigma)}$$

(OUT)	$S\langle out\ t.P\rangle$	$\overset{\varepsilon}{\hookrightarrow}$	$(S \mid t^\varepsilon)\langle P\rangle$	
(IN1)	$(S \mid t')\langle in\ t?P:Q\rangle$	$\overset{\varepsilon}{\hookrightarrow}$	$S\langle P\theta\rangle$	if $\theta = mgs(t',t^\varepsilon)$
(IN2)	$S\langle in\ t?P:Q\rangle$	$\overset{\varepsilon}{\hookrightarrow}$	$S\langle Q\rangle$	if $\nexists t' \in S$ and $mgs(t',t) \neq \bot$
(RD1)	$(S \mid t')\langle rd\ t?P:Q\rangle$	$\overset{\varepsilon}{\hookrightarrow}$	$(S \mid t')\langle P\theta\rangle$	if $\theta = mgs(t',t^\varepsilon)$
(RD2)	$S\langle rd\ t?P:Q\rangle$	$\overset{\varepsilon}{\hookrightarrow}$	$S\langle Q\rangle$	if $\nexists t' \in S$ and $mgs(t',t) \neq \bot$
(EV1)	$S\langle eval\ t?P:Q\rangle$	$\overset{\varepsilon}{\hookrightarrow}$	$S\langle P\rangle$	if $t^\varepsilon = \texttt{true}$
(EV2)	$S\langle eval\ t?P:Q\rangle$	$\overset{\varepsilon}{\hookrightarrow}$	$S\langle Q\rangle$	if $t^\varepsilon \neq \texttt{true}$
(D)	$S\langle D(t_1,\ldots,t_n)\rangle$	$\overset{\varepsilon}{\hookrightarrow}$	$S\langle P\{t_1^\varepsilon/x_1,\ldots,t_n^\varepsilon/x_n\}\rangle$	if $D(x_1,\ldots,x_n)$ `is` P

Figure 20.1: (a) Grammar, (b) Congruence, (c) Global semantics and (d) Local semantics

Figure 20.1 (b) introduces a congruence relation "$\equiv$", stating when two configurations are to be considered syntactically equal, and hence can be used one in place of the other. First line introduces standard multiset-like properties of operators " $|$ " and "$\otimes$". Second line states that scheduling operators can be lifted out of action prefix placed in parallel with the continuation, and can also distribute in parallel processes. Last line states that when a `finally` and `next` actions are in parallel composition, the latter can enter the former: this will in fact leave scheduling policy for Q unchanged.

20.2.2 Operational Semantics

We define operational semantics by transitions $C \xrightarrow{\lambda} C'$, where labels λ can have the syntax described in Figure 20.1 (a). Label "$\cdot$" means a silent action internal to a node σ; "$\sigma!P$" means device σ is broadcasting a message with content P; "$\sigma\tau?P$" means device σ starts a new computation round at (its local) time τ and still needs to gather messages with content P (at the top level it will take the form $\sigma\tau?0$); "$P \triangleright \sigma$" means an agent is injecting process P in the tuple space σ; and "$L : L'$" means (sub)topology L changes to L' to reflect some mobility or failure in the system. Semantic rules are shown in Figure 20.1 (c).

Rule (STR) defines classical structural congruence. Rules (BRO) and (SND) recursively handle broadcasting (mostly in line with [SRS10]), namely, create messages for all neighbours as soon as a process P is scheduled for broadcasting. Rule (SND) recursively selects a neighbour σ' at orientation t, and creates a message for it in which orientation variable ω is substituted with t. Rule (BRO) is the fixpoint: when all neighbours have been handled, scheduling action *neigh P* is removed. Note we do not send empty messages.

Similarly, rules (REC) and (NEW) recursively handle the reception of all messages when a new computation round starts. Rule (NEW) states that, given node σ in which Q is the process to execute at the next round, when a new round starts at time τ_2 and with overall incoming messages P, then the new process to start with is "$P|$*finally* Q", since we prescribe messages to be handled before Q as already described in previous section. Also note that this rule updates round times τ_1, τ_0 to τ_2, τ_1, and that it activates only when all incoming messages have been actually handled. Rule (REC) recursively gathers all incoming messages: it takes one with content P and proceeds recursively adding P to the set Q of messages considered so far.

Rule (FIN) handles semantics of *finally P* construct, by simply stating that when this is the only activity in a node, we can simply execute P—note all "finally-scheduled" processes can be gathered together (along with "next-scheduled" ones) because of congruence. Rule (RUN) handles one-step execution of a process, by simply deferring the task to transition relation $\overset{\varepsilon}{\hookrightarrow}$, defined in Figure 20.1 (d)—its rules are quite straightforward, as they correspond to the standard semantics of Linda primitives in their predicative version [BGZ97]. Note that $\overset{\varepsilon}{\hookrightarrow}$ takes the evaluation function to use, initialised in rule (RUN) with the proper value of `$this` and `$delay`. Finally, rule (MOV) addresses topological changes due to mobility or failures, and rule (AGN) models the injection of a process by an agent in the local node.

We conclude stating isolation and progress properties. First property allows one to reason about the execution of an activity into a node without considering its environment. Namely, we have that nodes get affected by the external environment only at the time a new computation round starts (because of reception of messages), otherwise they proceed in isolation possibly just spawning new messages.

Property 1. *If* $C \otimes [S]_\sigma^{\tau_0,\tau_0'} \xrightarrow{\lambda} C' \otimes [S']_\sigma^{\tau,\tau'}$ *with* $S \not\equiv S'$ *then* λ *is either* $\cdot$, $\sigma!P$, *or* $\sigma\tau?P$. *In the former two cases (namely, unless we change computation round),* $\tau_0 = \tau$, $\tau_0' = \tau'$, *and* $C' \equiv C \otimes C_m$ *(where* C_m *is either* 0 *or a broadcast), and moreover, for each* D *we have also* $D \otimes [S]_\sigma^{\tau_0,\tau_0'} \xrightarrow{\lambda} D \otimes [S']_\sigma^{\tau_0,\tau_0'} \otimes C_m$, *i.e., computation is independent of the environment.*

The progress property states instead that when a computation round is completed it is necessarily composed of a `next` scheduling: at that point (NEW) can surely fire for that node, starting a new computation round. This ensures that our computations never get stuck.

Property 2. $C \otimes [S]_\sigma^{\tau,\tau'} \not\dashrightarrow$ *and* $C \otimes [S]_\sigma^{\tau,\tau'} \overset{\sigma!P}{\dashrightarrow}$ *iff* $S \equiv (T \mid \mathit{next}\ P)$. *In that case, we have* $C \otimes [S]_\sigma^{\tau,\tau'} \xrightarrow{\sigma\tau_0?0} C' \otimes [S']_\sigma^{\tau_0,\tau}$ *for any* $\tau_0 > \tau$.

20.3 Case studies

20.3.1 Adaptive Crowd Steering

As a first example we study a specification able to support the case study presented in [VPMS12, PMV11], with the goal of showing how $\sigma\tau$-Linda can provide support to easily define complex, distributed and adaptive data structures, and how they can be used in practice in a pervasive computing scenario.

Our reference environment is a bidimensional continuous space made of various rooms connected by strict corridors. Inside rooms and corridors, a dense grid of computational devices (nodes) is set up. Each node hosts its own tuple space, receives coordination activities (programmed using our spatial language) by software agents running in it, interacts with nodes in its proximity, and has a sensor locally injecting a tuple `crowd(CrowdLevel)` where `CrowdLevel` is an estimation of the number of people sensed around. People want to reach a point of interest (POI) by the fastest path, and receives directions suggested by their handheld device and/or by public displays on the walls. It is worth noting that the fastest path does not correspond to the shortest: if everybody followed the same way, in fact, corridors would become crowded. We want the system to be able, relying only on local interactions, to avoid crowded paths, dynamically adapting to any emerging and unforeseen situation. However, we will not implement algorithms to predict future situations, but rather make information about a crowded area spread around such that it becomes a less attractive transiting place to reach a POI.

Our strategy is to build a computational gradient injected by an agent located in the POI. A computational gradient holds in any node the estimated distance to the source by the shortest

Crowd-aware gradient

```
% creating a gradient spreading tuple T
source(T) is (in-out(source(T)); grad(T,0,$this))

% gradient process for tuple T, at distance D,
% coming from node S
grad(T,D,S) is grad(T,D,S,$this)
grad(T,D,S,This) is (
  rd source(T)
    ? in-out(pre(T,0))
    : in pre(T,N) ? (eval N<D
      ? out pre(T,N)
      : (in target(T,M); out target(T,S); out pre(T,D)))
); finally (in pre(T,N)
  ? (in field(T,M);
    (rd crowd(C)
      ? out field(T,N-1.2*C)
      : out field(T,N));
        rd field(T,V);
        neigh grad(T,V+$distance,This)))
```

Figure 20.2: Definitions for the crowd-aware computational gradient. At each site, if this is the source we consolidate `pre(T,0)`. Otherwise, we replace the `pre` tuple if a smaller distance `D` is found, and `target` tuple is inserted as well. Finally, we take the remaining `pre` tuple, and apply the `crowd` factor: the resulting distance `N` goes into the `field` tuple.

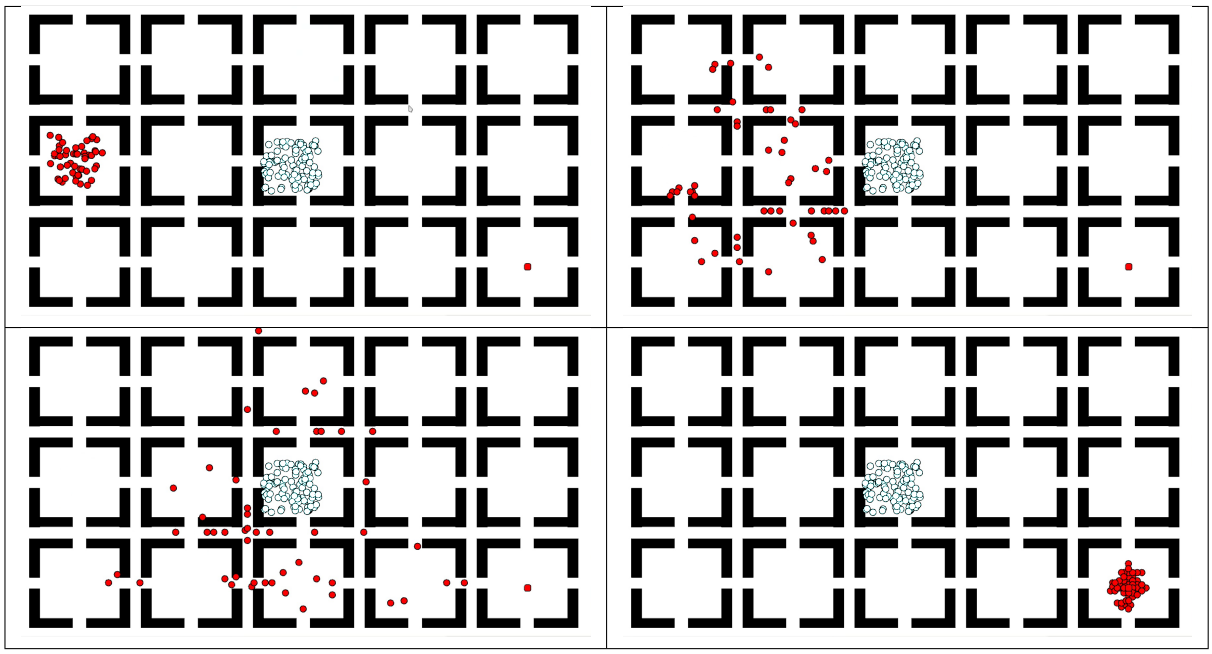


Figure 20.3: Simulation snapshots: the coloured visitors reach its POI avoiding crowd

path [BB06], computed by further spreading and then aggregating at the destination the local estimation of distance. This distributed data structure must take into account also the crowding level, increasing estimated distance where a crowd is forming, and thus deflecting people towards longer but less crowded paths. This strategy can be encoded as in Figure 20.2, where the goal is achieved by maintaining a tuple `target(Poi,Id)` containing the `Id` of the neighbour node where to steer people to following a certain `Poi`. The crowding level influences the local field generation, and is weighted using a constant $K_{crowd} = 1.2$. Values between 1 and 1.5 have been established as good ones after running several simulations: more generally, the higher K_{crowd}, the more sensitive is path computation to the presence of crowd.

We implemented and ran simulations using Alchemist simulator [PMV11], assuming that computation rounds are fired at the same rate for all nodes, and modelling such a rate following the Continuous-Time Markov Chain model. Four screenshots of a simulation run are provided in Figure 20.3, in which we built an environment of fifteen rooms with an underlying grid-like network of infrastructure nodes, an initial configuration with two groups of people, and a POI of interest for the first group which is reachable by a path crossing a crowded area. Note that not only every visitor reached the POI, but they all bypassed the crowded room (even if it is part of the shortest path, the large amount of people inside makes the whole area rather disadvantageous to walk); additionally, the visitors group is subject to "self-crowding", in that when a group is following a path it forms crowded areas itself (e.g. near doors), hence people behind the group tend to follow different paths. Further simulations we do not describe here for the sake of space show that the above properties hold for a large set of situations, including presence of crowds in

different locations and dynamic formation and movement of such crowds during simulation[1].

20.3.2 Linda in a mobile ad-hoc environment

As a second case study we show a possible extension for Linda standard primitives taking into account both time and spatiality. In particular, our aim is to show how would it be possible in a mobile ad-hoc environment to specify, along with an operation over a tuple, a spatial and temporal horizon of validity: only retrieval operations whose horizon embraces the respective target tuple will actually succeed. We will show an implementation for the spatio-temporal `out` (`stout`) and the spatio-temporal `in` (`stin`) primitives—the easier case of `strd` being a simple variant of `stin`.

The key idea is to make primitive Linda actions actually generate waveform-like space-time data structures, with limited extent in space and dissolving as a timeout expires. Those structures will be responsible to determine the pertinence in space and time of each operation. An example of such a structure is realised by the code shown in Figure 20.4 (top). A `wave` works similarly to the gradient in Figure 20.2, maintaining a `target` tuple reifying the shortest past through a similar specification. A main difference – other than the obvious absence of any crowd management – is the evaluation of the age and distance, which makes the wave disappear whenever and wherever the horizon is reached.

When a `stin` operation requiring retrieval of a tuple template `T` is triggered, it will spawn a messenger activity called `hermes` (with `Op` set to `in`) which will propagate to a matching tuple `T'` following the corresponding `wave` it generated. As soon as the tuple is found, a new `hermes` (with `Op` set to `in_back`) is spawned which will follow the `stin` gradient back. This behaviour can be coded as shown in Figure 20.4 (middle).

Given these two basic bricks, the `stout` and `stin` primitives would be encoded as in Figure 20.4 (bottom). For each, a tuple template, a spatial range and a validity time must be specified. `stout` implementation is concise, because it just needs to manifest itself trough a `wave` and make the tuple available; `stin`, instead, needs also to spawn a `hermes`, whose goal is to retrieve a tuple and move it to the tuple space where the operation was spawned.

These new primitives allow agents to publish/retrieve information flexibly tuning the space-time horizons, relying on lower-level gradients (and routing paths) which adapt to the mobility of the network [Bea09].

[1]The interested reader can download an example clip at: `http://apice.unibo.it/xwiki/bin/download/Publications/Coord2012/museum-small.avi`

Wave-form: a space-time gradient

```
wave(T,Range,Ttl) is wawe(T,Range,0,$this,$this,Ttl, 0)
wave(T,Range,D,Source,Ttl,Age) is
  wave(T,Range,D,Source,$this,Ttl,Age)
wave(T, Range, D, Dest, This, Ttl, Age) is (
  eval (Age>Ttl or D>Range)
    ? (in pre(T,D); in field(T,N))              % disappearing
    : rd source(T)
      ? (in-out(pre(T,0))      % default behaviour in a source
      : in pre(T,N) ? (eval N<D    % choosing minimum distance
        ? out pre(T,N)
        : ( in target(T,_);
          out target(T,Dest);
          out pre(T,D))))
); finally (in pre(T,N)? (              % consolidating target
  in field(T,M);
  out field(T,N);
  rd target(T, Dest);
  next wave(T,Range,D,Dest,This,Ttl,$delay);
  eval Age = 0 ?
    neigh wave(T,Range,N+$distance,This,Ttl,0)))
```

Tuple retrieval

```
hermes(Op, T, This) is
  eval This = $this
    ? (eval Op = in ? ( in T
        ? (rd target(op_in(T), Dest);
          neigh hermes(in_back, T, Dest))
        : (rd target(op_out(T), Dest);
          neigh hermes(in, T, Dest))))
    : (eval Op = in_back ? (in in_request(T)
      ? out(T)
      : (rd target(op_in(T), Dest);
      neigh hermes(in_back, T, Dest))))
```

Space-time Linda operations

```
stout(T,Range,Ttl) is out(T);
  wave(op_out(T), Range, Ttl)

stin(T, Range, Ttl) is out in_request(T);
  wave(op_in(T), Range, Ttl);
  hermes(in, T, $this)
```

Figure 20.4: Definitions of $\sigma\tau$Linda operations.

21

Scale independent computations

Spatially, pervasive networks will tend to be highly variable in their density across space and time, and this represents a key problem. For example, in a smart mobility scenario we can expect that certain areas (e.g., city centers) will host a high number of cars and a very dense deployment of traffic sensors and digital signs, while others (e.g., highways or country roads) will be very sparse. Likewise, the density of devices is likely to vary greatly in time, both as increasing numbers of devices are deployed and as events (e.g., sporting events, festivals) cause large fluctuations in the number of people and amount of accompanying devices in particular areas.

We contend that the development and maintenance of distributed systems in the face of such fluctuations will only be tractable if they make use of techniques that ensure "by construction" that computations are inherently resilient against such forms of variability. Toward that end, the first contribution of this chapter (as an extension of the work in [BVD14]), is the identification of a notion of *consistency* of a space-time computation, stating that the outcome of a computation converges as the density of computing devices increases toward infinity. This is further refined to *eventual consistency*, which focuses only on the stable outcome of a computation (e.g., self-stabilizing patterns as studied in [VD14]).

Second, we identify a set of sufficient conditions for this property to hold, expressed as constructs defining a general purpose spatial language [BDU$^+$13] useful for a wide variety of applications. The language is based on the notation of *computational field*, and it is defined as a restriction of the universal *Field calculus* introduced in Section 2.5.2 to a subset of expressions that generate only eventually consistent fields, and features a new construct, "gradient-following path integral" (`GPI`), that generalizes a number of frequently used self-organization patterns. We show how `GPI` can be used to implement well-known self-organization patterns such as distance calculation, broadcast, and path forecasting [MVFM$^+$13, VD14], (which are hence also proven to be eventually consistent). Finally, these patterns are applied in several situated network scenarios, and simulations confirm the predictions of eventual consistency.

Symbol	Definition
$\bar{x}$	Closure of a set; sequence of a token
M	A Riemannian space-time manifold
m	A point ("event") in a manifold M
c	Bound on the speed of information propagation
$T^+(m)$	Time-like future: events reachable from m slower than c
$T^-(m)$	Time-like history: events that can reach m slower than c
d	Computational device: a time-like curve in a manifold M
S_M	A space-like cross-section of a manifold M
D	Discrete subset of a manifold, e.g., events in execution of an algorithm on physical devices.
$\mathbb{V}$	Set of all possible data values
f	Field: a function $f : M \to \mathbb{V}$ assigning values to all points in the manifold M.
$\mathbb{F}_\mathbb{V}$	Space of all fields with range contained in $\mathbb{V}$
e	"Evaluation environment" input field
C	Computation: a higher-order function mapping fields to fields: $C : \mathbb{F}_\mathbb{V} \to \mathbb{F}_\mathbb{V}$
$\mathcal{N}(m)$	Communication neighborhood of m
r	Range within which all points are neighbors
$I^\infty(m)$	Intersection $\mathcal{N}(m) \cap \overline{T^-(m)}$, from which information is directly available to m

Figure 21.1: Table summarizing key symbols that will be used in the following discussion

21.1 Space-Time Computation and Discrete Approximation

This section presents a model of computation over continuous space-time, per [Bea10, BVD14, BUB13], and defines properties linking it with approximate implementation on a wireless network. The continuous abstraction allows specification of a physically situated computation without consideration of the particulars of the network that will be used to implement it. This in turn provides certain types of scalability and resilience: any computation that is *eventually consistent* (as defined below) produces results that are not sensitive to the precise locations of devices and can only improve (asymptotically) as the number of devices in the network increases. In addition, this abstraction supports higher-level specification of distributed algorithms, since many are natural to describe in geometric terms, e.g., distances, regions, information flow.

As many readers may be unfamiliar with prior results in this area, we begin with a review of the manifold model of space-time adapted from physics in [Bea05, Bea10], which then forms a basis for space-time computation, per [Bea10, BVD14, BUB13]. We then use these concepts to define *eventually consistent* programs and examine the utility of this definition.

For additional assistance to the reader, we provide a table of the symbols (Figure 21.1), and a chart of the relationship between definitions (Figure 21.2).

21.1.1 Modelling Networks as Space-Time Manifolds

Many physically situated networks perform computations that can be naturally described in terms of the physical space through which the devices comprising the network are distributed. As observed in [Bea05] and [Bea10], such a network can be viewed as a discrete approximation of the continuous physical space and programmed accordingly.

Under the continuous model developed in those papers, computation takes place on a Riemannian manifold M with both space and time dimensions. A manifold is a space that is locally Euclidean, but may have more complex structure over a longer range. Riemannian manifolds also support familiar geometric constructs like angles, lengths, curvature, integrals and derivatives. This allows communication and mobility constraints to be embedded in manifold's geometry, measuring distance through the manifold rather than using absolute (e.g., latitude/longitude) coordinates. For example, the walkable spaces of a building (e.g., Figure 21.3) form a Riemannian manifold in which the shortest distance between locations goes through doors and hallways, rather than through the walls.

Communication is modelled as a bound c on the speed at which information can propagate; given this bound, concepts and terminology from physics well describe the space-time relations of a manifold [TW92]. A point $m \in M$ is termed an "event," denoting its interest as both a spatial and temporal location (see examples in Figure 21.4), and the manifold may be partitioned with respect to m in space and time:

- Events that information can go to or from exactly at c are *simultaneous* with m.
- Events that can be reached from m moving slower than c are in its *time-like future* $T^+(m)$.

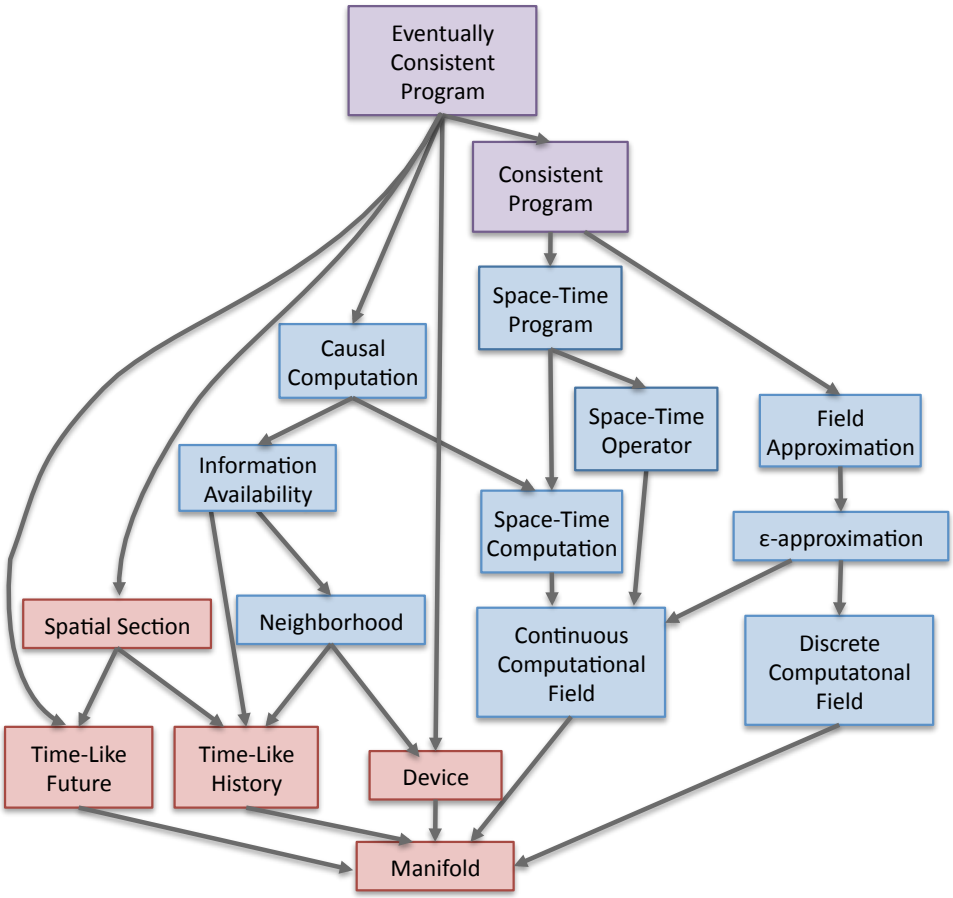


Figure 21.2: The definition of *eventually consistent* programs depends on prior work on continuous computation (blue) and space-time manifolds (red).

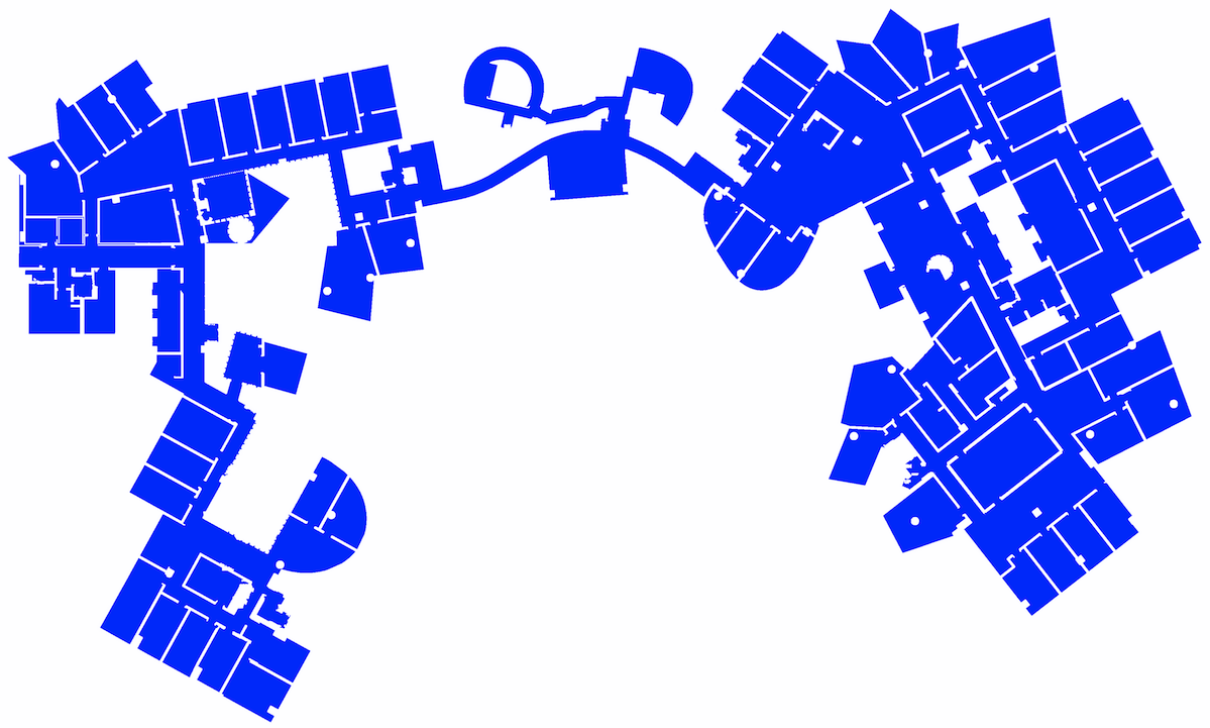


Figure 21.3: The spaces where situated computations take place are often quite complicated, e.g. the MIT Stata Center floor plan shown above. With a manifold representation, distance and other geometric relations conform to the space, e.g., shortest paths follow hallways rather than passing through walls.

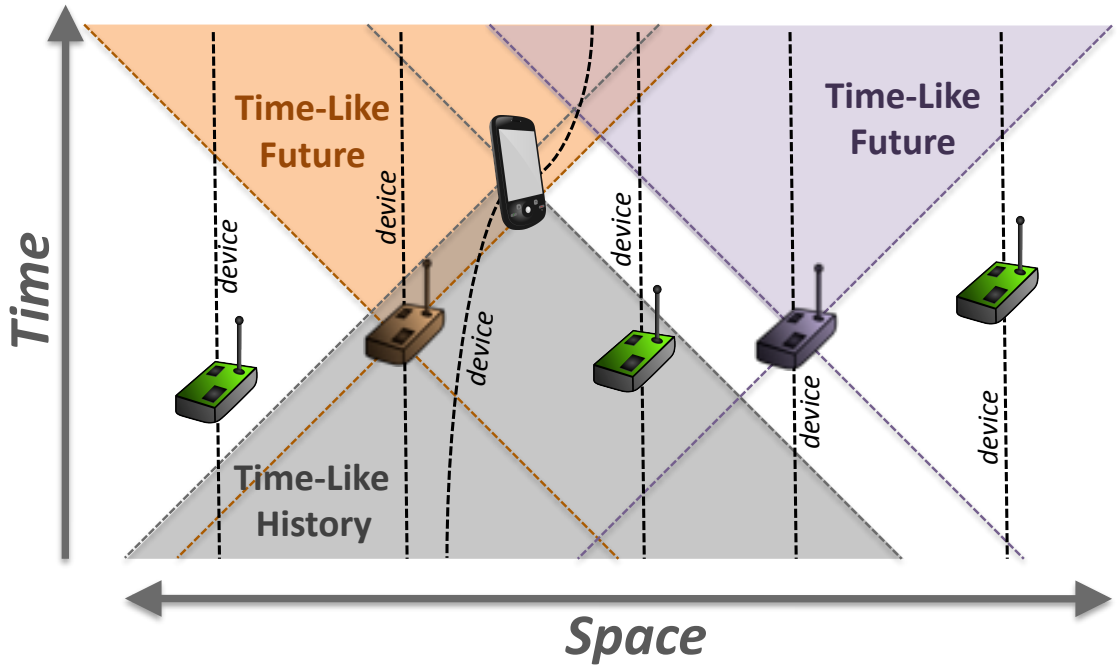


Figure 21.4: Example of space-time relations on a manifold representing a network of wireless devices. Sensor nodes are stationary, while the phone moves along a time-like trajectory (i.e., slope less than diagonal).

- Events whose information reach m moving slower than c are in its *time-like history* $T^-(m)$.

- All other events, which cannot share information because it would need to move faster than c, have *space-like separation* from m and no natural order.

A *device* d can be any one-dimensional curve in the manifold with purely time-like relations between the points[1]. The mathematical analysis in this section also makes the simplifying assumptions that the manifold is finite in diameter and that devices do not move[2]. Finally, a *spatial section* is any subspace S_M whose complement is its time-like future and history, i.e., $M - S_M = T^+(S_M) \cup T^-(S_M)$, implying all points in S_M have purely space-like separation. Spatial sections are essentially snapshots in time, without attempting to enforce a global notion of time.

[1] This is analogous to the physics notion of a world-line.

[2] In practice, the results we develop here often apply to slowly moving devices as well.

21.1.2 Computations on Continuous Space and Time

Typical formulations of distributed computation (e.g., [Lyn96]) cannot be applied to space-time manifolds, as any manifold contains an uncountably infinite number of events, and thus the events of the computation cannot be ordered. Instead, computation may be defined in terms of fields, per [Bea10, BVD14, BUB13]:

Definition 1 (Continuous Computational Field)**.** *A field is a function $f : M \to \mathbb{V}$ that maps every event m in a Riemannian manifold M to some data value in $\mathbb{V}$.*

Discrete computational fields (e.g., events in the execution of a distributed algorithm) may be defined likewise, except that the domain is limited to a discrete subset of events $D \subset M$.[3]

Let $\mathbb{F}_{\mathbb{V}}$ be the space of all fields whose range is contained in $\mathbb{V}$. A space-time computation C is then a higher-order function that maps from each possible field defining an evaluation environment to a corresponding field of values:

Definition 2 (Space-Time Computation)**.** *A computation C is a function $C : \mathbb{F}_{\mathbb{V}} \to \mathbb{F}_{\mathbb{V}}$, where the domain of the output field is always identical to the domain of the input field.*

In other words, a computation takes a field as input, whose domain defines the scope over which the computation executes and whose values are all of the environmental state that can affect its outcome (e.g., sensor readings). At every point of space and time in the execution scope, some output value is produced.

The computations we are interested in are those physically realizable: those that are both approximable, meaning that the output field is bounded in its intricacy, and causal, meaning that information moves subject to the speed bound c. Approximability is determined by comparing values in a continuous field to the nearest points in a corresponding discrete field:

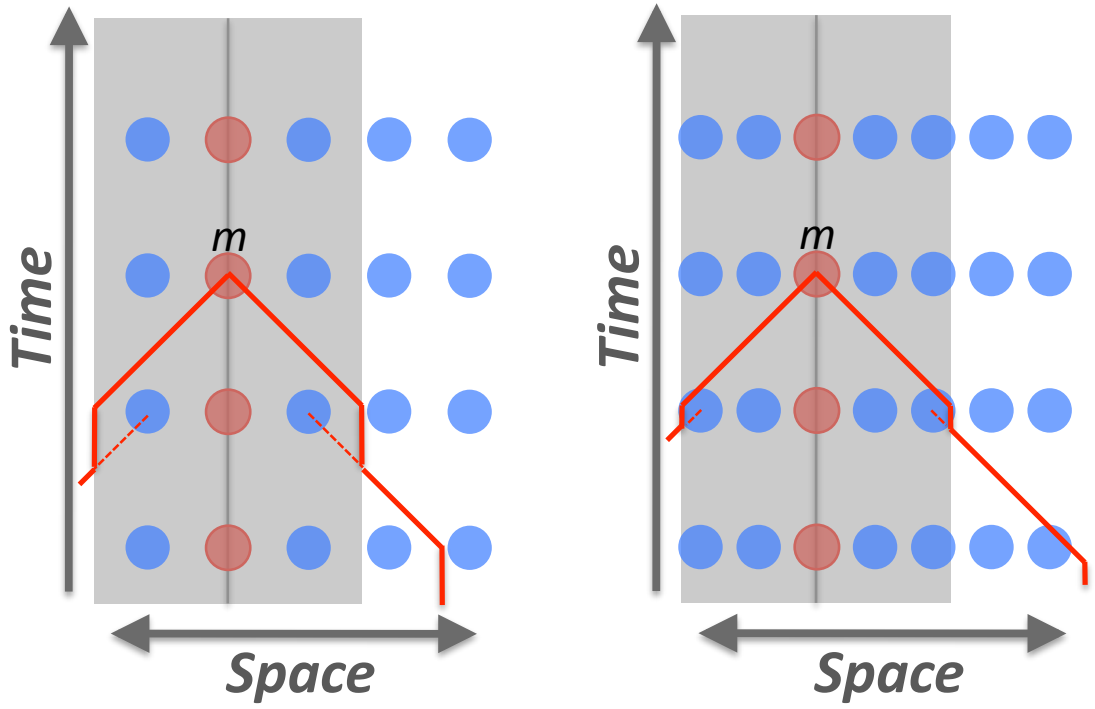
Definition 3 (ε-approximation)**.** *Let $D_\varepsilon \subset M$ be a discrete set such that every event $m \in M$ is within distance ε of some event in D_ε. The ε-approximation of field $f : D_\varepsilon \to \mathbb{V}$ is a field mapping every point in M to the value of f at the nearest point in D_ε (choosing arbitrarily for equidistant points).*

Definition 4 (Field Approximation)**.** *A countable sequence of ε_i-approximations f_i of manifolds M_i, with $\varepsilon_i < \varepsilon_{i-1}$ is said to approximate field $f : M \to \mathbb{V}$ if both the following hold:*

$$\lim_{i\to\infty} |(M \cup M_i) - (M \cap M_i)| = 0$$

$$\lim_{i\to\infty} \int_{M\cap M_i} |f - f_i| = 0$$

[3] Considering a discrete execution as a manifold subspace, rather than an abstract graph, preserves its relationship with the space in which devices are embedded.



(a) Slower Discretization (b) Faster Discretization

Figure 21.5: The effective speed of information flow for a discrete computation depends on the specifics of discretization relative to neighbourhood size. For example, in discretization (a) information moves significantly slower than in discretization (b), despite the neighbourhoods being the same, leading to a smaller information availability (red line) at event m of the red device.

In other words, a sequence of increasingly fine discrete sets approximates a continuous field if both the manifolds and the values assigned over them by the fields tend toward identical. A field produced by a computation is approximable if at least one such approximation sequence can be constructed. Some notes on technicalities.

- The reason to use a sequence of potentially different manifolds M_i is because program branches can create subspaces dynamically, and these necessarily depend on the details of approximation.
- It is generally necessary to evaluate the value difference using a Lebesgue integral [Kes60], since the more typical Riemann integral is ill-defined on many discontinuous fields.

The second condition for physical realizability is causality: causal computations have values

that are determined only by the information available at each event. Intuitively, information is available if it could have been relayed from event to event from its origin. To compute this, we add the concept of a neighbourhood, defining how far information can move in one step:

Definition 5 (Neighbourhood)**.** *Neighbourhoods are defined by a function $\mathcal{N}$ mapping each m to a connected open set $\mathcal{N}(m) \subset M$, containing at least every event within distance r of m (for some positive r).*

Information is thus directly available to an event m in device d from the region $I(m)$ of events that have been in its neighbourhood in the present or past:

$$I(m) = \overline{T^-(m)} \cap \mathcal{N}(d - T^+(m))$$

and the set of all events from which information is available is the transitive closure of direct information availability:

$$I^\infty(m) = I(m) \cup I(I(m)) \cup \ldots$$

For continuous computations, this is precisely equal to the closure of the past (i.e., $I^\infty(m) = \overline{T^-(m)}$). For discrete computations, however, the region depends on the specifics of the discretization, because information cannot be relayed from a device except at its events. For example, Figure 21.5 shows discrete computations with the same neighbourhood but different speeds of information flow due to differences in the position of discrete events within the neighbourhood. The difference between $I^\infty(m)$ and $\overline{T^-(m)}$ is bounded by ε, however, and converges to zero with ε.

Causal computations are then those whose values depend only on events from which information is available:

Definition 6 (Causality)**.** *Computation C is causal if for every pair of evaluation environments e and e' with the same domain M, it is the case that $e(I^\infty(m)) = e'(I^\infty(m))$ implies $C(e)(m) = C(e')(m)$.*

Some examples of causal and approximable computations:

- a computation whose value is always 3;
- a computation that applies the triangle inequality to measure the spatial distance to specific events designated by the environment;
- a gossip computation that spreads knowledge that a particular environmental condition has been sensed.

An example of a non-causal computation is a computation that instantaneously changes from 0 to 1 in response to a sensor value at a single event. An example of a non-approximable computation is one that maps all rational distances from an event to 0 if the numerator is even and 1 is the numerator is odd.

21.1.3 Space-time programs

One way of specifying space-time computations is by the functional composition of a basis set of operators:

Definition 7 (Space-Time Operator)**.** *A space-time operator is a function* $o : \mathbb{F}_{\mathbb{V}} \times \mathbb{F}_{\mathbb{V}}^k \to \mathbb{F}_{\mathbb{V}}$ *taking an evaluation environment and zero or more additional fields as inputs and producing a field as output.*

This is much like the definition of a computation, except that the domains of the fields may differ.

In this work, we consider only causal operators, any functional composition of which must clearly also be causal. Some examples of causal operators:

- + produces a scalar field whose value at each point is the sum of the input values at that point.
- `true` outputs a field mapping the domain of the evaluation environment to the Boolean value *true*.
- `temperature` reads a temperature sensor by extracting its value from the evaluation environment, producing a scalar field.

A *space-time program* is then any functional composition of operator instances to form a computation, such that the domains and ranges of the output are well-defined for all possible values of the inputs [4]. Allowing compositions to be defined as new operators by usual means such as lambda calculus, programs may be recursive as well, with an evaluation structure determined dynamically by their environment. In [BUB13] and [BVD14], space-time programs are formalized as dataflow graphs; here we use an equivalent syntactic formulation and consider only the restricted language developed next.

For clarity in the description of programs, we will abuse terminology; in the context of a program, a "field" is not actually the mathematical object itself, but the input or output of an operator instance, which takes on a field value when evaluated in the context of an environment.

21.1.4 Eventually Consistent Programs

We now apply the continuous model of computation to identify distributed algorithms that are resilient to changes in the number and position of devices. Our approach is as follows: consider

[4] Some notes on technicalities.

- Any well-defined composition of operators is itself an operator.
- Complete programs have no inputs except the environment. Any composition of operators that requires inputs (e.g., function definitions) may, however, be transformed into an equivalent program by "currying" the inputs to instead be supplied by the environment and adjoining a special "no value" value for events in the domain of the environment but not the input.

any distributed algorithm for which we can identify an analogous space-time program. If the distributed algorithm always approximates the space-time program, no matter the particular arrangements of discrete events, then it is clearly resilient to increasing numbers of devices and to small changes in device position:

Definition 8 (Consistent Program)**.** *Let* `P` *be a space-time program, e be an evaluation environment, and* e_i *a countable sequence of* ε*-approximations that approximate field e. Program* `P` *is* consistent *if* $\texttt{P}(e_i)$ *approximates* $\texttt{P}(e_i)$ *for every e and* e_i*.*

As noted in the discussion of causality, however, information may move at different speeds in different discrete approximations. This means any program with non-trivial interaction between devices will not be consistent, but that if a program eventually converges to a steady state, then it may be consistent after that point:

Definition 9 (Eventually Consistent Program)**.** *Consider a causal program* `P` *evaluated on environment e with domain M. Program* `P` *is* eventually consistent *if, for any environment e in which there is a spatial section* S_M *such that the values of e do not change at any device in the time-like future* $T^+(S_M)$*, there is always some spatial section* S'_M *such that* `P` *is consistent on the time-like future* $T^+(S'_M)$*.*

In other words: if the inputs ever converge, then the outputs eventually converge as well, and are consistent thereafter.

As defined, eventual consistency is a conservative property, saying nothing about behaviour before convergence. For many programs, however, the constructs used to ensure eventual consistency often produce more broad resilience.

21.2 Eventually Consistent Language

In this section we develop a language that generates only eventually consistent programs. This language, GPI-calculus, derives both its syntax and semantics from field calculus [VDB13], but is restricted to a set of eventually consistent operators that compose to produce programs that are also guaranteed to be eventually consistent.

21.2.1 From Consistency Failures to Fragility

Consistency is a useful property because it allows us to evaluate when a computation is fragile to the circumstances of its evaluation. Even though real networks are always finite, comparison with a continuous ideal can reveal vulnerabilities in a distributed algorithm through the manner in which it fails to be eventually consistent.

Unbounded recursion, of course, can create consistency failures: infinite loops are obviously ill-defined. There are also many ways to arrange a recursion that grows in depth with density and does not converge. We can eliminate this by the simple but harsh method of prohibiting

recursive function calls, which puts a finite bound on the number of operations that can be used in the evaluation of a program.

Let us turn next to a problem related to distributed computation: interactions between devices frequently lead to scalability problems due to implicit dependence on the distribution of devices. Consider, for example, the field calculus program:

```
(def non-convergent-nbr (v)
  (/ 1 (min-hood (nbr-range))))
```

As devices pack more closely, the distance to the closest device approaches zero, causing the inverse to rise without bound, meaning it does not converge. The program is thus not consistent, indicating its fragility to device distribution.

Likewise, state creates problems in programs that implicitly rely on the time between events:

```
(def non-convergent-rep ()
  (rep x 0 (- 1 x)))
```

This program switches values between 0 and 1 at every evaluation. As the frequency of evaluations rises, it does not settle one or the other, but oscillates faster and faster, failing to converge. It is thus not consistent, indicating its fragility to exactly when and how frequently state updates occur.

Thus, in order to ensure resilience, we will not allow direct access to either state or neighborhood functions, and instead embed them in higher-level constructs with guaranteed convergence.

Eliminating constructs that can directly give rise to consistency failures is not enough however. Even if a program produces approximable fields, the program may not be consistent because it may not be resilient against miniscule changes in its inputs. For example, the `distance-to` program presented in Section 2.5.2 always produces an approximable field of shortest path distances, but is not a consistent program because the value of the output may be greatly affected by individual points in the `source` field. Consider, for example, a `source` field where only one point is *true*: an ε-approximation containing that point has only finite values, while one without that point has infinity everywhere. Thus, it is possible to construct sequences that do not converge, because they alternate between including and not including the critical point.

Are such extreme inputs reasonable sources of concern, however? In fact, it is remarkably easy to produce fields with a set of critical values that has measure zero, i.e., being infinitely thin and therefore not guaranteed to be sampled in any approximation. For example, a bisecting boundary computed by comparing `distance-to` functions

```
(def bisector (point-1 point-2)
  (= (distance-to point-1) (distance-to point-2)))
```

computes a field that is *false* except at an infinitely thin boundary of *true* values. Fed to a program sensitive to such sets, such as `distance-to`, this can result in arbitrarily unpredictable behaviour from a distributed algorithm.

This is not a special case related to `distance-to`, but a deeper conflict for situated distributed algorithms, between the discrete values commonly used in algorithms (e.g., Booleans, branches, state machines) and the continuous space-time environment in which devices are embedded. In particular, any non-trivial field with a discrete range cannot be continuous (proof in Supplementary Information), meaning that it either is itself not approximable or else contains some measure-zero boundary region that, if handled badly, can generate unpredictable behaviour (as in the bisector example).

It is not practical to eliminate every program element that could either generate boundary regions or generate unpredictable behaviour due to values at a boundary. As the bisector example shows, this cannot be done without eliminating at least one of distance measure, comparison, or branching[5], and losing any of those constructs curtails expressiveness more drastically than we are willing to accept. Instead, the approach we present accepts that problematic boundaries will exist, and instead dynamically marks them and contains their effects.

21.2.2 GPI-calculus

Having identified ways in which consistency fails, introducing fragility to distributed algorithms, we now present a new language, GPI-calculus, that restricts the syntax of field calculus and ensures that all programs it can express are eventually consistent. We accomplish this by dynamically marking and tracking potential problems, with the following key changes to field calculus:

- A new value, $\mathscr{B}$, is adjoined to the set of data values, representing a boundary between value regions, and is passed unchanged through all functions.
- Comparing real numbers produces $\mathscr{B}$, rather than a Boolean, when the numbers are precisely equal.
- State and communication are available only through a new compound operator, `GPI`, which can implement a number of distance-based operations.

With these changes, well-written programs will ensure eventual consistency by effectively excluding a miniscule (often empty) set of devices from certain computations. Poorly written programs (e.g., `(= (sqrt 2) (sqrt 2))`) may still contaminate large areas with $\mathscr{B}$ values, but will still converge—just not to a particularly useful result.

Key to this approach is the new operator `GPI`, a "gradient-following path integral," which we define as the field calculus function:

[5]Equality testing can always be implemented by any Boolean-valued comparison of scalars plus Boolean branching.

```
(def GPI (source initial density integrand)
 (if (<= density 0)
  ℬ // Metric ill-defined if density non-positive
  (2nd
   (rep distance-integral
    (tuple infinity initial) // Initial value
    (mux source
     (tuple 0 initial)// Source is distance zero, initial value
     (min-hood' // Minimize lexicographically over non-self nbrs
      (+ (nbr distance-integral)
       (* (nbr-range) // Scalar multiplication of tuple
        (tuple (mean density (nbr density))
         (mean integrand (nbr integrand)))))))))))
```

For this implementation, we use a slightly modified version of the usual field-calculus `min-hood` operator, designated as `min-hood'`, which returns $\mathscr{B}$ if the minimal value for the first tuple element is held by more than one device.

The `GPI` operator thus performs two tasks simultaneously. First, like `distance-to`, `GPI` computes a field of shortest-path distances to a `source` region. Here, however, distance is "stretched" proportional to a scalar field `density` (representing e.g., crowd density slowing movements, hazards increasing danger of movement). Furthermore, the `min-hood'` operator ensures that all points in the distance field with more than one shortest path are $\mathscr{B}$. Second, `GPI` computes a path integral of the scalar field `integrand` following the gradient of the distance field upward away form the source, starting at the scalar value `initial` in the source region. The function definition binds these together via a tuple and lexicographic minimization, such that the value added to the line integral at each device is taken from the neighbor on the (sole) minimal path to the source.

Importantly, the `GPI` operation subsumes a number of useful and frequently used computations. For example, `distance-to` can be computed as:

```
(def distance-to (source)
  (GPI source 0 1 1))
```

which eliminates stretch by setting it to constant 1 and obtains distance by integrating 1 along each shortest path.

The complete syntax of GPI-calculus, shown in Figure 21.6, is then defined as a restriction of field calculus:

- The `rep` and `nbr` constructs are only available indirectly via the `GPI` construct as defined above.

```
l ::= 𝔹 | ℤ | ℝ                                   ;; Literals
b ::= m | mux | <                                ;; local operators
e ::= x | l | (b ē) | (f ē) | (sense ℤ⁺)          ;; expression
    | (if e e e) | (GPI e e e e e)               ;; special constructs
F ::= (def f(x̄) e)                               ;; function
P ::= F̄ e                                        ;; program
```

Figure 21.6: Syntax of GPI-calculus.

- Literals are restricted to only Booleans ($\mathbb{B}$), Integers ($\mathbb{Z}$) and real numbers ($\mathbb{R}$).
- Built-in operators are restricted to the elements `m`, `mux`, `<`, and `sense`.

The available built-in operators are:

- `m` is any strictly continuous mathematical function (e.g., addition, multiplication, logarithm, sine[6]), extended to have output $\mathscr{B}$ if any input is $\mathscr{B}$. Finally, for any *m* that can map integers to integers (e.g., addition), the output has integer type iff all inputs have integer type.
- `mux` is the piecewise multiplexer function: when its first input is *true*, it returns the second input; if it is *false*, it returns the third input; if it is $\mathscr{B}$, it returns $\mathscr{B}$.
- `<` compares two numerical inputs, returning *true* if the first is less and *false* if the second is less. If they are equal, then the result depends on type: if both are integers the result is *false*; if either is a real its $\mathscr{B}$. Finally, if either input is $\mathscr{B}$, then the result is also $\mathscr{B}$.
- (`sense` *k*) returns the *k*th value in the environment state (assumed to be a tuple), where *k* is the positive literal given as its input.

We can interpret the same syntax with both discrete and continuous semantics; our goal will be to show that the discrete is eventually consistent with the continuous for every well-defined program (i.e., one without syntax or semantic errors).

The discrete semantics of GPI-calculus is identical to that of field calculus, with one restriction (function calls may not be recursive) and one trivial modification to the semantics of `if`. In field calculus, the first input to `if`, the test value, must be a Boolean. In GPI-calculus, it may also be $\mathscr{B}$, in which case, neither branch is evaluated, state is erased from both branches, and the result is $\mathscr{B}$. This change, however, is entirely neutral with respect to the interactions of `if` with other elements of the semantics, and so does not affect any of the properties established in [VDB13].

[6]This also includes construction and referencing of tuples, which can be used to implement data structures, as well as higher order functions such as map and reduce. Some simple functions are excluded, however, such as division, which is discontinuous when the denominator is zero.

We may also interpret GPI-calculus with a continuous semantics following the model of [BUB13]. Given an environment domain M, literals are constant-valued fields, the built-in operators act pointwise on fields, function calls and variable references are evaluated through a substitution model, and `if` evaluates its subexpressions on environments with domains restricted to match the subspaces of its test argument. As with the discrete semantics, we prohibit recursion.

Specifying GPI-calculus as this minimal set of operations makes eventual consistency tractable to prove, yet still able to implement a much more general range of operations. For example, `mux` is sufficient to implement all logical operations, e.g.:

```
(def and (a b) (mux a b false))
(def or  (a b) (mux a true b))
(def not (x)   (mux x false true))
```

21.2.3 GPI calculus is Eventually Consistent

We can now prove that GPI calculus accomplishes the goal for which it has been designed: ensuring that any program that can be expressed using it is eventually consistent, meaning that it is resilient against the particulars of how many devices are in the network and how they are distributed in space.

Theorem 1 (Eventual Consistency of GPI-calculus)**.** *GPI-calculus programs are eventually consistent for all environments e that are continuous on* $e^{-1}(\mathbb{V} - \mathscr{B})$.

We approach this proof in three stages (see Chapter A). First, we prove that any finite composition of operators that are eventually consistent and preserve certain continuity properties is also eventually consistent and continuity preserving. We then show that each operator in GPI-calculus is individually at least eventually consistent and continuity preserving. Finally, we show that all GPI-calculus programs are equivalent to finite compositions of operators, which implies that all such programs are eventually consistent.

Thus, if a program is evaluated in a "well-behaved" environment, its results are predictable and resilient to scale and positioning of devices. Having proved this, in the next section we explore the usefulness of GPI-calculus and demonstrate its consistency properties empirically in simulation.

21.3 Example Applications

Having established a calculus for eventually consistent programs, we now aim to demonstrate its expressiveness. In this section we present a number of self-organization patterns that can be described in terms of GPI-calculus, with accompanying application scenarios in current or emerging large-scale situated networks: wireless sensor networks, ad-hoc networks created by mobile smart-devices, and urban-scale ecosystems of devices and services for crowd steering.

Simulations of these scenarios demonstrate both the potential impact of our result and confirm the consistency result and its effect in providing coordination behaviors resilient to changes in network density and scale.

21.3.1 Distance-Based Patterns

Distributed distance calculation is one of the most frequently used building blocks for self-organizing systems [FMDMSM+12]. As we have seen in the previous section, distance fields can be computed via a straightforward application of `GPI`. A second example uses the field of distance estimates as a "carrier" to broadcast by gossiping a value from the source. This can be defined from `GPI` as:

```
(def broadcast (source value)
  (GPI source value 1 0))
```

Here, `GPI` shifts the initial `value` outward by integrating 0 along the path up the distance field, so that no matter how far away, the value is always the same. When the source region has a single value v, this produces a field that stabilizes to v at every device, intuitively a "broadcast." In the more general case where different devices in the source have different values, the resulting field maps each device to the value of `value` in the nearest source device.

Functions `distance-to` and `broadcast` can be functionally composed to create a higher-level `channel` distributed structure useful for tasks such as corridor routing. This version of channel takes two source fields `a` and `b`, and a (typically constant) numerical field `w`, and yields a boolean field holding `true` only in those devices whose distance to `a` and `b` is no more than `w` greater than the shortest path:

```
(def channel (a b w)
  (< (+ (distance-to a) (distance-to b))
     (+ w (broadcast a (distance-to b)))))
```

The `broadcast` expression sends to every device the distance from `b` perceived by devices in `a`. Composition by the built-in functions < and + does the remainder of the job. This example emphasizes how function composition can generate a plethora of spatial structures, e.g., in the spirit of [MPV12].

Another key feature of GPI-calculus is the ability to modulate the behavior of algorithms by restricting their domain with construct `if`. For example, this can be used to create channels that circumvent a given area considered as an obstacle:

```
(def channel-with-obstacle (a b w obstacle)
  (if obstacle false (channel a b w)))
```

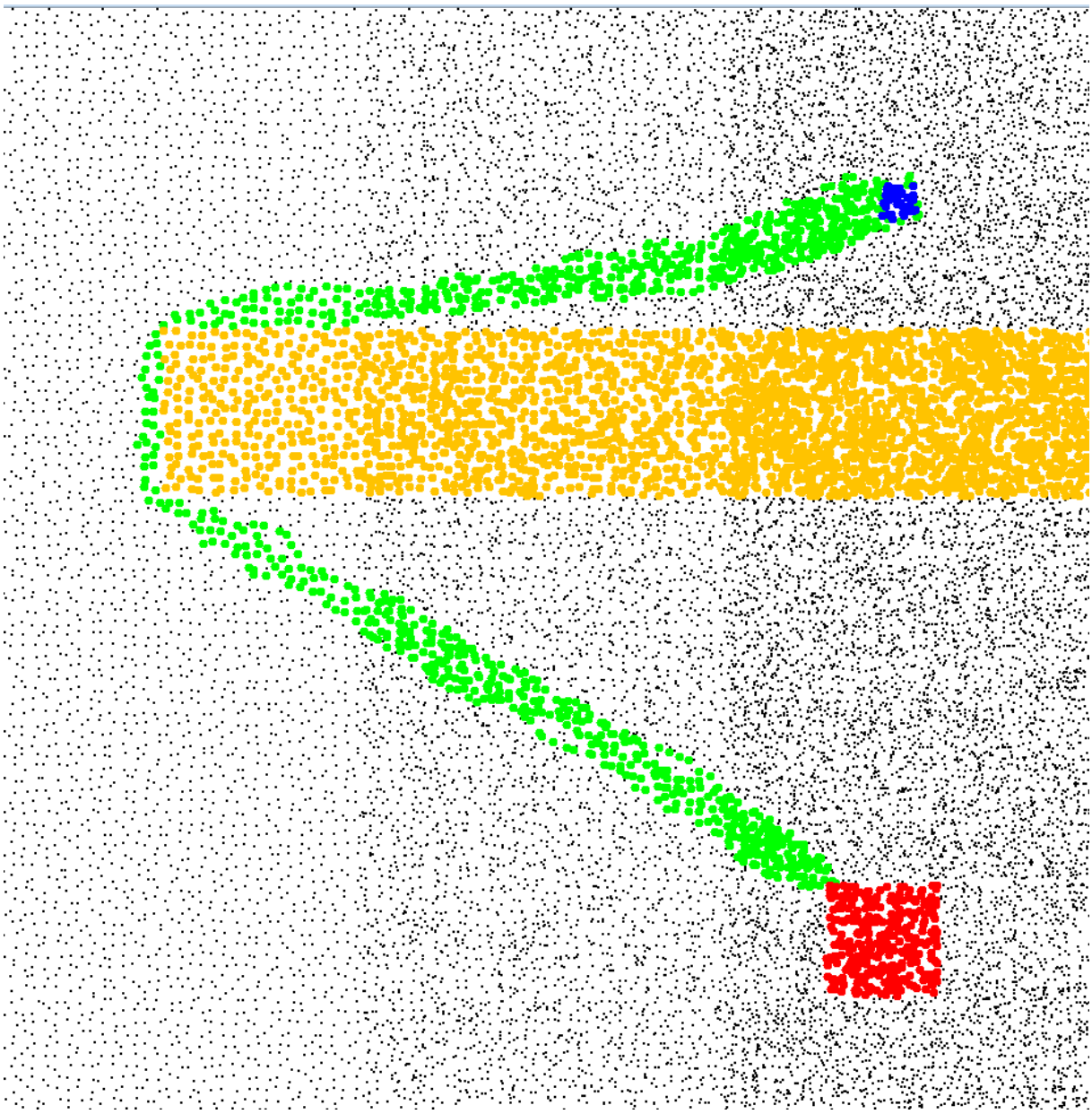


Figure 21.7: Channel pattern deployed on a wireless sensor nework with 20,000 devices deployed at heterogeneous density. Blue devices send data to red along the channel (green), avoiding the obstacle (yellow).

Where `obstacle` is `true`, there is no channel and `false` is returned; in the remainder of the network, we compute a channel in the usual way and manifold geometry routes it around the "missing" space.

As an application scenario, consider a wireless sensor network that is heterogeneous distributed in two dimensions and in which some devices much exchange a large amount of information, e.g., relaying video of an ongoing event to a mobile monitoring station. The goal is to identify a set of devices that can perform the relay, such that:

- the information spreads to only a small fraction of devices (e.g., to save battery energy);
- there is some replication along the transmission path, to reduce the probability of data loss;
- if some devices do not want to collaborate to the transfer (e.g. because their battery level is low, or because they have shown faults), they should be circumvented.

The `channel-with-obstacle` pattern provides one possible implementation, marking only a set of devices near the shortest path excluding non-participating devices. A `broadcast` restricted to this channel with `if` can then effectively relay the data with enough replication to prevent data loss, but with much less resource consumption than an unrestricted broadcast. Figure 21.7 shows an example of this scenario simulated in ALCHEMIST [PMV13]. Note how the consistency of the pattern does not change significantly for different densities of devices: a change in density only affects the precision of the channel shape.

21.3.2 Path Forecasting

In the uses of the `GPI` construct so far the `integrand` argument has been constant at 1 for measuring distance and 0 for broadcast. Most generally, however, this argument to `GPI` can be used to record the context along a path, by integrating the values encountered along the way.

An example of how this can be used is to forecast obstacles that may be encountered along a path, and provide warnings thereof. For example, if we set `integrand` to an indicator field that is 1 where there is an obstacle and 0 where there is not, then `GPI` outputs a field that holds a positive value only on those devices that are reached by crossing an obstacle:

```
(def obstacle-forecast (source obstacle)
  (GPI source 0 1 obstacle)
```

As an application scenario, consider a building occupied by a number of people, who need to navigate through it with various goals, e.g., finding a desired location, rendezvous with another person, exiting while avoiding hazards. Some areas of the building are currently problematic and should be avoided, e.g., due to crowding or temporary barriers or dangerous hazards. The goal is then to alert people who would normally transit across those areas while moving to their destination.

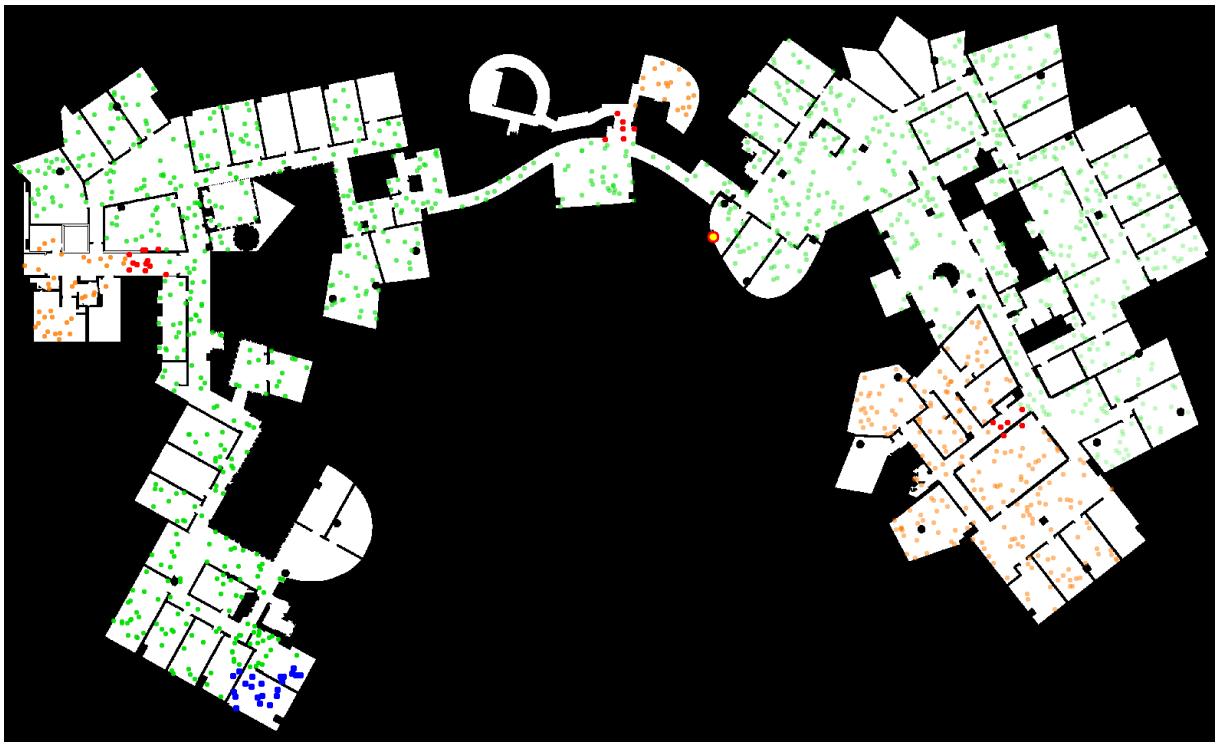


Figure 21.8: Obstacle forecast pattern deployed to alert people moving through a building of obstacles (red) on their path to a destination (blue). Distances are shown in green, and alerted areas where people must be advised of the obstacle are orange.

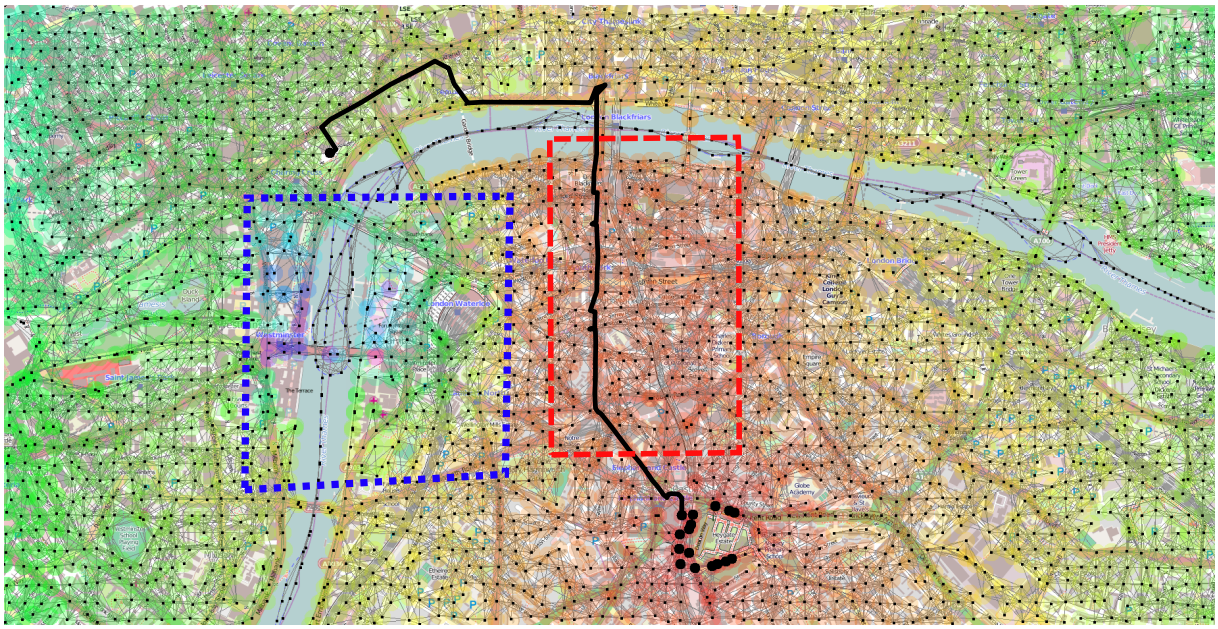


Figure 21.9: Context-sensitive distance computation deployed for navigation on a street map of London. Warmer coloured devices have a closer effective distance to the destination (black dots at bottom centre). Dashed rectangular areas represent unfavourable (blue) and favourable (red) areas for travel. An example path is shown (black line), originating near Charing Cross (black dot in upper left).

The `obstacle-forecast` pattern is a possible solution for this problem. Applying it with regards to a given destination (used, ironically, as the `source` argument), the alerted areas are those where `obstacle-forecast` returns a value greater than zero.

Figure 21.8 shows an example of this scenario simulated in ALCHEMIST for the MIT Stata Center floorplan from Figure 21.3. Note the complex manifold structure of the building modifies the shape of the sector pattern: the presence of walls considerably changes the distances compared to an open area. Despire the complexity, however, the manifold geometry ensures an eventually consistent pattern. Note also that, having alerted people to obstacles, the system could go on to provide alternate routes by running `distance-to` on a space omitting the obstacles.

21.3.3 Context-Sensitive Distance

The effective shortest path between a node and the source of a `GPI` is not necessarily the physically shortest path induced by the standard metric. Rather, the notion of effective distance may be influenced by other properties of the environment, either negatively (e.g., obstacles, congestion, pollution, tolls) or positively (e.g., safety, dedicated lanes, beauty). The `density` argument of `GPI` allows such factors to be taken into consideration as a multiplicative "stretching" of the base physical distance metric. Assuming there are penalising areas (`cons`) and favourable

areas (`pros`), both expressed as scalar fields with values between 0 (least significant) and 1 (most significant), then one way to define context sensitive distance for use in navigation is:

```
(def contextual-distance (source pros cons)
  (GPI source 0 (+ 1.1 (- cons pros))))
```

As an application scenario, consider guiding pedestrian or vehicle traffic in a complex urban environment. Devices are deployed along and around the streets. Some have the ability to sense environmental parameters such as crowding, current and expected traffic, and pollution; other parameters, such as presence of events and attractions or comments on beauty of an area, are drawn from databases of local information, either remote or distributed and localized in the network.

From these, the devices can compute the pro and con fields for contextual distance for people navigating through the city, building a spatial structure that alters the perceived distances toward a location by taking desirability into account, so that a longer path crossing desirable areas will be favored over a somewhat shorter path crossing undesirable areas.

Figure 21.9, we shows an example of this scenario for the center of London, simulated in ALCHEMIST. Two areas are marked as favorable and unfavorable. Distances are shown with respect to a destination on the South side of the Thames, showing significant asymmetries caused both by the shape of the city and the marked areas acting as an attractor and repulsor of pedestrians, respectively. In this scenario both a complex environment and a non-physical distance metric are used, and still provide eventually consistent results.

21.3.4 Confirmation of Results in Simulation

As a confirmation of our results for GPI-calculus and its application to the described scenarios, we have simulated each scenario on a wide range of densities of devices using ALCHEMIST. Based on previous analysis, we expect that for each scenario, the values yielded by devices will converge to a stable set of values that no longer change. Furthermore, as the number of devices increases, the values converged to will themselves converge as the network of devices more closely approximates a continuous space.

Conditions for the scenario simulations are as follows.

- Each scenario is run with five logarithmically scaled densities, ten runs per condition: the outdoor scenarios with 100 to 10,000 devices, the indoor scenario with 100 to 1000 devices.
- Devices are distributed randomly through the portion of the space not blocked by environmental obstacles.
- Communication range is set to 15% of width for the lowest density and reduced proportional to the square root of density, to ensure a consistent expected number of neighbours.

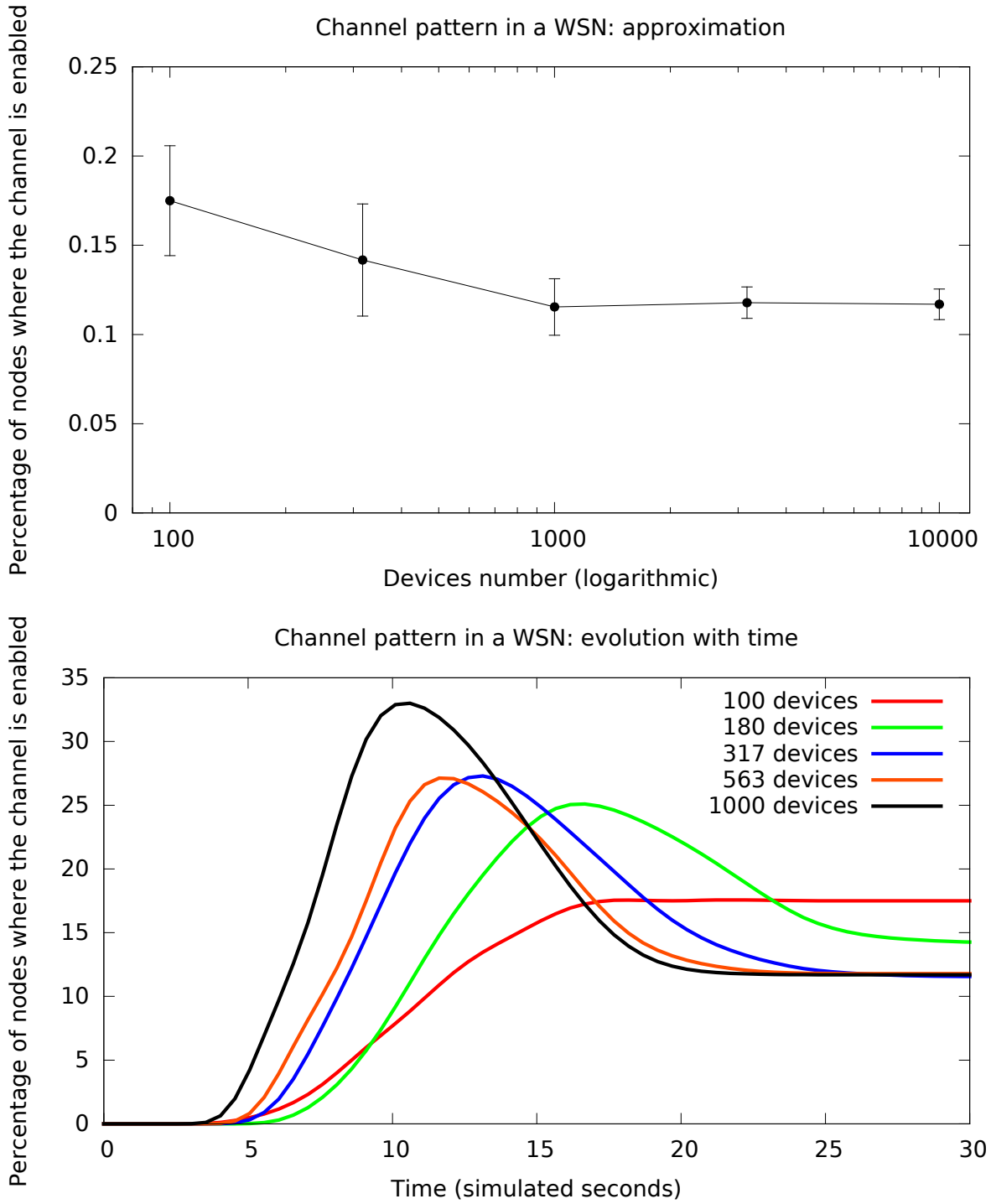


Figure 21.10: Wireless Sensor Network

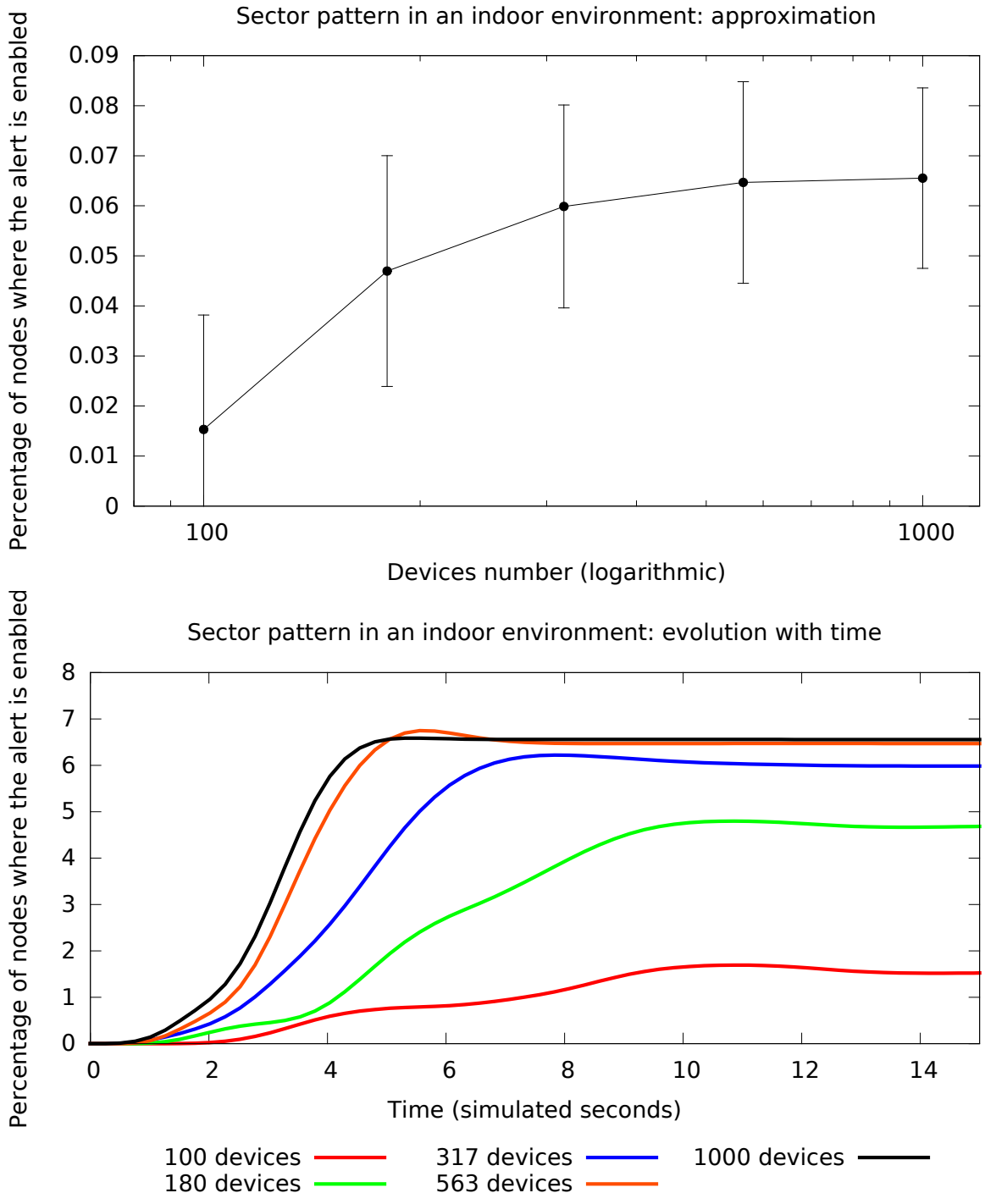


Figure 21.11: Building Alert

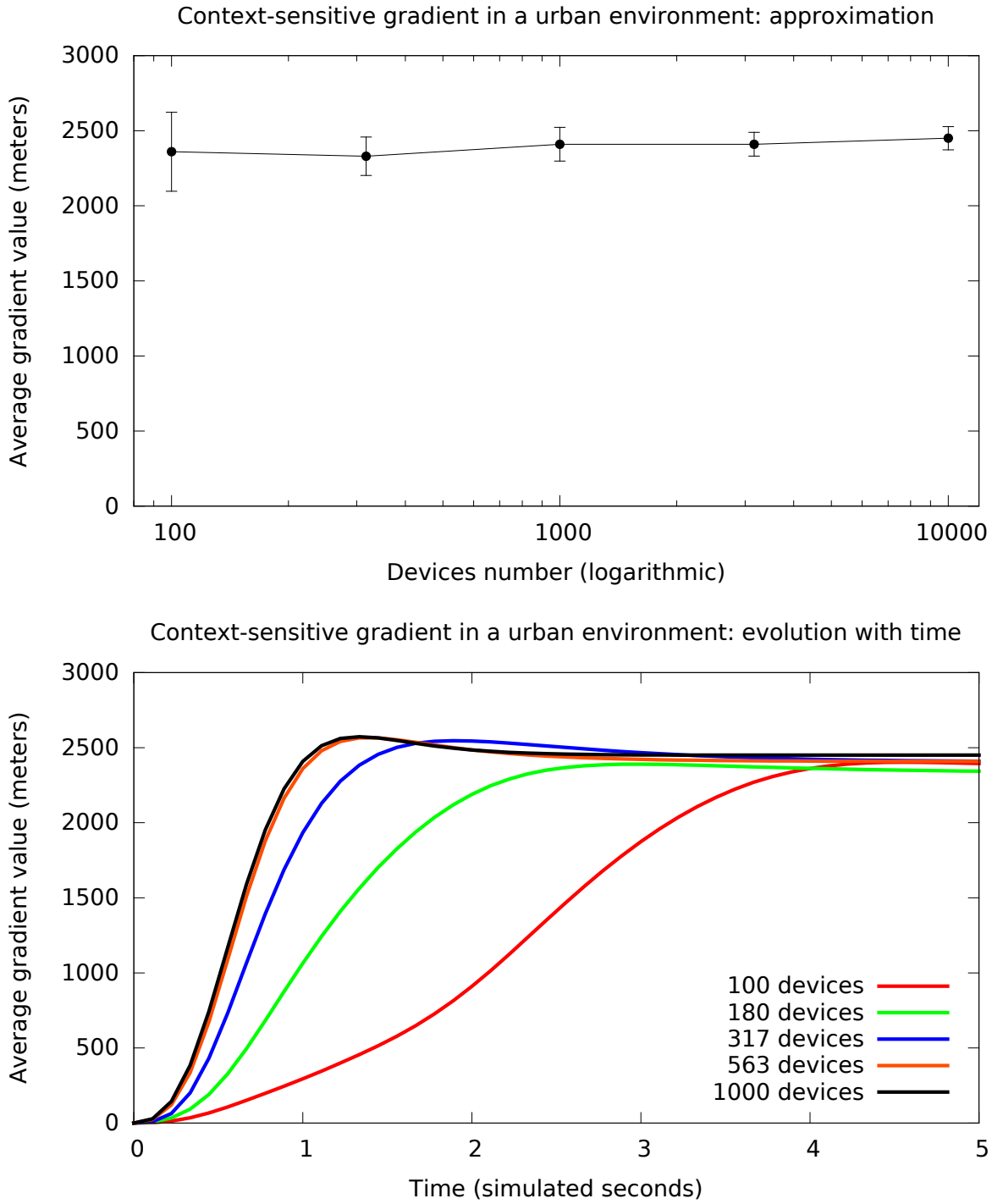


Figure 21.12: Urban Traffic Steering

Devices are connected with a unit disc rule, with permissive line-of-sight blocking by obstacles. In particular, two devices are connected if they are within distance r_1 and there is some position within a small distance $r_2 \propto r_1$ of each device in which no object blocks line-of-sight.

- Devices execute independently following a Poisson distribution, with frequency 1 Hertz for the lowest density; to compensate for shorter hops, frequency rises inversely proportional to communication range.

Results are summarized in Figure 21.10, Figure 21.11, and Figure 21.12, with sample snapshots shown in Figure 21.7, Figure 21.8, and Figure 21.9. For each set of simulations, we measure the mean and standard deviation of a key application property with respect to both density and time. The properties measured are:

- wireless sensor network: fraction of devices in channel;
- building alert: percentage of devices receiving the alert;
- urban traffic steering: average compensated distance value.

As predicted from our analytical results, these values converge with respect to both time and number of devices, confirming our predictions.

22 Higher order functions in field calculus

22.1 Impact on alignment

23 Protelis: practical aggregate programming

We have designed the Protelis language as an implementation of the field calculus [VDB13] closely related to Proto [BB06]. On the one hand, it incorporates the main spatial computing features of the field calculus, hence enjoying its universality, consistency, and self-stabilization properties [BVD14, VD14]. On the other hand, it turns the field calculus into a modern specification language, improving over Proto by providing

- access to a richer API through Java integration;
- support for code mobility through first-order functions;
- a novel syntax inspired by the more widely adopted C-family languages.

Protelis is freely available and open source, and can be downloaded as part of the ALCHEMIST distribution.

23.1 Syntax

We present the Protelis language in terms of its abstract syntax, provided in Figure 23.1 as a means to guide the discussion of the language's features. This syntax uses similar conventions to well-known core languages like Featherweight Java [IPW01]. We let meta-variable `f` range over names of user-defined functions, `x` over names of variables and function arguments, `l` over literal values (Booleans, numbers, strings), `b` over names of built-in functions and operators (including the "hood" functions described in Section 23.3), `m` over Java method names, and `#a` over aliases of static Java methods. All such meta-variables are used as non-terminal symbols in Figure 23.1. Overbar notation $\overline{y}$ generally means a comma-separated list $y_1, \ldots, y_n$ of elements of kind y, with the two exceptions that in $\overline{\texttt{F}}$ we use no comma separator, and in $\overline{\texttt{s}}$`;` semi-colon is used as separator instead.

```
P ::= Ī F̄ s̄;                                   ;; Program
I ::= import m | import m.*                     ;; Java import
F ::= def f(x̄) {s̄;}                             ;; Function definition
s ::= e | let x = e | x = e                     ;; Statement
w ::= x | l | [w̄] | f | (x̄)->{s̄;}               ;; Variable/Value
e ::= w                                         ;; Expression
    | b(ē) | f(ē) | e.apply(ē)                  ;; Fun/Op Calls
    | e.m(ē) | #a(ē)                            ;; Method Calls
    | rep(x<-w){s̄;}                             ;; Persistent state
    | if(e){s̄;}else{s̄′;}                        ;; Exclusive branch
    | mux(e){s̄;}else{s̄′;}                       ;; Inclusive branch
    | nbr{s̄;}                                   ;; Neighborhood values
```

Figure 23.1: Protelis abstract syntax, colored to emphasize definition and application (red), functions (blue), variables (green), and special field calculus operators (purple).

23.2 Ordinary Language Features

One of the distinctive elements of Protelis when compared to other aggregate programming languages (particularly Proto), is the adoption of a familiar C- or Java-like syntax, which can significantly reduce barriers to adoption. Despite this syntactic similarity, Protelis is a purely functional language: a program is made of a sequence of function definitions ($\mathtt{F}_1 \ldots \mathtt{F}_n$), modularly specifying reusable parts of system behavior, followed by a main block of statements. Following the style of C-family languages, a function's body is a sequence of statements surrounded by curly brackets. As in the Scala programming language[1], however, statements can also be just expressions, and a statement sequence evaluates to the result of the last statement. Each statement is an expression to be evaluated (`e`), possibly in the context of the creation of a new variable (`let x = e`) or a re-assignment (`x = e`)[2]. As an example, consider the following function taking four fields as parameters, after the "channel" pattern from [But02]:

```
def channel(distA, distB, distAB, width) {
    let  d  =  distA + distB;
    d  =  d - distAB;
    d < width
}
```

[1] http://www.scala-lang.org.

[2] Technically, "re-assignment" is actually the creation of a new variable that shadows the old.

This function assumes that its input `distA` maps each device to its distance to a region *A*, `distB` maps each device to its distance to a region *B*, `distAB` is a constant field holding at each device the minimum distance between regions *A* and *B*, and `width` is a constant field holding at each device the same positive number. The function then computes a Boolean field, mapping each device to `true` only if it belongs to a "channel" area around the shortest path connecting regions *A* and *B* and approximately `width` units wide. All devices elsewhere map to `false`.

Atomic expressions `w` can be literal values (`l`), variables (`x`), tuples (`[`$\overline{\texttt{w}}$`]`), function names (`f`) or lambdas (`(`$\overline{\texttt{x}}$`) ->{`$\overline{\texttt{s}}$`;}`). Structured expressions include three kinds of "calls": *(i)* `b(`$\overline{\texttt{e}}$`)` is application to arguments $\overline{\texttt{e}}$ of a built-in operation `b`, which could be any (infix- or prefix-style) mathematical, logical or purely algorithmic function [3] ; *(ii)* `f(`$\overline{\texttt{e}}$`)` is application of a user-defined function; and *(iii)* `e.apply(`$\overline{\texttt{e}}$`)` is application of arguments to an expression `e` evaluating to a lambda or function name. The following shows examples of such calls:

```
def square(x) {
   x * x;
}
let f = square;
let g = (x) -> {
   square(x) + 1
};
f.apply(g.apply(2))        // gives 25 on all devices
```

In addition, arbitrary Java method calls can be imported and used by Protelis: *(i)* `e.m(`$\overline{\texttt{e}}$`)` is method call on object `e` and *(ii)* `#a(`$\overline{\texttt{e}}$`)` is invocation of a static method, via an alias `#a` (always starting with '#') defined by an `import` clause. The alias is created automatically as the bare method name for single imports or imports of all methods in a class with `*`. Protelis can thus interact with Java reflection to support dynamic invocation of arbitrary Java code, as shown in the following example:

```
import java.lang.Class.forName
let c = #forName("String");
let m = c.getMethod("length");
m.invoke("Lorem ipsum dolor sit amet")// gives 26 on all devices
```

23.3 Special Field Calculus Operators

The remaining constructs of Protelis are the special operations specific to field calculus, dealing with the movement of information across space and time:

[3] For simplicity of presentation, we omit the syntax for infix operations and order of operations, which is closely patterned after Java.

- Construct `rep(x<-w){`$\overline{s}$`;}` defines a locally-visible variable `x` initialized with `w` and updated at each computation round with the result of executing body `{`$\overline{s}$`;}`: it provides a means to define a field evolving over time according to the update policy specified by `{`$\overline{s}$`;}`.

- Construct `nbr{`$\overline{s}$`;}` executed in a device gathers a map (actually, a field) from all neighbors (including the device itself) to their latest value from computing $\overline{s}$. A special set of built-in "hood" functions can then be used to summarize such maps back to ordinary expressions. For example, `minHood` finds the minimum value in the map.

- The branching constructs `mux(e){`$\overline{s}$`;}else{`$\overline{s}'$`;}` and `if(e){`$\overline{s}$`;}else{`$\overline{s}'$`;}` perform two critically different forms of branching. The `mux` construct is an inclusive "multiplexing" branch: the two fields obtained by computing $\overline{s}$ and $\overline{s}'$ are superimposed, using the former where `e` evaluates to `true`, and the second where `e` evaluates to `false`. Complementarily, `if` performs an exclusive branch: it partitions the network into two regions: where `e` evaluates to `true` $\overline{s}$ is computed, and elsewhere $\overline{s}'$ is computed instead.

The following code shows some example uses of these constructs:

```
def count() {
   rep(x<-0){ x + 1}
}

def maxh(field) {
   maxHood(nbr{field})
}

def distanceTo(source) {
   rep(d <- Infinity) {
      mux (source) {
         0
      } else {
         minHood(nbr{d} + nbrRange)
      }
  }
}

def distanceToWithObstacle(source,obstacle) {
   if (obstacle) {
      Infinity
   } else {
      distanceTo(source)
   }
}
```

Function `count` yields an evolving field, counting how many computation rounds have been executed in each device. Function `maxh` yields a field mapping each device the maximum value of `field` across its neighborhood—note a `nbr` construct should always be eventually wrapped inside a "hood" function. Function `distanceTo` nests `nbr` inside `rep` to create a chain of interactions across many hops in the network, computing minimum distance from any device to the nearest "source device" (i.e., where `source` holds `true`). It does so by a field `d` initially `Infinity` everywhere, and evolving as follows: `d` is set to 0 on sources by `mux`, and elsewhere takes the minimum across neighbors of the values obtained by adding to `d` the estimated distance to the current device—a triangle inequality relaxation computing a distance field also often termed *gradient* [LK87, BBVT08, VCMZ11]. Finally, function `distanceToWithObstacle` shows exclusive branch at work; `distanceTo(source)` is computed in the sub-region where there is no obstacle, which causes the computation of distances to implicitly circumvent such obstacles.

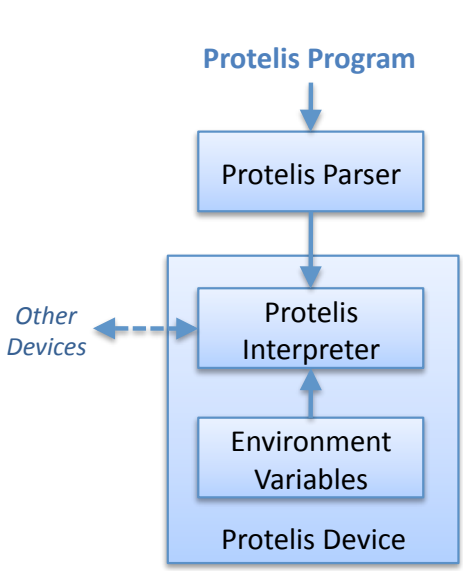
23.4 Architecture

In Protelis, we designed an architecture, subsumed in Figure 23.2(a), following the same general pattern as was used for the Proto Virtual Machine [BB07]. First, a parser translates a text Protelis program into a valid representation of field calculus semantics. This is then executed by a Protelis interpreter at regular intervals, communicating with other devices and drawing contextual information from environment variables implemented as a tuple store of $(token, value)$ pairs. This abstraction is instantiated for use on particular devices or simulations by setting when executions occur, how communication is implemented and the contents of the environment.

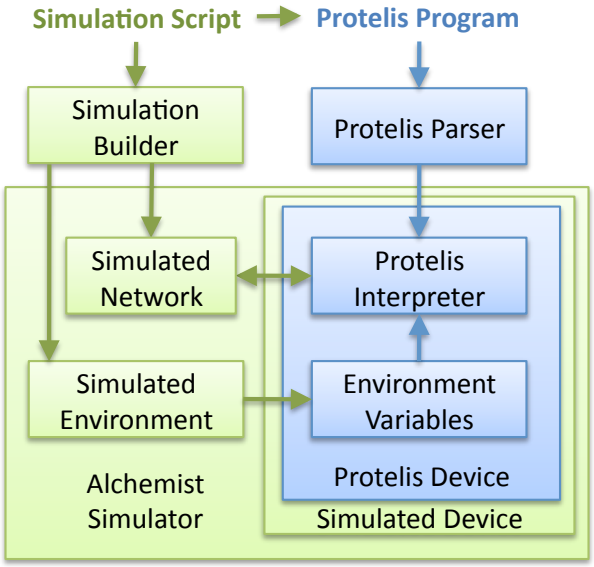
We have chosen to implement this architecture in Java. One key reason for this choice is that Java is highly portable across systems and devices. Another key reason (discussed further in the next section) is that Java's reflection mechanisms make it easy to import a large variety of useful libraries and APIs for use in Protelis. Finally, the pragmatics of execution on embedded devices have also changed significantly since the publication of [BB07]: a much wider variety of low cost embedded devices are now capable of supporting Java, while at the same time improvements in Java implementations have made it much more competitive in speed and resource cost with low-level languages like C [BSB+03, ORAI11].

In particular, we have chosen to implement Protelis and its architecture via the Xtext language generator [EB10] and within the ALCHEMIST framework [PMV13]. Usefully, Xtext also features support for generating a language-specific Eclipse plug-in, which provides developer assistance through code highlighting, completion suggestions, and compile-time error detection.

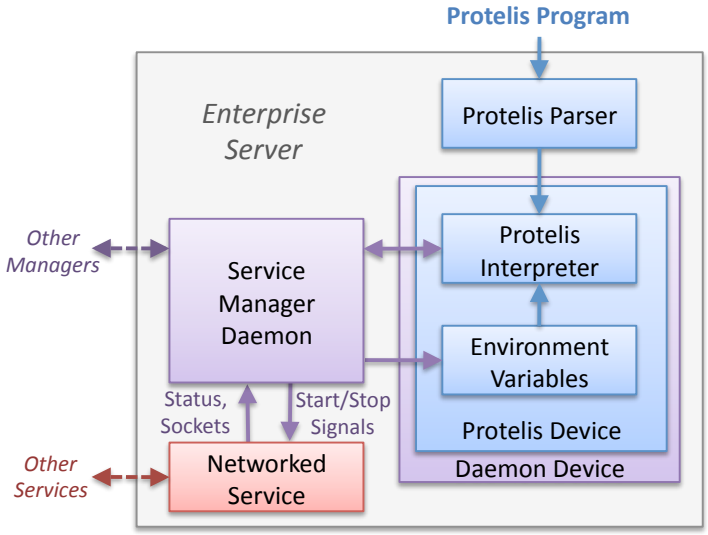
For an initial validation, we have exercised this architecture by construction of two instantiations: one in the ALCHEMIST framework for simulation of large-scale spatially-embedded systems; the other as a daemon for coordinating management of networked services. Figure 23.2(b) shows the ALCHEMIST instantiation: simulations are configured using a simple scripting language, which specifies a Protelis program as well as the collection of devices that will execute it, communication between those devices, and other aspects of the environment to be simulated. The ALCHEMIST event-driven simulation engine then handles execution scheduling, message delivery, and updates to the environment tuple store. Figure 23.2(c) shows the network service management instantiation. Here, each Protelis device lives on a separate server in an enterprise network, and is tethered to the networked service it is intended to manage by a service manager daemon. This daemon monitors the service, injecting information about its status and known dependencies into the environment and maintaining a neighborhood by opening parallel communication links to the corresponding daemons on any other servers that the monitored service communicates with. Examples using each of these implementations are shown in Section 23.5 and Section 23.6.



(a) Abstract Architecture

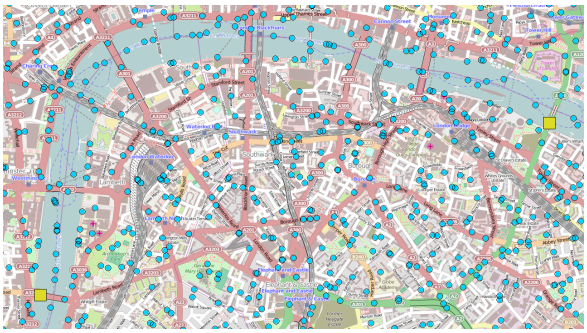


(b) ALCHEMIST Simulation

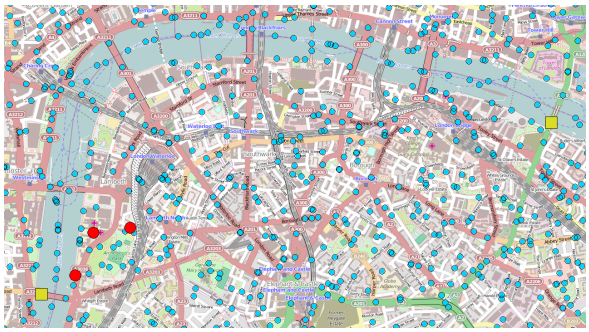


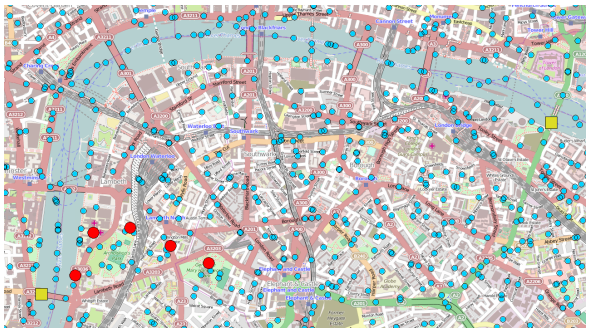
(c) Network Service Management

Figure 23.2: In the abstract Protelis architecture (a), an interpreter executes a pre-parsed Protelis program at regular intervals, communicating with other devices and drawing contextual information from a store of environment variables. This is instantiated by setting when executions occur, how communication is implemented and the contents of the environment. Two such instantiations are presented in this thesis: as a simulation in the ALCHEMIST framework (b) and as a daemon for coordinating management of networked services (c).

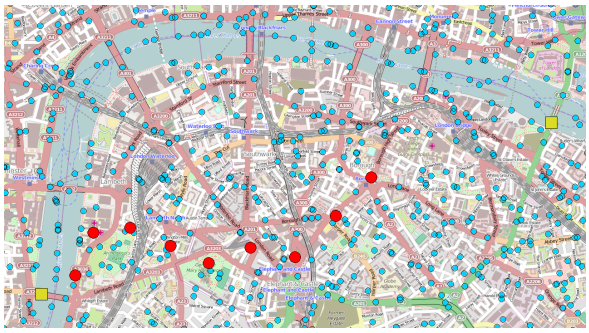


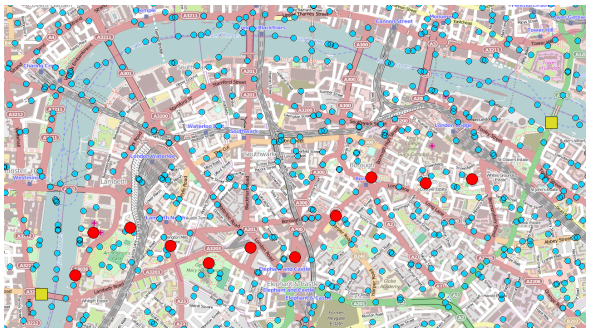
(a) Initial configuration



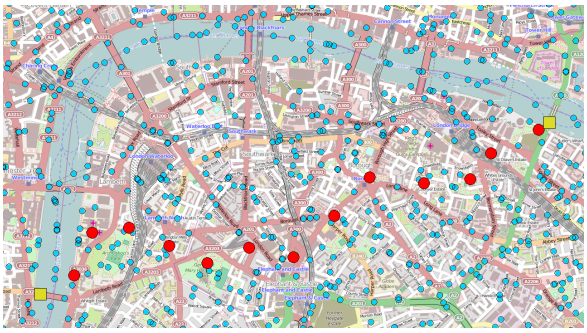


(b) Path begins to form





(c) Path continues to extend



(d) Path computation complete

Figure 23.3: Example of computing a rendezvous route for two people in a crowded urban environment.

23.5 Application example: Rendezvous at a Mass Event

A common problem in large public events is to rendezvous with other companions attending the same large public event. At mass events, access to external cloud-based services may be difficult or impossible, and pre-arranged rendezvous points may be inaccessible or inconveniently distant. Simple peer-to-peer geometric calculations across the network, however, can readily compute a route that will allow two people to rendezvous:

```
// Follow the gradient of a potential field down from a source
def descend(source,potential) {
    rep(path <- source) {
        let nextStep = minHood(nbr([potential, self.getId()]));
        if (nextStep.size() > 1) {
            let candidates = nbr([nextStep.get(1), path]);
            source || anyHood([self.getId(), true] == candidates)
        } else {
            source
        }
    }
}
def rendezvous(person1, person2) {
    descend (person1 == owner, distanceTo(person2 == owner))
}
// Example of using rendezvous
rendezvous("Alice", "Bob");
```

Figure 23.3 shows an example of running this rendezvous process in a simulated city center. We chose London as a simulation environment, using Alchemist's capability for importing OpenStreetMap data. We displaced 1000 devices randomly across the city streets (represented by pale blue dots), with a communication range of 475 meters (this range chosen to ensure no network segmentation). We then picked two devices whose owners want to meet: one device on Lambeth Bridge (lower left of the image) and one device on Tower Bridge (upper right), each marked with a yellow square. To mark the devices for Protelis, we injected their environments with a property `owner`, assigning the strings `"Alice"` and `"Bob"` as values for the first and the second device respectively.

Implementing this application requires only 21 lines of code: the listing above and the function `distanceTo` that can be found in Section 23.3. This implementation measures distance to one of the participants, creating a potential field, then, starting from the other one, builds an optimal path descending the distance potential field to return to the first participant at distance zero. The first half of the algorithm has already been described, and relies on `distanceTo`, while the second half is implemented by the function `descend`. This function,

given a device and a potential field, builds a path of devices connecting the former with the source of the latter. The strategy is to mark the device we want to connect to the potential field's source as part of the path, and then, in every device, compute which of the neighbors is closest to the destination. Given this information, a device is in the path if one of the neighbors is in the path already and has marked this device as the closest of its neighbors towards the destination. Note how the whole algorithm can be elegantly compressed into just a few lines of code, and how there is no need to explicitly declare any communication protocol for exchanging the required information, thanks to the repeated use of the `nbr` operator.

As Figure 23.3 shows, once the simulation starts, a chain of devices is rapidly identified (red dots), marking a sequence of way-points for both device owners to walk in order to meet in the middle. Note also that, due to the ongoing nature of the computation, if one of the device owners moves in a different direction instead, the path will automatically adjust so that it continues to recommend the best path for rendezvous.

23.6 Application example: Network Service Management

One of the common problems in managing complex enterprise services is that there are often many dependencies between different servers and services. Frequently, some of these services are legacy or poorly coded, such that they do not respond gracefully to the failure of their dependencies. These services may continue to attempt to operate for some time, creating inconsistent state, or may be unable to resume service correctly after the server they depend on is brought back on line.

Thus, responding to a service failure often requires a coordinated shutdown and restart of services in an order dictated by service dependencies. This type of service management can be automated by attaching a daemon that watches the state of each service, then communicates with the daemons of other services to coordinate shutdown and restart in accordance with their dependencies.

Figure 23.4(a) shows an example scenario of an enterprise network for a small manufacturing and supply company, with dependencies between two key databases and the internal and external servers running web applications. This scenario was implemented on a network of EmuLab [WLS+02] servers. The services were emulated as simple query-response networking programs in Java that entered a "hung" state either upon being externally triggered to crash or after their queries began to consistently fail.

Each service was wrapped with an embedded Protelis execution engine, which was interfaced with the services by a small piece of monitoring glue code that inserted environment variables containing an identifier for the `serviceID` running on that server, a tuple of identifiers for `dependencies`, and the current `managedServiceStatus` of `stop`, `starting`, `run`, `stopping`, or `hung`. The glue code also provides `stopService` and `startService` methods to send signals to the service, tracks interactions between the services in order to maintain the set of neighbors for Protelis, and allows an external monitoring application to attach

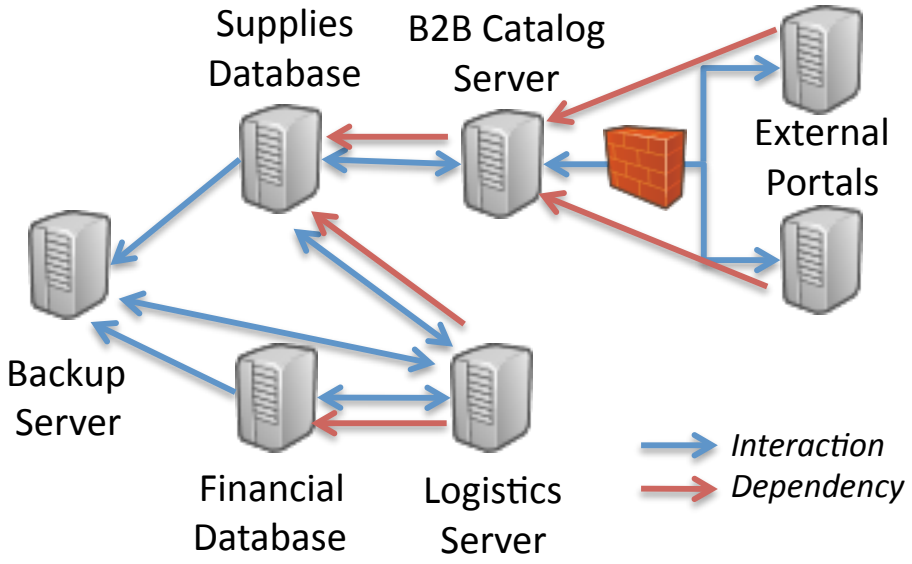
and receive status reports.

Dependency-directed coordination of service starting and stopping was then implemented as follows:

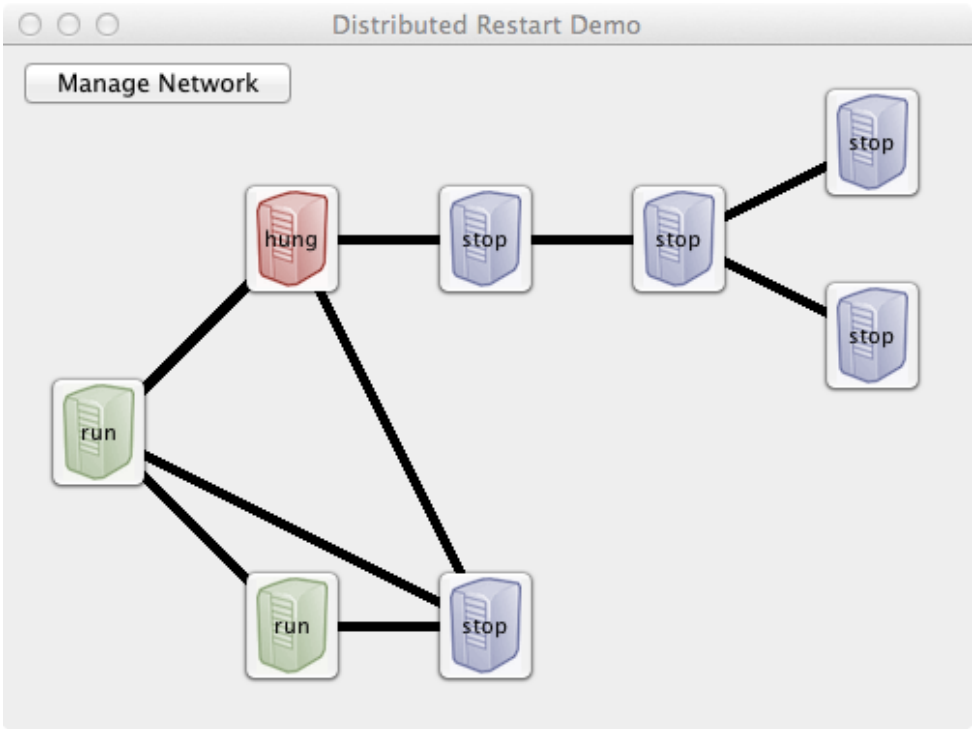
```
import it.unibo.alchemist.language.protelis.datatype.Tuple.*
import com.bbn.a3.distributedrestart.DaemonNode.*

// Compare required and available services
let nbr_set = unionHood(nbr([serviceID]));
let nbr_missing = dependencies.subtract(nbr_set);
let nbr_required = #contains(dependencies,nbr(serviceID));
let nbr_down = nbr(managedServiceStatus=="hung" ||
                   managedServiceStatus=="stop");

// Is service currently safe to run?
let problem = anyHood(nbr_down && nbr_required) ||
              !nbr_missing.isEmpty();

// Take managed service up and down accordingly
if (managedServiceStatus=="run" && problem) {
  #stopProcess(managedService);
} else {
  if (managedServiceStatus=="stop" && !problem) {
    #startProcess(managedService);
  } else {
    managedServiceStatus
  }
}
```

In this program, each device shares information about its service ID and status with its neighbors, enabling them to track which dependencies are currently down or missing. When there is a problem with dependencies, the device invokes `stopProcess` to shut its service down, when dependencies are good, it brings it up again with `startProcess`, and when it is hung it waits for a human to sort out the problem.

Figure 23.4(b) shows a typical screenshot of the network of services in operation on an EmuLab network of Ubuntu machines, one service per machine, as visualized by the monitoring application. In this screenshot, the supplies database has crashed, causing many of the other services to gracefully shut themselves down. As soon as the supplies database is restarted, however, the rest of the services automatically bring themselves up in dependency order.



(a) Example Dependent Services Scenario



(b) Example of Coordinated Restart Execution

Figure 23.4: (a) An example scenario of an enterprise network for a small manufacturing and supply company. (b) Example of execution on a network of 8 EmuLab [WLS+02] machines: the supplies database has crashed (red), and so all dependent services have shut themselves down (blue), while other services continue to run normally (green).

Part V

Conclusion

24 Results achieved

24.1 Integrated toolchain for pervasive ecosystems

24.2 Methods and patterns for self organisation

24.2.1 A process algebra for SAPERE

The key contribution of a process algebra to provide a unequivocal description of the components of the infrastructure, and also serves as an executable specification from which can be derived proofs of behavioural properties, like self-stabilisation of the self-organising spatial structures that are useful in the context of pervasive computing systems (along the lines of [Vir13]).

24.3 Aggregate programming languages

24.3.1 Scale independent computation in situated networks

We have developed the first practical distributed language which provides guarantees that all distributed algorithms expressed in this language are resilient against changes in the number and distribution of devices in a network. This is an important step towards a more general framework for supporting open ecosystems of pervasive wireless devices, which need to provide safe and resilient services despite running a shifting set of interacting services from many unrelated software suppliers. If it is possible to ensure that the only programs that can be expressed are those that interact well together, then it will greatly reduce the cost of providing reliable services in such environments.

24.3.2 Higher Order Functions in Field Calculus

24.3.3 Protelis

Protelis ensures universality and coherence between aggregate specification and local execution by building atop the field calculus introduced in [VDB13]. At the same time, accessibility,

portability, and ease of integration are ensured by embedding Protelis within Java. This enables Protelis programs to draw on the full breadth of available Java APIs and to readily integrate with a wide range of devices and applications, as illustrated by our examples of pervasive computing simulation and networked service management. This implementation of Protelis thus forms an important component of the toolchain necessary for practical application of aggregate programming principles and methods to address real-world problems.

25 Future and ongoing work

25.1 Aggregate programming

25.1.1 Scale independent computation in situated networks

We want to generalise the results of our scale independent computations: the theoretical framework of this paper treats only the case of stationary devices; in pervasive systems, many devices are mobile, and in practice the mechanisms used by GPI-calculus appear to perform quite well on mobile devices, so it will be important to extend the theoretical framework to handle mobile devices as well. One notable concern is that mobile devices must treat questions of Galilean vs. Einsteinian relativity [TW92], which we deferred for now.

Similarly, the theory currently only handles eventual approximation and behavior at the limit of high density networks, but these properties are most useful because in practice they tend to be indicators that an algorithm also behaves well in lower density networks and before it has finished converging. It should be possible to identify stronger properties that demarcate good performance at lower densities as well.

`GPI` is a powerful operator, but it only covers a few of the commonly used "building block" resilient distributed algorithms; another area for future extension is thus to broaden the scope of distributed applications for which resilience guarantees can be provided by extension of GPI-calculus to cover additional "building block" algorithms.

25.1.2 Protelis

The Protelis framework continues to be actively developed: we plan to enrich it in the future by adding higher-level abstractions for aggregate programming grounded on the mechanisms discussed in this work.

Also, we observe that the current situation is much like that of cooperative multitasking in the operating systems world, before the development of the current kernel-user distinction and pre-emptive multitasking that dominates the world now. We think that Protelis could be a new architecture for decentralized pre-emptive management of distributed processes, which provides these same sort of capabilities for distributed systems. In particular, we want to exploit the

field calculus model for aggregate programming of distributed systems, extending it to handle first-class functions, which enables its alignment mechanism for pre-emptive control of the scope of a distributed computation to be used for process managament. These mechanisms enable the construction of a prototype "thread manager" for pervasive and distributed systems, including the ability to preempt a process, to kill a rogue process, or to dynamically manage the scope of a process. This point of view will be deeply analysed a paper which is an ongoing work.

It will also be important to consider how static analysis, testing, and model-checking techniques can be incorporated to extend the capabilities, especially for ensuring that semantic faults in programs can be eliminated before runtime.

25.2 Biochemical meta model for ALCHEMIST

ciao

Bibliography

[AAC+00] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Jr., Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. Amorphous computing. *Communications of the ACM*, 43:74–82, 2000.

[AARC+12] Seyedreza Abdollahi, Hamed Al-Raweshidy, Erik Calvo, Luis Manuel Dahlman, Klaus David, Marian Codreanu, Panagiotis Demestichas, Markus Dillinger, Miguel Egido, Markus Gruber, Stefan Kaiser, Matti Latva-aho, Peter Merz, Xavier Mestre, Emmanouil Panaousis, Stephan Pfletschinger, Mahdi Pirmoradian, Christos Politis, Bernhard Raaf, Frank Schaich, Christoph Schmelz, Rahim Tafazolli, Antti Tölli, Nikola Vucic, and Rudolf Winkelmann. Spectrum Crunch. Technical report, Net!Works European Technology Platform, 06 2012.

[ABI09] Marco Autili, Paolo Benedetto, and Paola Inverardi. Context-aware adaptive services: The plastic approach. In *FASE '09 Proceedings*, pages 124–139, Berlin, Heidelberg, 2009. Springer-Verlag.

[AGS11] Michael Adeyeye and Paul Gardner-Stephen. The village telco project: a reliable and practical wireless mesh telephony infrastructure. *EURASIP J. Wireless Comm. and Networking*, 2011:78, 2011.

[AJL+02] Bruce Alberts, Alexander Johnson, Julian Lewis, Martin Raff, Keith Roberts, and Peter Walter. *Molecular Biology of the Cell*. Garland Science Textbooks. Garland Science, Fourth edition, June 2002.

[AM11] Musab AlTurki and José Meseguer. Pvesta: A parallel statistical model checking and quantitative analysis tool. In *CALCO*, 2011.

[AMS06] Gul A. Agha, José Meseguer, and Koushik Sen. PMaude: Rewrite-based specification language for probabilistic object systems. In Antonio Cerone and Herbert Wiklicky, editors, *QAPL 2005*, volume 153(2) of *ENTCS*, pages 213–239. Elsevier, 2006.

[Apa13] Tatjana Apanasevic. Factors influencing the slow rate of penetration of NFC mobile payment in western europe. In *12th International Conference on Mobile Business, ICMB 2013, Berlin, Germany, 10-13 June 2013*, page 8, 2013.

[APNF13] Bernhard Anzengruber, Danilo Pianini, Jussi Nieminen, and Alois Ferscha. Predicting social density in mass events to prevent crowd disasters. In Adam Jatowt, Ee-Peng Lim, Ying Ding, Asako Miura, Taro Tezuka, Gaël Dias, Katsumi Tanaka, Andrew J. Flanagin, and Bing Tian Dai, editors, *SocInfo*, volume 8238 of *Lecture Notes in Computer Science*, pages 206–215. Springer, 2013.

[ARGL+07] Michael P. Ashley-Rollman, Seth Copen Goldstein, Peter Lee, Todd C. Mowry, and Padmanabhan Pillai. Meld: A declarative approach to programming ensembles. In *IEEE International Conference on Intelligent Robots and Systems (IROS '07)*, pages 2794–2800, 2007.

[ASS+11] Henrik Abramowicz, Ernst-Dieter Schmidt, Lars Christoph Schmelz, Cornel Pampu, Cornelia Kappler, Mirko Schramm, Konstantinous Pentikousis, Dongming Zhou, Savo Glisic, Juan Palacios, Josep Mangues, Panagiotis Demestichas, Ralf Toenjes, Tipu Arvind Ramrekha, Grant Millar, Christos Politis, Sergi Figuerola, Dimitra Simeonidou, Alex Galis, and Rahim Tafazolli. Future Networks and Management . Technical report, Net!Works European Technology Platform, 06 2011.

[BB06] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.

[BB07] Jonathan Bachrach and Jacob Beal. Building spatial computers. Technical Report MIT-CSAIL-TR-2007-017, MIT, March 2007.

[BBF07] Jonathan Bachrach, Jacob Beal, and Takeshi Fujiwara. Continuous space-time semantics allow adaptive program execution. In *IEEE SASO 2007*, pages 315–319, New York, July 2007. IEEE.

[BBG06] Maurice Beek, Antonio Bucchiarone, and Stefania Gnesi. A survey on service composition approaches: From industrial standards to formal methods. In *Technical Report 2006TR-15, Istituto*, pages 15–20. IEEE CS Press, 2006.

[BBL+11] Kevin Burrage, PamelaM. Burrage, André Leier, Tatiana Marquez-Lago, and Jr Nicolau, DanV. Stochastic simulation for spatial modelling of dynamic processes in a living cell. In Heinz Koeppl, Gianluca Setti, Mario di Bernardo, and Douglas Densmore, editors, *Design and Analysis of Biomolecular Circuits*, pages 43–62. Springer New York, 2011.

[BBVT08] Jacob Beal, Jonathan Bachrach, Daniel Vickery, and Mark Tobenkin. Fast self-healing gradients. In *Proceedings of ACM SAC 2008*, pages 1969–1975, 2008.

[BCD+06] Özalp Babaoglu, Geoffrey Canright, Andreas Deutsch, Gianni Di Caro, Frederick Ducatelle, Luca Maria Gambardella, Niloy Ganguly, Márk Jelasity, Roberto Montemanni, Alberto Montresor, and Tore Urnes. Design patterns from biology for distributed computing. *TAAS*, 1(1):26–66, 2006.

[BDT99] Eric Bonabeau, Marco Dorigo, and Guy Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.

[BDU+13] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. Organizing the aggregate: Languages for spatial computing. In Marjan Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, pages 436–501. IGI Global, 2013. A longer version available at: `http://arxiv.org/abs/1202.5509`.

[Bea05] Jacob Beal. Programming an amorphous computational medium. In Jean-Pierre Banatre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 121–136. Springer Berlin Heidelberg, 2005.

[Bea09] Jacob Beal. Flexible self-healing gradients. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, pages 1197–1201. ACM, 2009.

[Bea10] Jacob Beal. A basis set of operators for space-time computations. In *Spatial Computing Workshop*, 2010. Available at: `http://www.spatial-computing.org/scw10/`.

[Ber92a] Gérard Berry. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

[Ber92b] Alan A. Berryman. The origins and evolution of predator-prey theory. *Ecology*, 73(5):1530–1535, October 1992.

[BFLM01] Jean-Pierre Banâtre, Pascal Fradet, and Daniel Le Métayer. Gamma and the chemical reaction model: Fifteen years after. In Cristian S. Calude, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Multiset Processing. Mathematical, Computer Science, and Molecular Computing Points of View*, volume 2235 of *LNCS*, pages 17–44. Springer, 2001.

[BGZ97] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. On the turing equivalence of linda coordination primitives. *Electr. Notes Theor. Comput. Sci.*, 7:75, 1997.

[BGZ98] Nadia Busi, Roberto Gorrieri, and Gianluigi Zavattaro. A process algebraic view of linda coordination primitives. *Theor. Comput. Sci.*, 192(2):167–199, 1998.

[BLM90] Jean-Pierre Banâtre and Daniel Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15(1):55–77, November 1990.

[BLM13] Luca Bortolussi, Diego Latella, and Mieke Massink. Stochastic process algebra and stability analysis of collective systems. In *COORDINATION*, volume 7890 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2013.

[BMM02] Özalp Babaoglu, Hein Meling, and Alberto Montresor. Anthill: A framework for the development of agent-based peer-to-peer systems. In *Proceedings of the 22 Nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, pages 15–, Washington, DC, USA, 2002. IEEE Computer Society.

[BMP+11] Chatschik Bisdikian, Bernhard Mitschang, Dino Pedreschi, Vincent S. Tseng, and Claudio Bettini. Challenges for mobile data management in the era of cloud and social computing. In *12th IEEE International Conference on Mobile Data Management*, page 6, 2011.

[BMS11] Jacob Beal, Olivier Michel, and Ulrik Pagh Schultz. Spatial computing: Distributed systems that take advantage of our geometric world. *ACM Transactions on Autonomous and Adaptive Systems*, 6:11:1–11:3, June 2011.

[BMV09a] Stefania Bandini, Sara Manzoni, and Giuseppe Vizzari. Agent based modeling and simulation: An informatics perspective. *Journal of Artificial Societies and Social Simulation*, 12:4, 2009.

[BMV09b] Stefania Bandini, Sara Manzoni, and Giuseppe Vizzari. Crowd Behavior Modeling: From Cellular Automata to Multi-Agent Systems. In Adelinde M. Uhrmacher and Danny Weyns, editors, *Multi-Agent Systems: Simulation and Applications*, Computational Analysis, Synthesis, and Design of Dynamic Systems, chapter 13, pages 389–418. CRC Press, June 2009.

[Bon99] Eric Bonabeau. Editor's introduction: Stigmergy. *Artificial Life*, 5(2):95–96, Spring 1999.

[BP09] Jean-Pierre Banâtre and Thierry Priol. Chemical programming of future service-oriented architectures. *JSW*, 4(7):738–746, 2009.

[BSB+03] J. Mark Bull, L. A. Smith, C. Ball, L. Pottage, and R. Freeman. Benchmarking java against c and fortran for scientific applications. *Concurrency and Computation: Practice and Experience*, 15(3-5):417–430, 2003.

[BUB13] Jacob Beal, Kyle Usbeck, and Brett Benyo. On the evaluation of space-time functions. *The Computer Journal*, 56(12):1500–1517, 2013. doi: 10.1093/comjnl/bxs099.

[But02] William Butera. *Programming a Paintable Computer*. PhD thesis, MIT, Cambridge, MA, USA, 2002.

[BV07] Stefania Bandini and Giuseppe Vizzari. Regulation function of the environment in agent-based simulation. In *Proceedings of the 3rd international conference on Environments for multi-agent systems III*, E4MAS'06, pages 157–169, Berlin, Heidelberg, 2007. Springer-Verlag.

[BVD14] Jacob Beal, Mirko Viroli, and Ferruccio Damiani. Towards a unified model of spatial computing. In *7th Spatial Computing Workshop (SCW 2014)*, AAMAS 2014, Paris, France, May 2014.

[BW08] Eduard Babulak and Ming Wang. Discrete event simulation: State of the art. *iJOE*, 4(2):60–63, 2008.

[CDB+12] Marco Conti, Sajal K. Das, Chatschik Bisdikian, Mohan Kumar, Lionel M. Ni, Andrea Passarella, George Roussos, Gerhard Tröster, Gene Tsudik, and Franco Zambonelli. Looking ahead in pervasive computing: Challenges and opportunities in the era of cyber-physical convergence. *Pervasive and Mobile Computing*, 8(1):2–21, 2012.

[CDPEV05] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1069–1075, New York, NY, USA, 2005. ACM.

[CG00] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, June 2000.

[CG09] Federica Ciocchetta and Maria Luisa Guerriero. Modelling biological compartments in bio-pepa. *Electr. Notes Theor. Comput. Sci.*, 227:77–95, 2009.

[CG10] Luca Cardelli and Philippa Gardner. Processes in space. In *Programs, Proofs, Processes, 6th Conference on Computability in Europe, CiE 2010*, volume 6158 of *Lecture Notes in Computer Science*, pages 78–87. Springer, 2010.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of mathematics*, 33(2):346–366, 1932.

[CLMZ03] Giacomo Cabri, Letizia Leonardi, Marco Mamei, and Franco Zambonelli. Location-dependent services for mobile users. *IEEE Transactions on Systems, Man, and Cybernetics, Part A*, 33(6):667–681, 2003.

[CLZ00] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[Cow00] Richard Cowan. Stochastic models for dna replication. In *In Handbook of Statistics*. Elsevier, 2000.

[CVG09] Matteo Casadei, Mirko Viroli, and Luca Gardelli. On the collective sort problem for distributed tuple spaces. *Science of Computer Programming*, 74(9):702–722, 2009.

[DDF+06] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems*, 1(2):223–259, December 2006.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DLM05] Rocco De Nicola, Diego Latella, and Mieke Massink. Formal modeling and quantitative analysis of klaim-based mobile systems. In *SAC*, pages 428–435. ACM, 2005.

[DS04] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. MIT Press, Cambridge, MA, July 2004.

[EAWS12] Thaddeus Eze, Richard Anthony, Chris Walshaw, and Alan Soper. Autonomic computing in the first decade: trends and direction. In *ICAS 2012, The Eighth International Conference on Autonomic and Autonomous Systems*, pages 80–85, 2012.

[EB10] Moritz Eysholdt and Heiko Behrens. Xtext: implement your language faster than the quick and dirty way. In *OOPSLA*, pages 307–309. ACM, 2010.

[EC89] Bradley R. Engstrom and Peter R. Cappello. The sdef programming system. *Journal of Parallel and Distributed Computing*, 7(2):201 – 231, 1989.

[EMRU07] Roland Ewald, Carsten Maus, Arndt Rolfs, and Adelinde M. Uhrmacher. Discrete event modeling and simulation in systems biology. *Journal of Simulation*, 1(2):81–96, 2007.

[FHA99] Eric Freeman, Susanne Hupfer, and Ken Arnold. *JavaSpaces Principles, Patterns, and Practice: Principles, Patterns and Practices*. The Jini Technology Series. Addison-Wesley Longman, June 1999.

[FMDMSM+12] Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, Sara Montagna, Mirko Viroli, and Josep Lluis Arcos. Description and composition of bio-inspired design patterns: a complete overview. *Natural Computing*, May 2012. Online First.

[FMSM12] Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, and Sara Montagna. Bio-core: Bio-inspired self-organising mechanisms core. In Emma Hart, Jon Timmis, Paul Mitchell, Takadash Nakamo, Foad Dabiri, Ozgur Akan, Paolo Bellavista, Jiannong Cao, Falko Dressler, Domenico Ferrari, Mario Gerla, Hisashi Kobayashi, Sergio Palazzo, Sartaj Sahni, Xuemin (Sherman) Shen, Mircea Stan, Jia Xiaohua, Albert Zomaya, and Geoffrey Coulson, editors, *Bio-Inspired Models of Networks, Information, and Computing Systems*, volume 103 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 59–72. Springer Berlin Heidelberg, 2012.

[FSS10] Regina Frei, Giovanna Di Marzo Serugendo, and Traian-Florin Serbanuta. Ambient intelligence in self-organising assembly systems using the chemical reaction model. *Journal of Ambient Intelligence and Humanized Computing*, 1(3):163–184, 2010.

[GB00] Michael A. Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Phys. Chem. A*, 104:1876–1889, 2000.

[Gel85] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[GGG05] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using kairos. In *Distributed Computing in Sensor Systems (DCOSS)*, pages 126–140, 2005.

[Gil77] Daniel T. Gillespie. Exact stochastic simulation of coupled crfreactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, December 1977.

[Gil06] Scott F. Gilbert. *Developmental Biology, Eighth Edition*. Sinauer Associates Inc., Eighth edition, March 2006.

[GJK+04] Vitaly V. Gursky, Johannes Jaeger, Konstantin N. Kozlov, John Reinitz, and Alexander M. Samsonov. Pattern formation and nuclear divisions are uncoupled in drosophila segmentation: comparison of spatially discrete and continuous models. *Physica D: Nonlinear Phenomena*, 197(3-4):286–302, October 2004.

[GJM12] Hongliang Guo, Yaochu Jin, and Yan Meng. A morphogenetic framework for self-organized multirobot pattern formation and boundary coverage. *ACM Transactions on Autonomous and Adaptive Systems*, 7(1):15, 2012.

[GLZ06] Roberto Gorrieri, Roberto Lucchi, and Gianluigi Zavattaro. Supporting secure coordination in secspaces. *Fundam. Inform.*, 73(4):479–506, 2006.

[GMCS05] Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. Computations in space and space in computations. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 137–152. Springer Berlin Heidelberg, 2005.

[GPO13] Maria Ganzha, Marcin Paprzycki, and Andrea Omicini. Software agents: Twenty years and counting. *Computing Now*, 6(11), November 2013.

[GPOS13] Pedro Pablo González Pérez, Andrea Omicini, and Marco Sbaraglia. A biochemically-inspired coordination-based model for simulating intracellular signalling pathways. *Journal of Simulation*, 7(3):216–226, August 2013. Special Issue: Agent-based Modeling and Simulation.

[Gra59] Pierre-Paul Grassé. La reconstruction du nid et les coordinations interindividuelles chez Bellicositermes natalensis et Cubitermes sp. la théorie de la stigmergie: Essai d'interprétation du comportement des termites constructeurs. *Insectes Sociaux*, 6(1):41–80, March 1959.

[GSP11] Paul Gardner-Stephen and Swapna Palaniswamy. Serval mesh software-wifi multi model management. In *Proceedings of the 1st International Conference on Wireless Technologies for Humanitarian Relief*, ACWR '11, pages 71–77, New York, NY, USA, 2011. ACM.

[GSW06] Dina Q. Goldin, Scott A. Smolka, and Peter Wegner, editors. *Interactive Computation: The New Paradigm*. Springer, September 2006.

[GVO07] Luca Gardelli, Mirko Viroli, and Andrea Omicini. Design patterns for self-organising systems. In Hans-Dieter Burkhard, Rineke Verbrugge, and László Zsolt Varga, editors, *Multi-Agent Systems and Applications V*, volume 4696 of *LNAI*, pages 123–132. Springer, September 2007. 5th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'07), Leipzig, Germany, 25–27 September 2007, Proceedings.

[HAES09] M. Shamim Hossain, Atif Alamri, and Abdulmotaleb El Saddik. A biologically inspired framework for multimedia service management in a ubiquitous environment. *Concurrency and Computation: Practice and Experience*, 21(11):1450–1466, 2009.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[HDBDM12] Dries Harnie, Theo D'Hondt, Elisa Gonzalez Boix, and Wolfgang De Meuter. Programming urban-area applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1516–1521. ACM, 2012.

[HDF+11] Philip N. Howard, Aiden Duffy, Deen Freelon, Muzammil Hussain, Will Mari, and Marwa Mazaid. Opening closed regimes: What was the role of social media during the arab spring? 2011.

[Hen96] T.A. Henzinger. The theory of hybrid automata. In *Logic in Computer Science, 1996. LICS '96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 278–292, Jul 1996.

[HQL+11] Fei Hu, Meikang Qiu, Jiayin Li, Travis Grant, Drew Taylor, Seth McCaleb, Lee Butler, and Richard Hamner. A review on cloud computing: Design challenges in architecture and security. *CIT. Journal of Computing and Information Technology*, 19(1):25–55, 2011.

[HSG+06] Stefan Hoops, Sven Sahle, Ralph Gauges, Christine Lee, Jörgen Pahle, Natalia Simus, Mudita Singhal, Liang Xu, Pedro Mendes, and Ursula Kummer. Copasi – a complex pathway simulator. *Bioinformatics*, 22(24):3067–3074, 2006.

[HVUM07] Alexander Helleboogh, Giuseppe Vizzari, Adelinde Uhrmacher, and Fabien Michel. Modeling dynamic environments in multi-agent simulation. *Autonomous Agents and Multi-Agent Systems*, 14(1):87–116, February 2007.

[HW08] Mordechai (Muki) Haklay and Patrick Weber. Openstreetmap: User-generated street maps. *IEEE Pervasive Computing*, 7(4):12–18, October 2008.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3), 2001.

[Jaz05] Mehdi Jazayeri. Species evolve, individuals age. In Motoshi Saeki, Gerardo Canfora, and Shuichiro Yamamoto, editors, *8th International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 3–9, Lisbon, Portugal, 5–6 September 2005.

[KC09] Rick Kazman and Hong-Mei Chen. The metropolis model a new logic for development of crowdsourced systems. *Commun. ACM*, 52:76–84, July 2009.

[Kes60] H Kestelman. *Modern Theories of Integration*, chapter "Lebesgue Integral of a Non-Negative Function" and "Lebesgue Integrals of Functions Which Are Sometimes Negative." (Chapter 5-6), pages 113–160. Dover, New York, 2nd. rev. ed. edition, 1960.

[Kie02] Andrzej M. Kierzek. Stocks: Stochastic kinetic simulations of biochemical systems with gillespie algorithm. *Bioinformatics*, 18(3):470–481, 2002.

[KR08] Lila Kari and Grzegorz Rozenberg. The many facets of natural computing. *Communications of the ACM*, 51:72–83, 2008.

[LBF13] Ivan Lanese, Luca Bedogni, and Marco Di Felice. Internet of things: a process calculus approach. In *SAC*, pages 1339–1346. ACM, 2013.

[LCRP+05] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Catalin Balan. Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527, 2005.

[LDB09] William J.R. Longabaugh, Eric H. Davidson, and Hamid Bolouri. Visualization, documentation, analysis, and communication of large-scale gene regulatory networks. *Biochimica et Biophysica Acta (BBA) - Gene Regulatory Mechanisms*, 1789(4):363–374, 2009.

[LIDP10] Paola Lecca, Adaoha E. C. Ihekwaba, Lorenzo Dematté, and Corrado Priami. Stochastic simulation of the spatio-temporal dynamics of reaction-diffusion systems: the case for the bicoid gradient. *J. Integrative Bioinformatics*, 7(1), 2010.

[LK87] Frank C. H. Lin and Robert M. Keller. The gradient model load balancing method. *IEEE Trans. Softw. Eng.*, 13(1):32–38, 1987.

[LMG+09] Uichin Lee, Eugenio Magistretti, Mario Gerla, Paolo Bellavista, Pietro Lió, and Kang-Won Lee. Bio-inspired multi-agent data harvesting in a proactive urban monitoring environment. *Ad Hoc Networks*, 7:725–741, 2009.

[LMMD88] C. Lasser, J.P. Massar, J. Miney, and L. Dayton. *Starlisp Reference Manual*. Thinking Machines Corporation, 1988.

[LPF12] P. Lukowicz, S. Pentland, and A. Ferscha. From context awareness to socially aware computing. *IEEE Pervasive Computing*, 11(1):32 –41, 2012.

[LT06] Jiming Liu and Kwok Ching Tsui. Toward nature-inspired computing. *Communications of the ACM*, 49(10):59–64, October 2006.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Mac90] Bruce MacLennan. Continuous spatial automata. Technical Report Department of Computer Science Technical Report CS-90-121, University of Tennessee, Knoxville, November 1990.

[Mac08] Rebecca MacKinnon. Flatter world and thicker walls? blogs, censorship and civic discourse in china. *Public Choice*, 134(1-2):31–46, 2008.

[MCOV11] Ambra Molesini, Matteo Casadei, Andrea Omicini, and Mirko Viroli. Simulation in agent-oriented software engineering: The SODA case study. *Science of Computer Programming*, August 2011. Special Issue on Agent-oriented Design methods and Programming Techniques for Distributed Computing in Dynamic and Complex Environments.

[MFHH05] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tinydb: An acqusitional query processing system for sensor networks. In *ACM TODS*, 2005.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.

[ML12] Mieke Massink and Diego Latella. Fluid analysis of foraging ants. In *COORDINATION*, volume 7274 of *Lecture Notes in Computer Science*, pages 152–165. Springer, 2012.

[MM04] Frank Manola and Eric Miller. RDF primer. W3C recommendation, World Wide Web Consortium, February 2004.

[MMTL13] Massimo Monti, Thomas Meyer, Christian F. Tschudin, and Marco Luise. Stability and sensitivity analysis of traffic-shaping algorithms inspired by chemical engineering. *IEEE Journal on Selected Areas in Communications*, 31(6):1105–1114, 2013.

[MMTZ06] M. Mamei, R. Menezes, R. Tolksdorf, and F. Zambonelli. Case studies for self-organization in computer science. *Journal of Systems Architecture*, 52:443–460, 2006.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[MN10] Charles M. Macal and Michael J. North. Tutorial on agent-based modelling and simulation. *Journal of Simulation*, 4(3):151–162, 2010.

[MO13] Stefano Mariani and Andrea Omicini. Molecules of knowledge: Self-organisation in knowledge-intensive environments. *Intelligent Distributed Computing VI*, pages 17–22, 2013.

[MOV09] Ambra Molesini, Andrea Omicini, and Mirko Viroli. Environment in Agent-Oriented Software Engineering methodologies. *Multiagent and Grid Systems*, 5(1):37–57, 2009. Special Issue "Engineering Environments in Multi-Agent Systems".

[MPR06] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A model and middleware supporting mobility of hosts and agents. *ACM Trans. on Software Engineering and Methodology*, 15(3):279–328, 2006.

[MPV12] Sara Montagna, Danilo Pianini, and Mirko Viroli. Gradient-based self-organisation patterns of anticipative adaptation. In *Proceedings of 6th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2012)*, pages 169–174, September 2012.

[MT03] Ronaldo Menezes and Robert Tolksdorf. Adaptiveness in Linda-based coordination models. In Giovanna Di Marzo Serugendo, Anthony Karageorgos, Omer F. Rana, and Franco Zambonelli, editors, *Engineering Self-Organising Systems. Nature-Inspired Approaches to Software Engineering*, volume 2977, pages 212–232. Springer, 2003.

[MTUG09] Aneil Mallavarapu, Matthew Thomson, Benjamin Ullian, and Jeremy Gunawardena. Programming with models: modularity and abstraction provide powerful capabilities for systems biology. *Journal of the Royal Society, Interface / the Royal Society*, 6(32):257–270, March 2009.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[MV10] Sara Montagna and Mirko Viroli. A framework for modelling and simulating networks of cells. *Electronic Notes in Theoretical Computer Science*, 268:115–129, December 2010. Proceedings of the 1st International Workshop on Interactions between Computer Science and Biology (CS2Bio'10).

[MVFM+13] Sara Montagna, Mirko Viroli, Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, and Franco Zambonelli. Injecting self-organisation into pervasive service ecosystems. *Mobile Networks and Applications*, 18(3):398–412, 2013.

[MYT07] Thomas Meyer, Lidia Yamamoto, and Christian F. Tschudin. An artificial chemistry for networking. In *First Workshop on Bio-Inspired Design of Networks, BIOWIRE 2007, Cambridge, UK*, pages 45–57, 2007.

[MZ05] Marco Mamei and Franco Zambonelli. Programming stigmergic coordination with the TOTA middleware. In *4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005)*, pages 415–422, New York, NY, USA, 2005. ACM.

[MZ06a] Marco Mamei and Franco Zambonelli. *Field-Based Coordination for Pervasive Multiagent Systems. Models, Technologies, and Applications*. Springer Series in Agent Technology. Springer, March 2006.

[MZ06b] Marco Mamei and Franco Zambonelli. Self-maintained distributed tuples for field-based coordination in dynamic networks. *Concurrency and Computation: Practice and Experience*, 18(4):427–443, 2006.

[MZ07] Marco Mamei and Franco Zambonelli. Pervasive pheromone-based interaction with RFID tags. *TAAS*, 2(2), 2007.

[MZ09] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. Softw. Eng. Methodol.*, 18(4):1–56, 2009.

[MZL03] Marco Mamei, Franco Zambonelli, and Letizia Leonardi. Co-fields: Towards a unifying approach to the engineering of swarm intelligent systems. In Paolo Petta, Robert Tolksdorf, and Franco Zambonelli, editors, *Engineering Societies in the Agents World III*, volume 2577 of *LNCS*, pages 68–81. Springer, April 2003. 3rd International Workshop (ESAW 2002), Madrid, Spain, 16–17 September 2002. Revised Papers.

[Nag01] Radhika Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT, Cambridge, MA, USA, 2001.

[NFP98] Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese. Klaim: A kernel language for agents interaction and mobility. *IEEE Trans. Software Eng.*, 24(5):315–330, 1998.

[NHCV07] Michael J. North, Tom R. Howe, Nicholson T. Collier, and Jerry R. Vos. A declarative model assembly infrastructure for verification and validation. In Shingo Takahashi, David Sallach, and Juliette Rouchier, editors, *Advancing Social Simulation: The First World Congress*, pages 129–140. Springer Japan, 2007.

[NLPT14] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: The scel language. *ACM Trans. Auton. Adapt. Syst.*, 9(2):7:1–7:29, July 2014.

[NW04] Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *First International Workshop on Data Management for Sensor Networks (DMSN)*, pages 78–87, August 2004.

[NZ01] Michael S. Noble and Stoyanka Zlateva. Scientific computation with javaspaces. In *Proceedings of the 9th International Conference on High-Performance Computing and Networking*, pages 657–666. Springer-Verlag, 2001.

[OC13] Andrea Omicini and Pierluigi Contucci. Complexity and interaction: Blurring borders between physical, computational, and social systems. Preliminary notes. In Costin Bădică, Ngoc Thanh Nguyen, and Marius Brezovan, editors, *Computational Collective Intelligence. Technologies and Applications*, volume 8083 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin Heidelberg, 2013. 5th International Conference (ICCCI 2013). Craiova, Romania, 11–13 September 2013, Proceedings. Invited Paper.

[OP07] Jan Ondrus and Yves Pigneur. An assessment of nfc for future mobile payment systems. In *Proceedings of the International Conference on the Management of Mobile Business*, ICMB '07, pages 43–, Washington, DC, USA, 2007. IEEE Computer Society.

[ORAI11] Bogdan Oancea, Ion Gh. Rosca, Tudorel Andrei, and Andreea Iluzia Iacob. Evaluating java performance for linear algebra numerical computations. *Procedia CS*, 3:474–478, 2011.

[ORV05] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. An algebraic approach for modelling organisation, roles and contexts in MAS. *Applicable Algebra in Engineering, Communication and Computing*, 16(2-3):151–178, August 2005.

[ORV06] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. The multidisciplinary patterns of interaction from sciences to Computer Science. In *Interactive Computation: The New Paradigm*, pages 395–414. Springer, September 2006.

[ORV08] Andrea Omicini, Alessandro Ricci, and Mirko Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, December 2008. Special Issue on Foundations, Advanced Topics and Industrial Perspectives of Multi-Agent Systems.

[OZ99] Andrea Omicini and Franco Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, September 1999.

[Par06] H. Van Dyke Parunak. A survey of environments and mechanisms for human-human stigmergy. In Danny Weyns, H. Van Dyke Parunak, and Fabien Michel, editors, *Environments for Multi-Agent Systems II*, volume 3830 of *LNCS*, pages 163–186. Springer, 2006.

[PBS02] H. Van Dyke Parunak, Sven Brueckner, and John Sauter. Digital pheromone mechanisms for coordination of unmanned vehicles. In Cristiano Castelfranchi and W. Lewis Johnson, editors, *1st International Joint Conference on Autonomous Agents and Multiagent systems*, volume 1, pages 449–450, New York, NY, USA, 15–19July 2002. ACM.

[PCBB07] Julien Pauty, Paul Couderc, Michel Banatre, and Yolande Berbers. Geo-linda: a geometry aware distributed tuple space. In *IEEE 21st International Conference on Advanced Networking and Applications (AINA '07)*, pages 370–377, May 2007.

[PH14] T.-T.T. Pham and J.C. Ho. What are the core drivers in consumer adoption of nfc-based mobile payments?: A proposed research framework. In *Management of Engineering Technology (PICMET), 2014 Portland International Conference on*, pages 3041–3049, July 2014.

[PJRG06] Theodore J Perkins, Johannes Jaeger, John Reinitz, and Leon Glass. Reverse engineering the gap gene network of *Drosophila Melanogaster*. *PLoS Computational Biology*, 2(5):e51, 05 2006.

[PMV11] Danilo Pianini, Sara Montagna, and Mirko Viroli. A chemical inspired simulation framework for pervasive services ecosystems. In *Proceedings of the Federated Conference on Computer Science and Information Systems*, pages 667–674. IEEE Computer Society Press, 2011.

[PMV13] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with Alchemist. *Journal of Simulation*, 2013.

[Pol89] Lissa F. Pollacia. A survey of discrete event simulation and state-of-the-art discrete event languages. *SIGSIM Simul. Dig.*, 20(3):8–25, September 1989.

[PPSR09] Andrei Pisarev, Ekaterina Poustelnikova, Maria Samsonova, and John Reinitz. Flyex, the quantitative atlas on segmentation gene expression at cellular resolution. *Nucleic Acids Research*, 37(Database-Issue):560–566, 2009.

[Pri95] Corrado Priami. Stochastic pi-calculus. *Comput. J.*, 38(7):578–589, 1995.

[PRSS01] Corrado Priami, Aviv Regev, Ehud Shapiro, and William Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Information Processing Letters*, 80:25–31, October 2001.

[PVM+10] Danilo Pianini, Sascia Virruso, Ronaldo Menezes, Andrea Omicini, and Mirko Viroli. Self organization in coordination systems using a wordnet-based ontology. In *SASO*, pages 114–123, 2010.

[RCD01] Davide Rossi, Giacomo Cabri, and Enrico Denti. Tuple-based technologies for coordination. In Andrea Omicini, Franco Zambonelli, Matthias Klusch, and Robert Tolksdorf, editors, *Coordination of Internet Agents: Models, Technologies, and Applications*, chapter 4, pages 83–109. Springer-Verlag, March 2001.

[RCKZ13] Vaskar Raychoudhury, Jiannong Cao, Mohan Kumar, and Daqiang Zhang. Middleware for pervasive computing: A survey. *Pervasive and Mobile Computing*, 9(2):177 – 200, 2013.

[RHR+08] Peter Van Roy, Seif Haridi, Alexander Reinefeld, Jean-Bernard Stefany, Roland Yap, and Thierry Coupaye. Self-management for large-scale distributed systems: an overview of the selfman project. In *Formal Methods for Components and Objects, LNCS No. 5382*, pages 153–178. Springer Verlag, 2008.

[RL93] F Raimbault and D Lavenier. Relacs for systolic programming. In *Int'l Conf. on Application-Specific Array Processors*, pages 132–135, October 1993.

[RMCM04] Anand Ranganathan, Robert E. McGrath, Roy H. Campbell, and M. Dennis Mickunas. Use of ontologies in a pervasive computing environment. *Knowledge Engineering Review*, 18(3):209–220, 2004.

[ROD02] Alessandro Ricci, Andrea Omicini, and Enrico Denti. Objective vs. subjective coordination in agent-based systems: A case study. In Farhad Arbab and Carolyn L. Talcott, editors, *COORDINATION*, volume 2315 of *Lecture Notes in Computer Science*, pages 291–299. Springer, 2002.

[RPJ96] Rolando Rivera-Pomar and Herbert Jackle. From gradients to stripes in drosophila embryogenesis: filling in the gaps. *Trends in Genetics*, 12(11):478 – 483, 1996.

[RPV11] Alessandro Ricci, Michele Piunti, and Mirko Viroli. Environment programming in multi-agent systems – an artifact-based perspective. *Autonomous Agents and Multi-Agent Systems*, 23(2):158–192, September 2011.

[RS95] John Reinitz and David H. Sharp. Mechanism of eve stripe formation. *Mechanisms of Development*, 49(1-2):133–158, January 1995.

[RVO04] Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Agent coordination context: From theory to practice. In Robert Trappl, editor, *Cybernetics and Systems 2004*, volume 2, pages 618–623, Vienna, Austria, 2004. Austrian Society for Cybernetic Studies. 17th European Meeting on Cybernetics and Systems Research (EMCSR 2004), Vienna, Austria, 13–16 April 2004. Proceedings.

[SGJ07] Michael Schumacher, Laurent Grangier, and Radu Jurca. Governing environments for agent-based traffic simulations. In *Proceedings of the 5th international Central and Eastern European conference on Multi-Agent Systems and Applications V*, CEEMAS '07, pages 163–172, Berlin, Heidelberg, 2007. Springer-Verlag.

[Sha04] Nigel Shadbolt. From the Editor in Chief: Nature-Inspired Computing. *IEEE Intelligent Systems*, 19(1):2–3, January–February 2004.

[SJ07] Drew Stovall and Christine Julien. Resource discovery with evolving tuples. In *International Workshop on Engineering of Software Services for Pervasive Environments: In Conjunction with the 6th ESEC/FSE Joint Meeting*, ESSPE '07, pages 1–10, New York, NY, USA, 2007. ACM.

[SKK+08] Svetlana Surkova, David Kosman, Konstantin Kozlov, Manu, Ekaterina Myasnikova, Anastasia A. Samsonova, Alexander Spirov, Carlos E. Vanario-Alonso, Maria Samsonova, and John Reinitz. Characterization of the Drosophila segment determination morphome. *Developmental Biology*, 313(2):844 – 862, 2008.

[Skl07] Elizabeth Sklar. Netlogo, a multi-agent simulation environment. *Artificial Life*, 13(3):303–311, 2007.

[SMA05] Koushik Sen, Viswanathan M., and Gul Agha. Vesta: A statistical model-checker and analyzer for probabilistic systems. In *Quantitative Evaluation of Systems, 2005.*, pages 251–252, 2005.

[SRAA10] Norman Salazar, Juan A. Rodriguez-Aguilar, and Josep L. Arcos. Robust coordination in large convention spaces. *AI Communications*, 23(4):357–372, 2010.

[SRM+13] Daniel Schuster, Alberto Rosi, Marco Mamei, Thomas Springer, Markus Endler, and Franco Zambonelli. Pervasive social context: Taxonomy and survey. *ACM Transactions on Intelligent Systems and Technology*, 4(3), 2013.

[SRS10] Anu Singh, C. R. Ramakrishnan, and Scott A. Smolka. A process calculus for mobile ad hoc networks. *Sci. Comput. Program.*, 75(6):440–469, 2010.

[STP08] Alexander Slepoy, Aidan P. Thompson, and Steven J. Plimpton. A constant-time kinetic monte carlo algorithm for simulation of large biochemical reaction networks. *The Journal of Chemical Physics*, 128(20):205101, 2008.

[SV13] Stefano Sebastio and Andrea Vandin. MultiVeStA: Statistical model checking for discrete event simulators. In *VALUETOOLS*, 2013.

[SYD+13] Graeme Stevenson, Juan Ye, Simon Dobson, Danilo Pianini, Sara Montagna, and Mirko Viroli. Combining self-organisation, context-awareness and semantic reasoning: the case of resource discovery in opportunistic networks. In Sung Y. Shin and José Carlos Maldonado, editors, *SAC*, pages 1369–1376, Coimbra, Portugal, 18–22 March 2013. ACM.

[TB99] Guy Theraulaz and Eric Bonabeau. A brief history of stigmergy. *Artificial Life*, 5(2):97–116, Spring 1999.

[TM03] Robert Tolksdorf and Ronaldo Menezes. Using swarm intelligence in linda systems. In Andrea Omicini, Paolo Petta, and Jeremy Pitt, editors, *ESAW*, volume 3071 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 2003.

[TNN12] Michal Terepeta, Hanne Riis Nielson, and Flemming Nielson. Recursive advice for coordination. In *Coordination Models and Languages - 14th International Conference, COORDINATION 2012, Stockholm, Sweden, June 14-15, 2012. Proceedings*, volume 7274 of *Lecture Notes in Computer Science*, pages 137–151. Springer, 2012.

[TOCH14] Garry Wei-Han Tan, Keng-Boon Ooi, Siong-Choy Chong, and Teck-Soon Hew. Nfc mobile credit card: The next frontier of mobile payment? *Telemat. Inf.*, 31(2):292–307, May 2014.

[TW92] Edwin F. Taylor and John Archibald Wheeler. *Spacetime Physics: Introduction to Special Relativity*. W. H. Freeman & Company, 2nd ed. edition, 1992.

[UP05] Adelinde M. Uhrmacher and Corrado Priami. Discrete event systems specification in systems biology - a discussion of stochastic pi calculus and devs. In *Proceedings of the 37th conference on Winter simulation*, WSC '05, pages 317–326. Winter Simulation Conference, 2005.

[VB08] Cristian Versari and Nadia Busi. Efficient stochastic simulation of biological systems with multiple variable volumes. *Electronic Notes in Theoretical Computer Science*, 194(3):165–180, 2008.

[VBC11] Mirko Viroli, Jake Beal, and Matteo Casadei. Core operational semantics of Proto. In *26th Annual ACM Symposium on Applied Computing (SAC 2011)*, Tunghai University, TaiChung, Taiwan, 21–25 March 2011. ACM.

[VC09] Mirko Viroli and Matteo Casadei. Biochemical tuple spaces for self-organising coordination. In John Field and Vasco Thudichum Vasconcelos, editors, *COORDINATION*, volume 5521 of *Lecture Notes in Computer Science*, pages 143–162. Springer, 2009.

[VCMZ11] Mirko Viroli, Matteo Casadei, Sara Montagna, and Franco Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Transactions on Autonomous and Adaptive Systems*, 6(2):14:1 – 14:24, June 2011.

[VCNO10] Mirko Viroli, Matteo Casadei, Elena Nardini, and Andrea Omicini. Towards a chemical-inspired infrastructure for self-* pervasive applications. In Danny Weyns, Sam Malek, Rogério de Lemos, and Jesper Andersson, editors, *Self-Organizing Architectures*, volume 6090 of *LNCS*, chapter 8, pages 152–176. Springer, July 2010. 1st International Workshop on Self-Organizing Architectures (SOAR 2009), Cambridge, UK, 14-17 September 2009, Revised Selected and Invited Papers.

[VCO09] Mirko Viroli, Matteo Casadei, and Andrea Omicini. A framework for modelling and implementing self-organising coordination. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 1353–1360. ACM, 2009.

[VD14] Mirko Viroli and Ferruccio Damiani. A calculus of self-stabilising computational fields. In eva Kühn and Rosario Pugliese, editors, *16th Conference on Coordination Languages and Models (Coordination 2014)*, volume 8459 of *LNCS*, pages 163–178. Springer-Verlag, June 2014. Proceedings of the 16th Conference on Coordination Models and Languages (Coordination 2014), Berlin (Germany), 3-5 June.

[VDB13] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In Carlos Canal and Massimo Villari, editors, *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Science*, pages 114–128. Springer Berlin Heidelberg, 2013.

[VHR+07] Mirko Viroli, Tom Holvoet, Alessandro Ricci, Kurt Schelfthout, and Franco Zambonelli. Infrastructures for the environment of multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 14(1):49–60, July 2007.

[Vir11] Mirko Viroli. A self-organising approach to competition and composition of pervasive services. Technical Report SAPERE TR.WP2.2011.4, 2011.

[Vir13] Mirko Viroli. Engineering confluent computational fields: from functions to rewrite rules. In *Spatial Computing Workshop (SCW 2013)*, AAMAS 2013, Saint Paul, Minnesota, USA, May 2013.

[VO06] Mirko Viroli and Andrea Omicini. Coordination as a service. *Fundamenta Informaticae*, 73(4):507–534, 2006.

[VPB12] Mirko Viroli, Danilo Pianini, and Jacob Beal. Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In Marjan Sirjani, editor, *Coordination Languages and Models*, volume 7274 of *LNCS*, pages 212–229. Springer-Verlag, June 2012. Proceedings of the 14th Conference of Coordination Models and Languages (Coordination 2012), Stockholm (Sweden), 14-15 June.

[VPMS12] Mirko Viroli, Danilo Pianini, Sara Montagna, and Graeme Stevenson. Pervasive ecosystems: a coordination model based on semantic chemistry. In Sascha Ossowski, Paola Lecca, Chih-Cheng Hung, and Jiman Hong, editors, *27th Annual ACM Symposium on Applied Computing (SAC 2012)*, Riva del Garda, TN, Italy, 26–30 March 2012. ACM.

[VZ10] Mirko Viroli and Franco Zambonelli. A biochemical approach to adaptive service ecosystems. *Information Sciences*, 180(10):1876–1892, May 2010.

[VZSD12] Mirko Viroli, Franco Zambonelli, Graeme Stevenson, and Simon Dobson. From soa to pervasive service ecosystems: an approach based on semantic web technologies. In Javier Cubo and Guadalupe Ortiz, editors, *Adaptive Web Services for Modular and Reusable Software Development: Tactics and Solution*, chapter 8, pages 207–237. IGI Global, 2012.

[WBH06] Danny Weyns, Nelis Boucké, and Tom Holvoet. Gradient field-based task assignment in an agv transportation system. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 842–849, New York, NY, USA, 2006. ACM.

[WBYI08] Justin Werfel, Yaneer Bar-Yam, and Donald Ingber. Bioinspired environmental coordination in spatial computing systems. In *1st International SASO Workshop on Spatial Computing*, pages 338–343, Washington, DC, USA, 2008. IEEE Computer Society.

[Weg97] Peter Wegner. Why interaction is more powerful than algorithms. *Commun. ACM*, 40(5):80–91, May 1997.

[WLS+02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. pages 255–270, Boston, MA, December 2002.

[WM04] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[WMLF98] Peter Wyckoff, Stephen W. McLaughry, Tobin J. Lehman, and Daniel A. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[WRG+07] Z. Wu, A. Ranabahu, K. Gomadam, A.P. Sheth, and J.A. Miller. Automatic composition of semantic web services using process and data mediation. In *Proc. of the 9th Intl. Conf. on Enterprise Information Systems*, pages 453–461, 2007.

[WSBC04] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*. ACM Press, 2004.

[Yam07] Daniel Yamins. *A Theory of Local-to-Global Algorithms for One-Dimensional Spatial Multi-Agent Systems*. PhD thesis, Harvard, Cambridge, MA, USA, December 2007.

[Zam12] Franco Zambonelli. Toward sociotechnical urban superorganisms. *IEEE Computer*, 47(8), 2012.

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

[Zei76] Bernard P. Zeigler. *Theory of Modeling and Simulation*. John Wiley, 1976.

[Zei84] B. Zeigler. *Multifaceted modelling and Discrete Event Simulation*. Academic Press, 1984.

[ZO04] Franco Zambonelli and Andrea Omicini. Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems*, 9(3):253–283, 2004.

[ZOA+14] Franco Zambonelli, Andrea Omicini, Bernhard Anzengruber, Gabriella Castelli, Francesco L. DeAngelis, Giovanna Di Marzo Serugendo, Simon Dobson, Jose Luis Fernandez-Marquez, Alois Ferscha, Marco Mamei, Stefano Mariani, Ambra Molesini, Sara Montagna, Jussi Nieminen, Danilo Pianini, Matteo Risoldi, Alberto Rosi, Graeme Stevenson, Mirko Viroli, and Juan Ye. Developing pervasive multi-agent systems with nature-inspired coordination. *Pervasive and Mobile Computing*, 2014. Special Issue on "10 years of Pervasive Computing" in honor of Chatschik Bisdikian.

[ZV11] Franco Zambonelli and Mirko Viroli. A survey on nature-inspired metaphors for pervasive service ecosystems. *International Journal of Pervasive Computing and Communications*, 7(3):186–204, 2011.

List of Figures

1.1 Apple iPhone . . . 4
1.2 Smart watches . . . 5
1.3 Google Glass . . . 6
1.4 Bandwidth of international mobile telecommunications services . . . 8
1.5 iBeacon . . . 10
1.6 NFC tags . . . 11

2.1 SAPERE reference architecture . . . 21
2.2 Discrete approximation of continuous . . . 23
2.3 Layered approach for development of spatially-distributed systems via aggregate programming. . . . 27

3.1 Indexed Priority Queue extended with descendant count per branch . . . 36

4.1 ALCHEMIST computational model . . . 42
4.2 ALCHEMIST model of reaction . . . 42

5.1 ALCHEMIST architecture . . . 44

6.1 Chart showing the performance scaling of ALCHEMIST . . . 48

8.1 Analysis of a crowd steering scenario with ALCHEMIST and MULTIVESTA . . . 60
8.2 Performance scaling with of the number of running servers. . . . 62
8.3 Navigation modes in ALCHEMIST . . . 64

9.1 Drosophila Melanogaster embryo at the cleavage cycle 14A temporal class 8 . . . 69
9.2 Hierarchy of genes establishing the anterior-posterior body plan . . . 70
9.3 Gene regulatory relationships as in the model presented in [PJRG06, RPJ96, GJK+04] . . . 72
9.4 Simulation results for the Drosophila development . . . 73
9.5 Drosophila experimental data . . . 74

11.1 Meta-variables for a SAPERE algebra . . . 83
11.2 Syntax of a SAPERE algebra . . . 84
11.3 Congruence table of a SAPERE algebra . . . 84
11.4 A process algebra for SAPERE: abstracted functions . . . 85
11.5 A process algebra for SAPERE: architectural rules . . . 85
11.6 A process algebra for SAPERE: coordination primitives . . . 85
11.7 A process algebra for SAPERE: eco-laws . . . 85

12.1 Multi-agent system for creating a distributed gradient data structure . . . 92
12.2 Distributed channel . . . 94
12.3 Multi-agent system for the distributed channel . . . 95
12.4 Channels routes in a dense distributed environment . . . 96

12.5 Channel deployed in London . . . 97
12.6 Devices correctly building the channel . . . 98
12.7 Devices correctly building the channel, with a new obstacle . . . 99
12.8 Devices correctly building the channel, removing an existing obstacle . . . 100

15.1 Laws describing the museum application. . . . 109
15.2 Simulation run of indoor steering in ALCHEMIST . . . 110
15.3 Effect of crowding . . . 110
15.4 Crowd parameter . . . 111

16.1 Self-organising patterns and their relationships . . . 116
16.2 Rules for gradients . . . 121
16.3 Additional rules for anticipative gradients. . . . 123
16.4 Satisfaction values for different compositions changing over time. . . . 124

19.1 Vienna simulated in ALCHEMIST . . . 131
19.2 Crowd sensitive steering in Vienna . . . 132
19.3 Urban crowd steering effectiveness . . . 133

20.1 Grammar, Congruence and semantics of $\sigma\tau$-Linda . . . 143
20.2 Context sensitive gradient in $\sigma\tau - Linda$ . . . 146
20.3 Simulation of crowd sensitive gradient in $\sigma\tau$-Linda . . . 147
20.4 Definitions of $\sigma\tau$Linda operations. . . . 149

21.1 Table summarizing key symbols that will be used in the following discussion . . . 152
21.2 Continuous computation and space-time manifolds . . . 153
21.3 MIT Stata Center floor plan . . . 154
21.4 Example of space-time relations on a manifold . . . 155
21.5 Effective speed of information flow . . . 157
21.6 Syntax of GPI-calculus. . . . 164
21.7 Channel pattern in a WSN . . . 167
21.8 Indoor obstacle forecast . . . 169
21.9 Context-sensitive distance computation in London . . . 170
21.10 Wireless Sensor Network . . . 172
21.11 Building Alert . . . 173
21.12 Urban Traffic Steering . . . 174

23.1 Abstract syntax of Protelis . . . 180
23.2 Abstract Protelis architecture . . . 185
23.3 Rendezvous route for two people in a crowded urban environment . . . 186
23.4 Enterprise network for a small manufacturing and supply company . . . 190

A.1 Discontinuous field path integral created by `GPI` . . . 229

List of Tables

8.1 Time performance improvements reusing simulations (seconds). 63

Appendices

A
Proofs of Theorems

A.1 Discontinuity of Discrete-Valued Fields

Theorem 2. *Any field f with a discrete range and at least one event m with a neighbor m' such that $f(m) \neq f(m')$ cannot be continuous.*

Consider the neighborhood $\mathscr{N}(m)$; by assumption, the set $f(\mathscr{N}(m))$ is finite and discrete, thus all of its subsets are open. If f is continuous, then both preimages $f^{-1}(f(m))$ and $f^{-1}(f(\mathscr{N}(m)) - f(m))$ are open. Their subsets intersecting with $\mathscr{N}(m)$ must also be open, and form a partition of $\mathscr{N}(m)$. Since $\mathscr{N}(m)$ is a connected subset of a manifold M, however, the complement of an open set is closed and not open. This can only be true if one of the two sets is empty, but since there is a neighbor m' mapping to a different value than m, we have a contradiction and f cannot be continuous.

A.2 Consistency of GPI-calculus

We prove consistency of GPI-calculus in three stages: First, we prove that any finite composition of eventually consistent and continuity preserving operators is also eventually consistent and continuity preserving. We then show that each operator in GPI-calculus is individually at least eventually consistent and continuity preserving. Finally, we show that all GPI-calculus programs are finite compositions of operators, which implies that all such programs are eventually consistent.

We begin by showing that eventual consistency and eventual continuity are preserved through composition:

Lemma 3. *Consider a set of operators that are eventually consistent and where, when there is a spatial section S_M such that environment e and inputs f_i are continuous on $e^{-1}(\mathbb{V} - \mathscr{B}) \cap T^+(S_M)$ and $f_i^{-1}(\mathbb{V} - \mathscr{B}) \cap T^+(S_M)$, then there must be a spatial section S_M such that the output f_o is continuous on $f_o^{-1}(\mathbb{V} - \mathscr{B}) \cap T^+(S_M)$ Any finite composition of instances of such operators has the same properties.*

Consider a well-defined program comprised of a finite sequence of operator instances, such that the outputs of some operator instances are inputs for others. Because the sequence is finite and the program is well-defined, it must be the case that the operators instances can be ordered, such that the ith operator instance depends only on the environment and the output fields of prior operators in the sequence.

For the *i*th operator instance in this sequence, it is possible to construct an equivalent program comprising only that operator instance, an environment e_i containing its inputs, and `sense` operator instances mapping the environment values to its inputs. As noted in Lemma 8, the `sense` operators trivially satisfy the consistency and continuity conditions, so the *i*th operator instance has its input conditions satisfied. Thus, by the assumption about operator properties, we also have that this miniature program is eventually consistent and that the continuity properties hold as well.

This output can then be added to the environment for the $i+1$ operator instance, ensuring that if the preconditions are satisfied for the *i*th operator instance, they will be satisfied for the $i+1$st operator instance as well. Thus, for any finite composition of operators, the eventual consistency and continuity property stated above holds for the composition if it holds for all of the individual operators[1].

We now move on to proving that each of the operators in GPI-calculus provides sufficient consistency and continuity, beginning with the trivial case of literals:

Lemma 4. *Literals* `l` *are consistent and have a continuous output.*

Trivially true, since the output field of a literal `l` is equal to `l` at every point in its domain.

Lemma 5. *Any built-in operators* `m` *is consistent and has output f_o continuous on $f_o^{-1}(\mathbb{V}-\mathscr{B})$ if its inputs f_i are approximable and continuous on $f_i^{-1}(\mathbb{V}-\mathscr{B})$.*

The output of any `m` operator at any event m is affected only by the values of its inputs at m. `m` is also continuous by definition, and (by the semantics of field calculus) requires all inputs to have the same domain. Thus, since the composition of continuous functions is continuous, when each input f_i is continuous on $f_i^{-1}(\mathbb{V}-\mathscr{B})$, it must be the case that the output f_o is continuous on $\bigcap_i f_i^{-1}(\mathbb{V}-\mathscr{B})$. The complementary space, $\bigcup_i f_i^{-1}(\mathscr{B})$ covers only points where at least one input has value $\mathscr{B}$, and thus by the definition of `m`, f_o maps all events in this space to $\mathscr{B}$.

Only consistency remains to be shown. All ε-approximation sequences must approximate f_o on subspace $f_o^{-1}(\mathbb{V}-\mathscr{B})$ because it is continuous on this space. The complementary space $\bigcup_i f_i^{-1}(\mathscr{B})$ must also always be approximated because it holds the constant value $\mathscr{B}$ on a finite union of subspaces.

Lemma 6. *Built-in operator* `mux` *is consistent and has output f_o continuous on $f_o^{-1}(\mathbb{V}-\mathscr{B})$ if its inputs f_i are approximable and continuous on $f_i^{-1}(\mathbb{V}-\mathscr{B})$.*

If `mux` is continuous on its first input $f_1^{-1}(\mathbb{V}-\mathscr{B})$, then the preimages of $f_1^{-1}(\mathit{true})$ and $f_1^{-1}(\mathit{false})$ must both be open. The output is then defined piecewise as:

$$f_o(m) = \begin{cases} f_2(m) & m \in field_1^{-1}(\mathit{true})) \\ f_3(m) & m \in field_1^{-1}(\mathit{false})) \\ \mathscr{B} & m \in field_1^{-1}(\mathscr{B})) \end{cases}$$

Since f_2 and f_3 are continuous on their non-$\mathscr{B}$ events, that must also hold for the output field on the open subspaces selected by the preimages of *true* and *false*. Likewise, any ε-approximation sequence

[1]Note that if the program were not guaranteed finite evaluation, then this result would not hold: the eventual consistency of the *i*th instance evaluated would still hold, but no *i* would be high enough to cover the entire sequence.

that approximates both f_2 and f_3 must also approximate the output field on those subspaces, because the values on each subspace are identical to one of the two inputs that is being approximated.

Finally, the space of events mapping to $\mathscr{B}$ in the output field must always be approximated as well because it holds the constant value $\mathscr{B}$ on a subspace constructed by finite intersection and union of subspaces.

Lemma 7. *Built-in operator < is consistent and has output f_o continuous on $f_o^{-1}(\mathbb{V}-\mathscr{B})$ if its inputs f_i are approximable and continuous on $f_i^{-1}(\mathbb{V}-\mathscr{B})$.*

The < operator only outputs three values, *true*, *false*, and $\mathscr{B}$, so this theorem will be satisfied if the preimages of both *true* and *false* are open.

We must consider two conditions: real number comparison and integer comparison. For each, we will describe the case only of events that f_o maps to *true*; the *false* case follows by symmetry.

For real number comparison, consider an event m mapping to *true*. This means that in the input fields, $f_2(m)-f_1(m)=\varepsilon>0$. Because the inputs are continuous at m, There must then be some δ, such that the open set of all events m' within δ of m also have difference $f_2(m')-f_1(m')>0$, meaning they also map to true. Any union of open sets is open, so $field_o^{-1}(true)$ must be open.

Integer comparison works the same way, except that the radius δ open sets around m are guaranteed to have inputs being equal to $f_1 m$ and $f_2 m$. Thus, it is also possible in the *false* case for the two input values to be equal and still produce an open preimage.

Lemma 8. *Built-in operator `sense` is consistent and has output f_o continuous on $f_o^{-1}(\mathbb{V}-\mathscr{B})$ if its environment is approximable and continuous on $e^{-1}(\mathbb{V}-\mathscr{B})$*

The environment is assumed to be a field mapping to a tuple of values at each point, and `sense` outputs a field created by indexing into said tuples by a literal integer. A field of tuples cannot be continuous unless each of its elements is also continuous. The output is then a copy of one of those elements, and so must be continuous on the same range.

Lemma 9. *Branch operator `if` is consistent and has output f_o continuous on $f_o^{-1}(\mathbb{V}-\mathscr{B})$ if its environment e and inputs f_i are approximable and continuous on $e^{-1}(\mathbb{V}-\mathscr{B})$ and $f_i^{-1}(\mathbb{V}-\mathscr{B})$ respectively.*

The `if` operator is identical to `mux` except that its branch expressions are evaluated with respect to the subspaces $f_1^{-1}(true)$ and $f_1^{-1}(false)$ respectively (i.e., the domain of the evaluation environment is reduced). Since these are open subspaces of the environment's domain M, the approximability and continuity properties of e are not affected by the domain reduction. Thus, if the branch expressions would have conformed with the preconditions for M they will also conform with the preconditions for the branch subspaces. The output field is then assembled piecewise identically to `mux`, and is approximable and continuous on $f_o^{-1}(\mathbb{V}-\mathscr{B})$ by the same reasoning.

Lemma 10. *Function call operator `f` is eventually consistent and has some spatial section S_M such that output f_o is continuous on $f_o^{-1}(\mathbb{V}-\mathscr{B})$ if its environment e and inputs f_i are approximable and continuous on $e^{-1}(\mathbb{V}-\mathscr{B})$ and $f_i^{-1}(\mathbb{V}-\mathscr{B})$ respectively.*

Consider a function definition `f`; either `f` contains other function calls or it does not. If it does not, then evaluating `f` is equivalent to evaluating a composition of operators satisfying Lemma 3 (adjoining the

function arguments to the environment and substituting `sense` functions for variable references). Thus the desired consistency and continuity properties hold.

If `f` does contain function calls, then the properties hold if they hold for all of the function calls within `f` (meaning that once again Lemma 3 can apply). Since GPI-calculus does not allow recursion, it must be the case that these dependencies between function definitions can be arranged in a finite directed acyclic graph. The nodes of such a graph may then be ordered, such that each subsequent node only depends on nodes before it in the order. Since the set is finite, there must be at least one node that has no dependencies and thus forms a base case for induction showing that the properties hold for all function calls.

Lemma 11. *Operator `GPI` is eventually consistent and has output f_o continuous on $f_o^{-1}(\mathbb{V}-\mathscr{B})$ in the future of some spatial section S_M if its environment e and inputs f_i are approximable and continuous in the future of some spatial section S_M on $e^{-1}(\mathbb{V}-\mathscr{B})$ and $f_i^{-1}(\mathbb{V}-\mathscr{B})$ respectively.*

Following the semantics of field calculus to interpret the `GPI` algorithm given in Section 3 of the main text, the first element of the tuple computed in the `rep` statement implements a computation of distance via the triangle inequality (`nbr-range` is a metric, and a metric multiplied by a continuous positive scalar function is still a metric).

Thus, if there is a spatial section S_M such that the values of all of the inputs do not change at any device on $T^+(S_M)$, then since we assume that manifolds have finite diameter, it must be the case that all the distance estimates (first values of the `distance-integral` tuple) eventually converge to a continuous field of distance estimates. The values of the integral are co-computed with the values of the distance estimates, so they too will stop changing once the inputs have stopped changing. Thus it must be possible to choose a spatial section of S_M on which values of `distance-integral` do not change at any device.

Note also that due to the use of `min-hood'` in computing the triangle inequality, any point in the field of distances with more than one shortest path leasding to it will be replaced by a $\mathscr{B}$. This eliminates only a set of measure zero: because the distance function is continuous, its gradient cannot be discontinuous on a space of more than measure zero. The set eliminated is, however, precisely the set of points for which the gradient of the distance field is not continuous.

This is important because this is also the set of points on which the integral calculated in the second element of the `distance-integral` tuple might not be continuous. Consider, for example, the case shown in Figure A.1, in which a hole causes there to be two very different shortest paths to a point. The integrals computed along such paths may be very different indeed, necessarily creating a discontinuity in the value of the integral and hence the output of `GPI`. This type of problem can also come from other sources besides topological complexity: similar patterns (and failures) can be caused by `if` statements, distortions in the distance measure, or the shape of the source—anything that can cause a discontinuity in the gradient.

Since `min-hood'` ensures that such points are $\mathscr{B}$, however, they are eliminated from the region on which we must establish continuity. If we instead consider any m with precisely one shortest path to the `source` region, then because both integrand and the gradient of the computed distance field are continuous on this path, it must be the case that for any given ε, there must be a δ such that the open set of events within distance δ have integral values that are less than δ different, and thus the field of integrals output by `GPI` is continuous on $f_o^{-1}(\mathbb{V}-\mathscr{B})\cap T^+(S_M)$. Because it is continuous on this space, it must also be approximable; the complementary space of $\mathscr{B}$ values must also be approximable, as it is a union

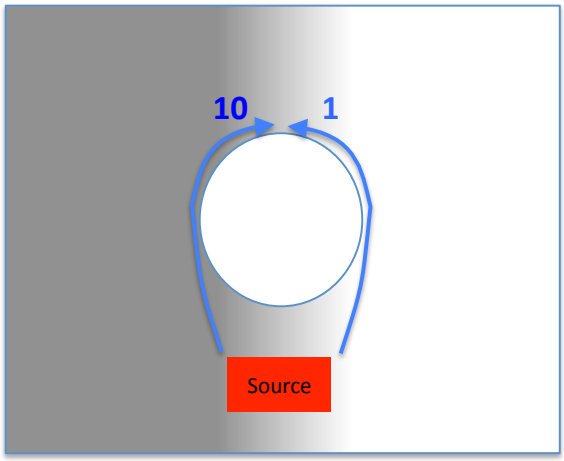


Figure A.1: The field of path integral values created by `GPI` can be discontinuous at points reached by two different shortest paths, as in this example where the left path around the hole goes through a region where the integrand is much higher than on the right path, resulting in discontinuity between a value of 10 from the left and 1 from the right.

of the $\mathscr{B}$ values of the inputs (which must be approximable) plus the measure zero set of $\mathscr{B}$ events added from computation of the distance function.

Theorem 12. *GPI-calculus programs are eventually consistent for all environments e that are continuous on $e^{-1}(\mathbb{V}-\mathscr{B})$.*

By Lemmas 4 through 11, we know that every operator in GPI-calculus has the property that if its inputs f_i and environment e are continuous on $f_i^{-1}(\mathbb{V}-\mathscr{B})$ and $e^{-1}(\mathbb{V}-\mathscr{B})$ respectively, then it is eventually consistent (all consistent programs are of course also eventually consistent). We further know that, when there is a spatial section S_M such that inputs are approximated on $T^+(S_M)$, there must also be some spatial section S_M such that the output is approximated on $T^+(S_M)$ and continuous on $T^+(S_M)\cap f_o^{-1}(\mathbb{V}-\mathscr{B})$ (note that operators continuous $f_o^{-1}(\mathbb{V}-\mathscr{B})$ are also continuous on open subsets thereof, e.g., $T^+(S_M)$).

We now consider a program as a finite sequence of operator instances. Because recursion is prohibited, it must be the case that the number of operators evaluated in the evaluation of a GPI-calculus program must be finite for any given ε-approximation. Since `if` is the only method of branching evaluations, it must thus be the case that in any approximation sequence there is some i, after which no $\varepsilon_{j>i}$-approximation evaluates an operator instance that is not also evaluated in some prior ε-approximation, considering any instance in which an operator instance is not evaluated to be an evaluation with null domain.

Given the assumed base case of an environment continuous on $e^{-1}(\mathbb{V}-\mathscr{B})$, we thus satisfy the conditions of Lemma 3 and have that the program must be eventually consistent.