ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
CAMPUS DI CESENA
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

A FOUNDATIONAL LIBRARY FOR AGGREGATE PROGRAMMING

Tesi in
Ingegneria dei Sistemi Software Adattativi Complessi

Relatore
Prof. MIRKO VIROLI
Correlatori
Dott. JACOB BEAL
Dott. DANILO PIANINI

Presentata da
MATTEO FRANCIA

ANNO ACCADEMICO 2015–2016
SESSIONE III

KEYWORDS

Aggregate programming

Programming languages

Self-organisation

Application programming interface

Simulation

To my beloved parents,
sources of inspiration

Acknowledgements

This thesis is the result of the three months I spent as a visiting researcher at the University of Iowa, a life-changing experience.

I am grateful to Mirko Viroli, who supervised me in the past six months, and to Danilo Pianini, for his continuous support while working on this dissertation. Special thanks to "Jake" Beal, who supervised me during the time I spent at the University of Iowa.

I wish to thank the people with whom I shared the APICe laboratory, for the fun we had together.

I would like to show my deepest gratitude to my family, especially to my parents, for encouraging me to achieve my life goals, and to Laura, for the life we share together.

Matteo Francia. March 16, 2017

Contents

Sommario

L'elevata diffusione di entità computazionali ha contribuito alla costruzione di sistemi distribuiti fortemente eterogenei. L'ingegnerizzazione di sistemi auto-organizzanti, incentrata sull'interazione tra singoli dispositivi, è intrinsecamente complessa, poiché i dettagli di basso livello, come la comunicazione e l'efficienza, condizionano il design del sistema. Una pletora di nuovi linguaggi e tecnologie consente di progettare e di coordinare il comportamento collettivo di tali sistemi, astraendone i singoli componenti. In tale gruppo rientra il *field calculus*, il quale modella i sistemi distribuiti in termini di composizione e manipolazione di *field*, "mappe" dispositivo-valore variabili nel tempo, attraverso quattro operatori sufficientemente generici e semplici al fine di rendere universale il modello e di consentire la verifica di proprietà formali, come la stabilizzazione di sistemi auto-organizzanti. L'*aggregate programming*, ponendo le sue fondamenta nel field calculus, utilizza field computazionali per garantire elasticità, scalabilità e composizione di servizi distribuiti tramite, ad esempio, il linguaggio Protelis. Questa tesi contribuisce alla creazione di una libreria Protelis per l'aggregate programming, attraverso la creazione di interfacce di programmazione (*API*) adatte all'ingegnerizzazione di sistemi auto-organizzanti con crescente complessità. La libreria raccoglie, all'interno di un unico *framework*, algoritmi tra loro eterogenei e *meta-pattern* per la coordinazione di entità computazionali. Lo sviluppo della libreria richiede la progettazione di un ambiente minimale di *testing* e pone nuove sfide nella definizione di *unit* e *regression* testing in ambienti auto-organizzanti. L'efficienza e l'espressività del lavoro proposto sono testate e valutate empiricamente attraverso la simulazione di scenari *pervasive computing* a larga scala.

Abstract

Ubiquitous computing has led to the creation of highly-heterogeneous distributed systems. Engineering these systems is challenging, particularly in mapping from collective specifications to the behaviour of individual devices. Researchers have developed new technologies and DSLs that abstract from the device-to-device interaction. Among them, field calculus models distributed systems in terms of composition and manipulation of fields—a mapping from a device to an arbitrary value—by means of four general constructs also tractable for formal analysis. Aggregate programming, leveraging field calculus, engineers self-organising systems using the field abstraction to provide inherent guarantees of resilience, scalability, and safe composition (e.g., via the Protelis Java-hosted language). However, field calculus operators are too low-level for pragmatic use in complex systems development. We thus present a prototype API intended to raise the level of abstraction and thereby provide an accessible and user-friendly interface for the construction of complex resilient distributed systems. In particular, we have systematically organised in a unified framework a large and heterogeneous collection of algorithms and usage patterns, including methods for common tasks such as leader election, distance estimation, collection of distributed values, and gossip-based information dissemination. This library is tested against a new testing framework for the aggregate programming, rising new challenges such as defining what unit and regression testing are in the field of self-organising systems. We illustrate the efficacy and expressiveness of this library through scenarios of large-scale pervasive computing and their empirical evaluation in simulation.

Chapter 1

Introduction

Pervasive computing has led to the creation of complex and heterogeneous distributed systems [1, 2]. Computation has become cheap enough to embed computing devices (FPGAs, micro-controllers, etc.) in any aspect of our lives. Yet a wide gap lies between the requirements of fully-distributed systems and their design. A programmer knows what the aggregate behaviour of the system should be, but its design, implementation and debug are significant challenges to overcome.

The need of global-to-local[1] compilation strategy has been recognised before [3]. Indeed, the design of aggregate distributed systems should make coordination implicit, enable a modular and unanticipated composition of heterogeneous services, and leverage heterogeneous coordination mechanisms to address applications with different space-time abstractions [4].

Researchers have deployed new DSLs (domain specific languages) to describe aggregate behaviours in a number of models, systems, and technologies across many different fields. Despite the heterogeneity of these prior approaches and the problems they aim to address, from a software engineering perspective they tend to cluster into five main classes of approach [3]: making device interaction implicit (e.g., TOTA [5], MPI [6], NetLogo [7], Hood [8]), providing means to compose geometric and topological constructions (e.g.,

[1]Compiling aggregate specifications into actions and interactions of individual devices.

Origami Shape Language [9], Growing Point Language [10], ASCAPE [11]), and providing generalisable constructs for space-time computing (e.g., Protelis [12], Proto [13], MGS [14]).

It is from this last, and particularly Proto, that field calculus and the aggregate programming approach derive, aiming at a generalisation that can effectively encompass the vast majority of the above approaches, as will be generally necessary for building complex distributed systems across a wide range of application domains. Aggregate programming is a paradigm that models large-scale adaptive systems, shifting the focus from the perspective of a single component to the whole aggregate. Its layered approach (Figure 1.1) addresses IoT (Internet of Things) systems [15] with five abstraction layers.

With a bottom-up view: *(i)* Each device has its own capabilities. Communication and device-to-device interaction are extremely heterogeneous and might require ad-hoc considerations that should not impact on the design of the aggregate. *(ii)* Field calculus [16] is a prominent approach to aggregate programming that addresses distributed systems as a functional composition of computational fields. Being a terse model, field calculus allows one to formally verify properties such as self-stabilisation, guaranteeing that feed-forward compositions of its constructs maintain the same properties. *(iii)* Core calculi are too low-level for system construction [17], allowing one to write unsafe and fragile programs. Field calculus constructs are then combined into resilient[2] coordination operators ("building block" algorithms) [18] leveraged to organise adaptive systems around well known coordination and state-tracking patterns. The analysis of self-organising systems suggests three basic mechanisms needed to ground complex applications: diffusion of information in the network, aggregation of distributed information, and "evaporation" of information [19]. *(iv)* User-friendly APIs (application programming interfaces) compose building blocks into functions which encapsulate general complex mechanisms. *(v)* Application code imports APIs and built-in functions to fulfil the system requirements: an aggregate program is

[2]Providing the ability to adapt to unexpected changes in working conditions.

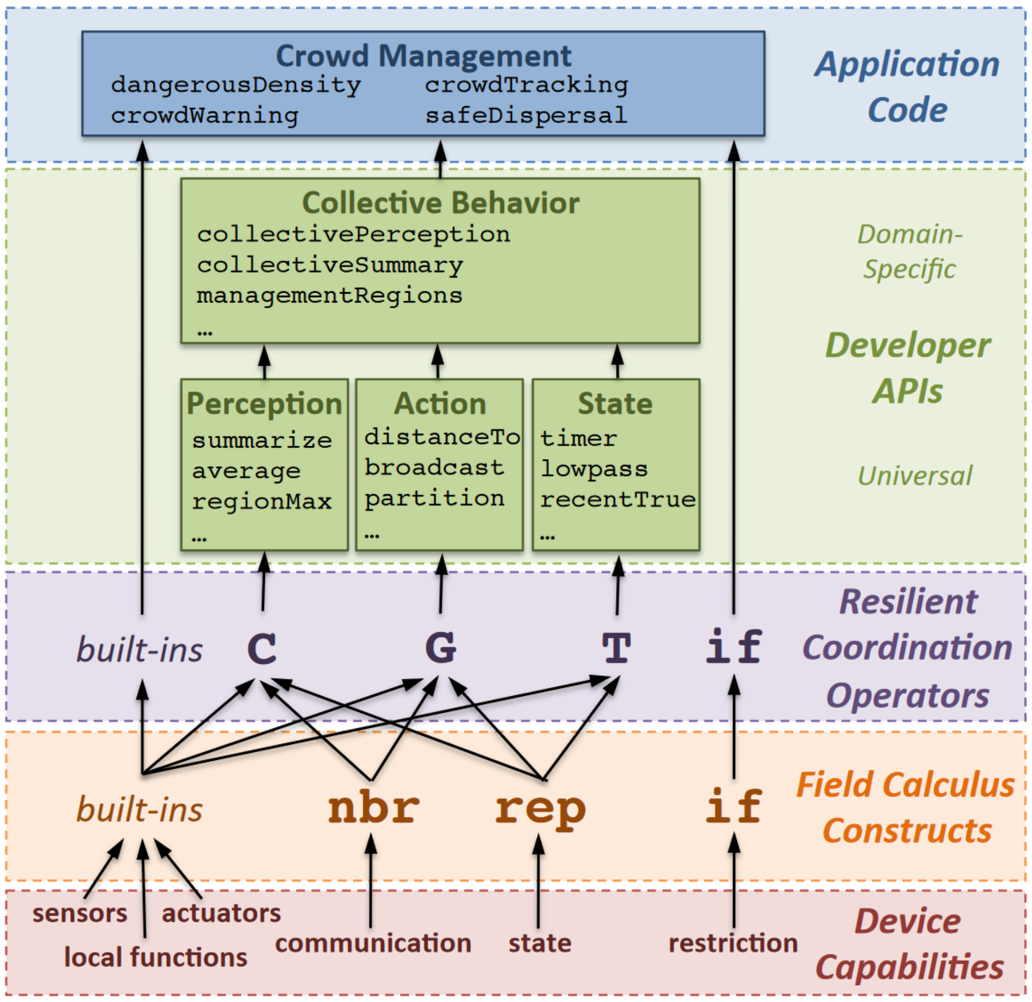


Figure 1.1: The layered approach of aggregate computing shifts the focus from the single device perspective to a cooperative collection of devices. Software and hardware capabilities of devices are leveraged to implement aggregate-level field calculus constructs. "Building block" algorithms with provable resilience properties combine these constructs, and are further combined to deploy user-friendly APIs for a fully-resilient coordination of IoT systems. Adapted from [2].

a manipulation of data constructs across a region (either a discrete network or continuous space).

This dissertation contributes with the creation of `protelis-lang`, a new foundational library for self-organising distributed systems deployed within the Protelis framework. The Protelis language [12] implements the semantic of field calculus. Its highly-extensible functional approach addresses the terseness of field calculus, allowing an incremental deployment of building block algorithms and APIs for a fully-resilient coordination of distributed systems. In particular, `protelis-lang` captures and systematically organises in a unified framework a large and heterogeneous collection of algorithms and usage patterns, including methods for common tasks such as leader election, distance estimation, collection of distributed values, and gossip-based information dissemination.

The remainder of this dissertation is organised as follows. Chapter 2 reviews the vision of the aggregate programming paradigm and field calculus as universal approach to aggregate programming, and then provides an overview of the Protelis language for programming aggregate systems. Chapter 3 describes how `protelis-lang` is engineered around bio-inspired patterns from [19] and their implementation as building blocks. Chapter 4 extends the workflow proposed in [20] and describes how a testing framework affects the deployment of both aggregate algorithms and `protelis-lang`, showing how unit testing and performance evaluation can be carried out. In Chapter 5 the efficacy and expressiveness of this library are illustrated through scenarios of large-scale pervasive computing and their empirical evaluation in simulation.

Chapter 2

Background

This section summarises the layered architecture of aggregate programming, starting with its vision and motivation. Field calculus is a universal approach to aggregate programming. Despite its universality, field calculus is still too terse to be leveraged as an actual engineering tool for complex systems. The Java-based Protelis language implements field calculus and represents a step forward to address aggregate programming challenges. New user-friendly APIs can be built leveraging Protelis, hence dramatically reducing the abstraction gap between requirements and design of aggregate systems.

2.1 Motivation

Computational entities pervade the environment, highlighting the need for new paradigms and coordination strategies to govern a collection of devices. Pervasive computing and smart cities both envision a future in which interconnected devices will "augment" everyday life. Infrastructural, personal and wearable "smart" devices will lead to a pervasive continuum—a distributed and very dense substrate of devices—that hosts services to manage aspects of our lives. Device-to-device interaction will be context-dependent and adaptive to unexpected contingencies.

Programming and managing such complex distributed systems is challenging and is subject of ongoing investigation in contexts such as cyberphysical systems, pervasive computing, robotic systems, and large-scale wireless sensor networks [21]. Distributed systems raise challenges such as robustness to faults, adaptiveness to changes in network topology and the native openness of pervasive scenarios. These challenges require flexible and dynamical deployment of code to devices across the network, to adaptively change its scope of execution, and to predictably integrate it with the existing services. Researchers have recognised the need of new paradigms to engineer these collective systems and, particularly, to coordinate complex and distributed activities [18]. Aggregate programming models distributed systems in terms of their aggregate behaviours, e.g., "distribute a dispersal signal in overcrowded areas at mass events," and encapsulates self-organisation techniques to guarantee a fully-resilient and robust coordination of complex services. Effective models and programming languages are needed to handle distributed systems as native aggregates of devices [16], hence diverging from a device-centric perspective: devices are instead perceived as a single collective entity. Languages approaching aggregate programming should then provide both mechanisms for writing aggregate specifications and a global-to-local mapping to translate them into means of coordination of individual devices.

Several approaches to aggregate programming model the entire system as dynamically evolving fields [3]. Among them, field calculus is a universal approach to aggregate programming in which fields are first-class abstractions leveraged to model sensors, state, and results of computation. The operational semantics of the field calculus produce a unified model supporting self-organisation and code mobility. Fields of first-class functions allow the distribution of code by means of device interactions and higher-order calculus.

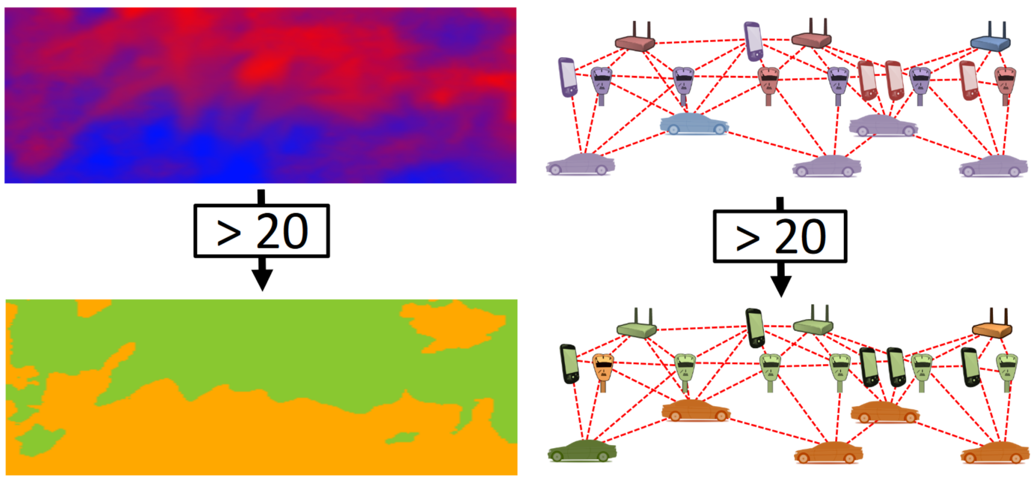


Figure 2.1: Manipulation of continuous and discrete computational fields. Fields of temperatures are transformed into boolean fields. Adapted from [22].

2.2 Field calculus

Many approaches based on computation over continuous space and time have addressed the aggregate programming challenge: [3] identifies more than 100 significant languages and models for spatial computers. Spatial computing covers a wide range of applications and biases in their approach to computational models. Examples of spatial computers include sensor networks, robot swarms, mobile ad-hoc networks, etc. Several of these leverage coordination models and languages (e.g., TOTA [23], Linda [24]) through which diffusion, recombination, and composition of information scattered in the system produce global, dynamically evolving computational fields. Formally, a computational field generalise the notion of field in physics [16]: "a computational field is a map from every computational device in a space to an arbitrary computational object" (Figure 2.1). Fields are aggregate-level distributed data structures that gradually adapt to changes in the underlying topology and interaction with the environment. Fields can be composed to implement self-organising coordination patterns.

Field calculus captures and formalises abstractions of existing coordination languages or models targeting aggregate programming. This "core calculus" approach captures semantics in a tiny language, expressive enough to be universal [25] yet tractable for mathematical analysis. Historically, field calculus is inspired by Proto [13]. They both express aggregate behaviour by a functional composition of operators that manipulate continuous fields and predict the behaviour of computational fields from underlying interactions between individual devices. These manipulations are compiled into local rules iteratively executed in asynchronous "computation rounds:" each device receives messages from its neighbours, computes the local value of fields, and finally spreads the result of this computation to its neighbours[1]. Both behaviour (computation and state of a device) and interaction (message content) are modelled as annotated evaluation trees. Field construction, propagation, and restriction are then supported by local evaluation "against" the evaluation trees received from neighbours.

The higher-order extension of field calculus (HFC) embeds first-class functions, allowing the handling of functions just like any other value. Code can be dynamically injected, moved, and executed in network domains: functions can be fed with other functions as arguments, return new functions, and be assigned to variables. HFC also supports anonymous functions and code migration—the functionalities being executed by a device can change over time. Together, first-class functions (what to compute) and domain-restriction (where to compute) allow predictable and safe composition of robust self-organisation mechanisms [21].

The syntax of higher-order field calculus is depicted in Figure 2.2. Five constructs are composed into programs using a Lisp-like syntax.

- Built-in function call `(b` $\mathtt{e}_1$ `...` $\mathtt{e}_n$`)`: "point-wise" operations involving neither state nor communication. The built-in function `b` is applied to its $\mathtt{e}_1$, ..., $\mathtt{e}_n$ input fields, and its output field maps each device to the result of a local computation, e.g., mathematical functions (e.g.,

[1]The definition of "neighbourhood" and communication between neighbours are abstracted away by field calculus.

$$\begin{array}{lcll}
\mathtt{l} & ::= & \mathtt{c}\langle\overline{\mathtt{l}}\rangle \mid \lambda & \text{;; Local value}\\
\lambda & ::= & \mathtt{b} \mid \mathtt{f} \mid (\mathtt{fun}\ (\overline{\mathtt{x}})\ \mathtt{e}) & \text{;; Function value}\\
\mathtt{e} & ::= & \mathtt{l} \mid \mathtt{x} \mid (\mathtt{e}\ \overline{\mathtt{e}}) & \text{;; Expression}\\
 & \mid & (\mathtt{rep}\ \mathtt{x}\ \mathtt{w}\ \mathtt{e}) & \\
 & \mid & (\mathtt{nbr}\ \mathtt{e}) & \\
 & \mid & (\mathtt{if}\ \mathtt{e}\ \mathtt{e}\ \mathtt{e}) & \\
\mathtt{w} & ::= & \mathtt{x} \mid \mathtt{l} & \text{;; Variable or local value}\\
\mathtt{F} & ::= & (\mathtt{def}\ \mathtt{f}(\overline{\mathtt{x}})\ \mathtt{e}) & \text{;; Function declaration}\\
\mathtt{P} & ::= & \overline{\mathtt{F}}\ \mathtt{e} & \text{;; Program}
\end{array}$$

Figure 2.2: Syntax of higher order field calculus. Adapted from [21].

`(add 1 2)`) and context-dependent operators (e.g., `(uid)` returns the device UID).

- Function definition and function call: abstraction and recursion are supported by function definition $\mathtt{def\ f(x_1,\ \ldots,\ x_n)\ e}$, where $\mathtt{x}_i$ are formal arguments and `e` is the function body. $\mathtt{(f\ e_1\ \ldots\ e_n)}$ applies `f` to `n` input fields.

- Time evolution $\mathtt{(rep\ x\ e_0\ e)}$: the "repeat" construct supports stateful evolving fields. `rep` initialises the state variable `x` to $\mathtt{e}_0$, then updates it computing `e` against the previous value of `x`.

- Neighbourhood field construction `(nbr e)`: `nbr` encapsules interaction between devices, and returns a field that maps each neighbouring device to its most recent available value of field `e`. Such "neighbouring" fields can then be manipulated and summarised with built-in `*-hood` operators, e.g., `(min-hood (nbr e))` outputs a field mapping each device to the minimum value of `e` amongst its neighbours.

- Domain restriction $\mathtt{(if\ e_0\ e_1\ e_2)}$: branching is implemented by this construct, which computes $\mathtt{e}_1$ in the restricted domain where $\mathtt{e}_0$ is `true`,

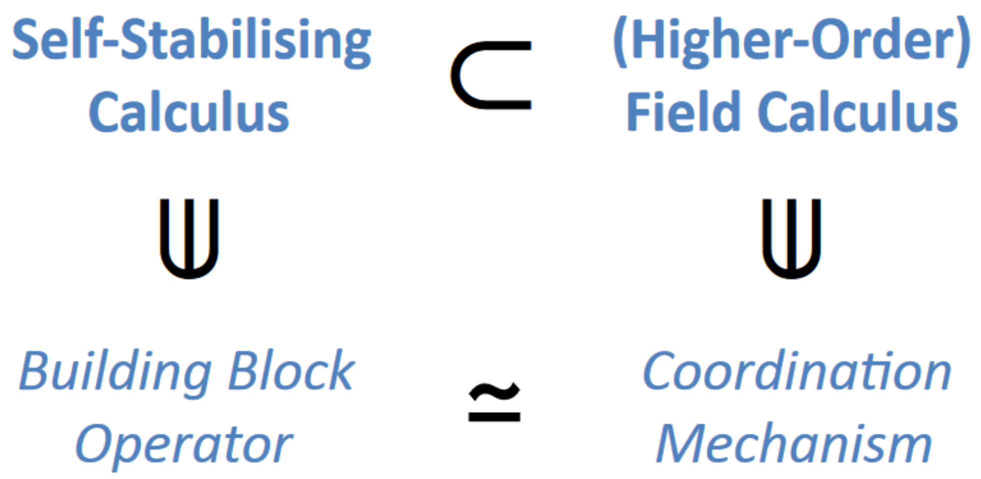


Figure 2.3: Every coordination mechanism can be expressed in field calculus, but many may be difficult or impossible to express within its guaranteed self-stabilising subset. If one coordination mechanism is asymptotically equivalent to another mechanism in the self-stabilising subset, however, then it is guaranteed to be safely composable as well. Adapted from [20].

and $\mathtt{e}_2$ in the restricted domain where $\mathtt{e}_0$ is `false`.

A field calculus program is then interpretable either as an aggregate-level computation on fields or as an equivalent "compiled" version automatically generated with local interaction rules. Each program consists of a set of function definitions and a main expression $\mathtt{e}_M$ evaluated within a network of interconnected devices. When a device fires, it computes $\mathtt{e}_M$ and outputs its "value tree:," a tree which tracks the results of the sub-expressions encountered during the evaluation. Each evaluation on a device is performed against the most recently received value-trees of its neighbours, and the produced value-tree is conversely made available to the device's neighbours at the end of the computational round.

2.2.1 Self-stabilising field calculus

Under fixed environmental conditions K, a network in state N is stable if no device firing changes its state. A network in state N self-stabilises to a stable state N_0 iff through a sufficiently long fair[2] sequence of transitions it necessarily reaches N_0 and remains there. If the network self-stabilises to N_0, then it does so to a globally unique state that is unequivocally determined by the environmental conditions K (namely, is independent of N). Then, N_0 can be interpreted as the output of the aggregate computation [20]. Hence, a program P is self-stabilising if its field computations react to and recover from any change in environmental conditions, eventually reaching N_0 regardless any current state N.

In the context of an open system, self-stabilisation entails that any sub-expression of the aggregate program can be associated to a final and stable field, reached in finite time while adapting to changes in the underlying environment. This acts as the sought bridge between the sub-expressions in program code and the emergent global outcome[3] [26].

Self-stabilisation is generally undecidable, given that computational rounds are not even guaranteed to terminate due to the universality of local computation. Thus, ensuring self-stabilisation is a matter of isolating fragments of the calculus that produce only self-stabilising field expressions. This problem has been addressed in [20] which also describes a self-stabilising sub-language of field calculus.

2.2.2 Substitution principle

Because of their generality, some coordination algorithms may not achieve good dynamic performance. More performing mechanisms exist but may be

[2]Assumption under which devices fire at almost the same frequency, assuring that no device is perceived as disconnected by its neighbours.

[3]While engineering a field calculus program, we can reason in terms of the field each expression stabilises to, rather than the expression itself. The global result is eventually a manipulation of self-stabilised fields.

difficult to express in the self-stabilising calculus (Figure 2.3).

The "substitution principle" extends the properties of self-stabilising calculus: "given functions λ, λ' with same type, λ is substitutable for λ' iff for any self-stabilising list of expressions `e`, $(\lambda\ e)$ always self-stabilises to the same value as $(\lambda'\ e)$" [20]. Namely, two self-stabilising functions are "substitutable" if they eventually converge to the same stable state N_0 when fed with the same inputs. Indeed, as self-stabilisation does not consider the transients of these functions, as long as the converged values are the same, the two functions are swappable without affecting self-stabilisation. A coordination mechanism with desirable dynamic properties can replace another one, improving overall performance.

2.2.3 Resilience

Resilience is the ability to adapt to unexpected changes in working conditions [27], ensuring that the system achieves its goals in spite of certain classes of change (e.g., density of devices, perturbations). The aggregate computing framework (Figure 1.1) should provide *inherent* resilience: adaptation to changes is detracted from the responsibilities of an aggregate program and is demanded to the underlying layers.

Self-stabilisation guarantees resilience only to occasional disruption. Field evolution reaches the stable state N_0 only if there is enough time following the last perturbation. However, even small perturbations to the network topology can significantly affect the result of computation.

[20, 28] propose two approaches to ensure resilience to ongoing perturbations: *(i)* the former presents an engineering methodology to replace coordination mechanisms with alternative and more specialised implementations that can better trade off speed with adaptiveness in certain contexts of usage (described in Chapter 4); *(ii)* the latter turns gossiping into a self-stabilising process by means of running parallel replicas of gossiping.

2.3 Protelis: an aggregate programming language

The key idea of aggregate computing is to consider computational fields as a first-class abstraction: any computation is therefore seen as a purely functional transformation of fields. The computational field calculus [29, 30] provides a universal [25] formal foundation for this approach (syntax, typing, denotational and operational semantics, behaviour properties such as self-stabilisation). Being a theoretical model, any implementation of field calculus requires both an interpreter and an architecture to handle communication, execution, and interfacing with external components. In addition, a framework for field calculus should be portable across both simulation environments and real networked devices.

The Protelis language [12] has been developed as an implementation of field calculus similarly to the Proto VM [31]. A parser translates Protelis code into a field calculus semantics that is executed by the interpreter at regular intervals, communicating with other devices and drawing contextual information from environment variables implemented as a tuple of key-value pairs. Protelis includes the universality and self-stabilisation properties of field calculus [29] in a modern programming language with the following features [12]: *(i)* a functional paradigm with an imperative Java-like syntax, which significantly reduces barriers to adoption; *(ii)* full interoperability with the Java runtime and API; *(iii)* complete coverage of the field calculus constructs; and *(iv)* higher-order mechanisms to enhance reusability and flexibility, and to support code mobility.

Embedding Protelis within Java ensures accessibility, portability, and ease of integration. Java reflection allows dynamic invocation of arbitrary Java code, thus allowing integration of aggregate programs with a rich existing ecosystem of libraries, devices, and applications. Still, Protelis is overall a purely functional language: a program is made of a set of function definitions (essentially, libraries formed by modules) with a main expression as starting

$$\begin{array}{lll}
\texttt{P} ::= \overline{\texttt{I}}\ \overline{\texttt{F}}\ \overline{\texttt{s}}; & & \text{;; Program} \\
\texttt{I} ::= \texttt{import m} \mid \texttt{import m.*} & & \text{;; Protelis/Java import} \\
\texttt{F} ::= \texttt{def f}(\overline{\texttt{x}})\ \{\overline{\texttt{s}};\} & & \text{;; Function definition} \\
\texttt{s} ::= \texttt{e} \mid \texttt{let x = e} \mid \texttt{x = e} & & \text{;; Statement} \\
\texttt{w} ::= \texttt{x} \mid \texttt{l} \mid [\overline{\texttt{w}}] \mid \texttt{f} \mid (\overline{\texttt{x}})\texttt{->}\{\overline{\texttt{s}};\} & & \text{;; Variable/Value} \\
\texttt{e} ::= \texttt{w} & & \text{;; Expression} \\
\quad \mid \texttt{b}(\overline{\texttt{e}}) \mid \texttt{f}(\overline{\texttt{e}}) \mid \texttt{e.apply}(\overline{\texttt{e}}) & & \text{;; Fun/Op Calls} \\
\quad \mid \texttt{e.m}(\overline{\texttt{e}}) & & \text{;; Method Calls} \\
\quad \mid \texttt{rep(x<-w)}\{\overline{\texttt{s}};\} & & \text{;; Persistent state} \\
\quad \mid \texttt{if}(\texttt{e})\{\overline{\texttt{s}};\}\ \texttt{else}\ \{\overline{\texttt{s}}';\} & & \text{;; Exclusive branch} \\
\quad \mid \texttt{mux}(\texttt{e})\{\overline{\texttt{s}};\}\ \texttt{else}\ \{\overline{\texttt{s}}';\} & & \text{;; Inclusive branch} \\
\quad \mid \texttt{nbr}\{\overline{\texttt{s}};\} & & \text{;; Neighborhood values}
\end{array}$$

Figure 2.4: Protelis abstract syntax. Adapted from [12].

point of the global computation—ultimately carried on by the collection of available devices undergoing repetitive computational rounds.

The abstract syntax of Protelis is shown in Figure 2.4—overbar semi-formal notation is used to denote sequences of syntactic elements. In Protelis, any expression denotes a whole computational field (a space-time data structure), hence functions compute fields out of fields. Local values `l` (numbers, strings, Booleans), tuples (e.g., `[1, 2, "s"]`), function names `f`, and anonymous functions (`(`$\overline{\texttt{x}}$`)->{e}`), all represent "constant" fields (mapping to the same value at every device at every time). Each statement is an expression to be evaluated, and a statement sequence $\overline{\texttt{s}}$ evaluates to the result of the last statement. Expressions are variables, values, and function calls (applied to a built-in function `b`, a user-defined function `f`, a Java method `e.m`, and—by `apply` notation—an expression `e` returning a function itself).

Protelis implements the field-calculus constructs (introduced in Section 2.2) to address space-time computation, and adds `mux` as an additional branching operator:

- `rep(x<-w){`$\overline{\texttt{s}}$`;}`, illustrated in Figure 2.5(a), defines a time-varying field that is initially `w`, and is continuously updated at each round by the unary function taking variable `x` and evaluating body $\bar{s}$ (e.g., `rep(x<-0){x + 1}` is the field counting computation rounds at each device);

- `nbr{e}`, illustrated in Figure 2.5(b), models device-to-device interaction and creates a field where each device maps its neighbours (including itself) to their latest available evaluation of `e` (such fields are then usually reduced again with a built-in `hood` functions like max, min, average, and so on, as described below);

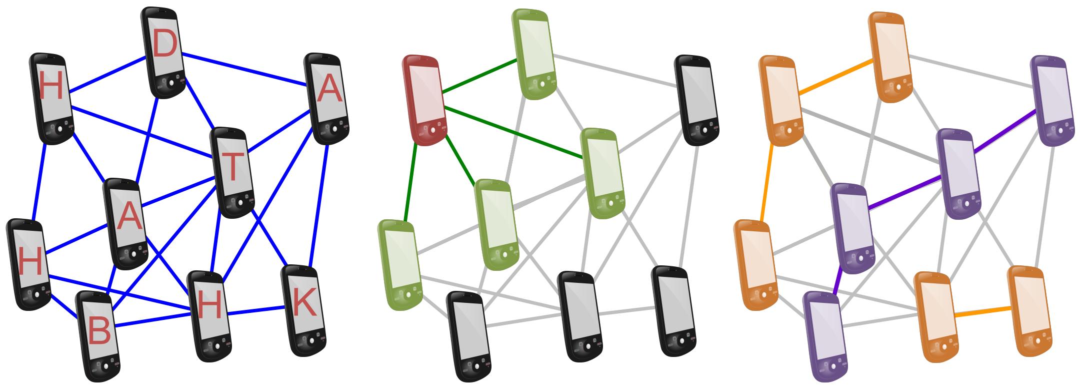
- `if(e){`$\bar{s}$`;} else {`$\bar{s}'$`;}`, illustrated in Figure 2.5(c), performs an exclusive branch, partitioning the network into two space-time subregions (where `e` evaluates to `true`/`false`, respectively), computing $\bar{s}$ in the former and $\bar{s}'$ in the latter, in isolation;

- `mux(e){`$\bar{s}$`;} else {`$\bar{s}'$`;}` construct is an inclusive multiplexing branch: the two fields obtained by computing $\bar{s}$ and $\bar{s}'$ are superimposed, using the former where `e` evaluates to `true`, and the second where `e` evaluates to `false`.

Listing 2.1 shows a composition of field calculus operators into higher-level and more user-friendly functions written in Protelis.

2.4 Building blocks

Though field calculus is a step toward practical applications, as shown in Figure 1.1, its terseness makes it too low-level for programming resilient systems [17]. The "Resilient coordination" layer composes field calculus constructs in a collection of higher-level building block algorithms, simple and generalised basis elements of an "algebra" of programs with desirable resilience properties [12]. Four self-stabilising algorithms (Figure 2.6) are introduced in [18] and elaborated in [20]: `G` (spreading), `C` (aggregation), `T` (temporary state),

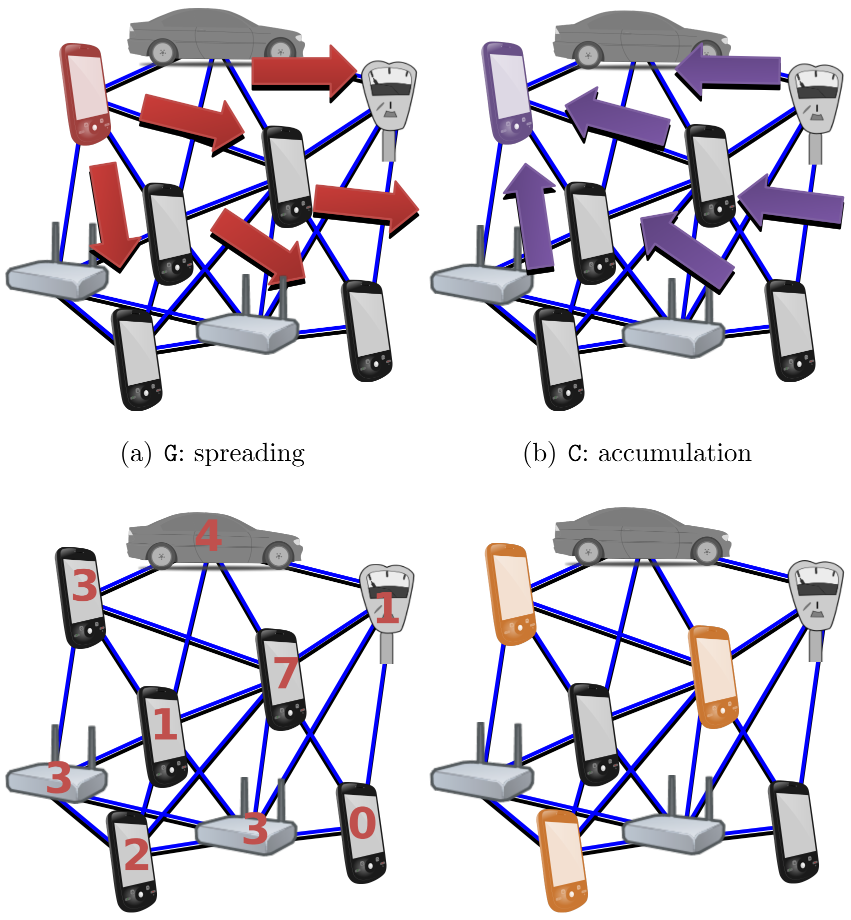


(a) `rep`: time-varying field (b) `nbr`: field from neighbours (c) `if`: exclusive branching

Figure 2.5: Field calculus core operators. Adapted from [17].

```
/* Estimating distance to a source region */
def distanceTo(source) {
  /* Time-varying field: minimum distance to source */
  rep(d <- Infinity) {
    /* Inclusive multiplexing: all devices evaluate both branches.
     * Returns 0 if source is true, a positive distance otherwise */
    mux (source) {
        0
    } else {
        minHood(nbr{d} + nbrRange)
    }
  }
}
/* Estimating distance to a source region while avoiding obstacles */
def distanceToWithObstacle(source, obstacle) {
  /* Exclusive branching: evaluate distanceTo if obstacle is false,
   * evaluate Infinity otherwise */
  if (obstacle) {
      Infinity
  } else {
      distanceTo(source)
  }
}
```

Listing 2.1: Leveraging `rep`, `nbr`, `if`, `mux` to build higher-level and user-friendly functions. Adapted from [12].



(a) G: spreading (b) C: accumulation

(c) T: time (d) S: sparse choice

Figure 2.6: Self-organising distributed systems are often built on bio-inspired mechanisms, summarised in [19], such as diffusion and accumulation of information over space and time, and symmetry breaking through mutual inhibition. The four building-block algorithms proposed in [18] and refined in [20] address these mechanisms. Their self-stabilisation and eventual consistency—adaptiveness to changes in device density, topology, etc.—have been formally proved in [20, 22], and transfer to any feed forward composition of these blocks. Reproduced from [17].

and S (symmetry breaking through mutual inhibition). Critically, any program constructed using only these operators for coordination and state (or their equivalents, per the modular proof established in [20]) is guaranteed to be self-stabilising and to have good scaling properties (though timing details of course vary depending on usage details). "Feed-forward" compositions of such algorithms are self-stabilising too. Indeed, if the input of an algorithm is stable, then its output self-stabilises as well. Any algorithm consuming its output will be fed with an input that stops changing, and eventually all algorithms will self-stabilise.

Building blocks are highly general, allowing a programmer to build and coordinate distributed application across different applications. Each of them captures best practises to develop flexible decentralised specifics, hiding the low-level details of field calculus and satisfying three features: *(i) self-stabilisation*, building blocks eventually converge and, in addition, they are capable of reactively adjusting to changes in the input field and network structure; *(ii) scalability* to large networks; *(iii) resiliency*, compositions of building blocks inherit their resilient features.

2.4.1 Spreading

G, illustrated in Figure 2.6(a), produces resilient diffusion of information away from a source region, spreading this information outward along a spanning tree built applying the triangle inequality constraint, and possibly modifying that information as it spreads. In the case of multiple sources, the space is effectively partitioned into sub-regions, one per source, with each device receiving information only from its nearest source. Many functions addressing information diffusion can be based on G, e.g., estimating distance to one or more designated source devices and broadcasting a value from a source (Listing 2.2).

```
/* source: from where information is spread
 * init: field spread from source
 * metric: how to estimate distance
 * accumulate: how to accumulate information ascending along the gradient
 */
def G(source, init, metric, accumulate) {
  rep (distanceValue <- [Infinity, init]) {
    mux (source) { [0, init] }
    else {
      let ndv = nbr(distanceValue);
      minHood([ndv.get(0) + metric.apply(), accumulate.apply(ndv.get(1))])
    }
  }.get(1)
}
/* Distance to source */
def distanceTo(source) {
  /* nbrRange: field of distances of current device to its neighbours */
  distanceToWithMetric(source, nbrRange)
}
def distanceToWithMetric(source, metric) {
  G(source, 0, metric, (v) -> {v + metric.apply()})
}
/* Spread value from a source */
def broadcast(source, value) {
  G(source, value, () -> {nbrRange}, (v) -> {v})
}
```

Listing 2.2: G implementation and examples of G-related functions. Adapted from [18].

```
/* potential: gradient descended to aggregate information
 * accumulate: how to aggregate information descending along the gradient
 * local: local value
 * null: null value in case of no neighbours
 */
def C(potential, accumulate, local, null) {
  rep (v <- local) {
    reduce.apply(local,
      hood(
        /* built-in operator to reduce a field to a value */
        (acc, element) -> { reduce.apply(acc, element) }, null,
        /* build a spanning tree descending along a potential */
        mux (nbr(getParent(potential)) == self) { nbr(v) }
        else { null }
    ))
  }
}
/* Count devices in a given region */
def countDevices(sink) {
  C(distanceTo(sink), (acc, elem) -> {acc + elem}, 1, 0)
}
/* Estimate the average value in a network */
def average(sink, value) {
  C(distanceTo(sink), (acc, elem)->{acc+elem}, value, 0)/countDevices(sink)
}
/* Share a global agreement on scattered data */
def summarize(sink, accumulate, local, null) {
  broadcast(sink, C(distanceTo(sink), accumulate, local, null))
}
```

Listing 2.3: C implementation and examples of C-related functions. Adapted from [18].

G: implementation details

Each device keeps track of the `distanceValue` which is initialised to the tuple `[Infinity, init]`. Its first element represents the actual shortest distance to the closest source, while the latter is the accumulated value. Devices evaluate both branches of `mux`. While the former is a point-wise expression, the latter maps the neighbours of each device to their most recent available value of `distanceValue`. This value is updated applying the `accumulate` function to the sum of the field of distances to the closest source and the field of distances to each device's neighbours. The resulting field is finally reduced to the tuple containing the shortest distance to the source region. Devices in which `source` is `true` are the origins of the gradient, and return `[0, init]`.

2.4.2 Accumulation

`C`, illustrated in Figure 2.6(b) is the complement of `G`, addressing resilient aggregation of information across space: `C` is fed with what to aggregate, then performs that aggregation along a spanning tree down a potential gradient towards a source device that thus eventually reduces all the information scattered through a region into a single summary value (Listing 2.3).

C: implementation details

Each device keeps track of the value `v` initialised to `local`, and then aggregates values scattered along the network. The `hood` built-in function reduces a field to a single value iteratively applying `reduce` to pairs of elements. `acc` is initialised to the first element of the field, and is updated to the result of each aggregation. In this case `hood` is fed with: the `reduce` function, a `null` value, and a field containing the latest available value of `v` for the neighbours whose parent[4] is the current device and `null` for the others.

[4]Neighbour with the lowest potential.

```
/* length: time period
 * zero: value representing zero state
 * decay: how to decrease time
 */
def T(length, zero, decay) {
  rep (leftTime <- length) {
    min(length, max(zero, decay.apply(leftTime)))
  }
}
/* Return true once every length time */
def cyclicTimerWithDecay(length, decay) {
  rep (leftTime <- length) {
    if (leftTime == 0) {
      length
    } else {
      T(length, 0, (t) -> {t - decay})
    }
  } == length
}
```

Listing 2.4: `T` implementation and examples of `T`-related functions. Adapted from [18].

```
/* grain: component size
 * metric: unit measure of grain
 */
def S(grain, metric) {
  breakUsingUids(randomUid(), grain, metric)
}
/* Return the tuple [random number, device UID] */
def randomUid() {
  rep (identifier <- [self.nextRandomDouble(), self.getDeviceUID()]) {
      identifier
  }
}
/* Break network symmetry */
def breakUsingUids(uid, grain, metric) {
  rep (lead <- uid) {
    distanceCompetition(distanceToWithMetric(uid == lead, metric),
        lead, uid, grain, metric)
  } == uid
}
/* Compete to find the leader */
def distanceCompetition(d, lead, uid, grain, metric) {
  mux (d > grain) { uid }
  else {
    let thr = 0.5 * grain;
    mux (d >= thr) { [Infinity, Infinity] }
    else {
      minHood(
        mux (nbr(d) + metric.apply() >= thr) { [Infinity, Infinity] }
        else { nbr(lead) }
      )
    }
  }
}
```

Listing 2.5: `S` implementation. Adapted from [18].

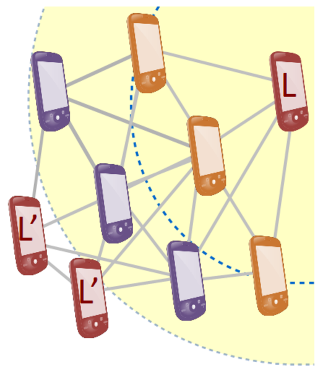


Figure 2.7: Distance competition in `S` from the perspective of leader `L`. `L` is the current leader of the yellow component with a `grain` radius. Orange devices, located within `grain/2`, compete with `L`, while purple devices, more distant than `grain/2` from `L`, are ignored during the competition. Devices `L'`, more distant than `grain` from `L`, propose themselves as leaders of a new partition.

2.4.3 Time

`T`, illustrated in Figure 2.6(c) implements a flexible timer, which progresses from some initial state to a "zero" state at a potentially time-varying state (Listing 2.4).

`T`: implementation details

Each device keeps track of the `leftTime` value which is initialised to the `length` of the timer, and then returns the minimum between `length` and the maximum between `zero` and the value returned applying the `decay` function to the remaining time.

2.4.4 Sparse choice

`S`, illustrated in Figure 2.6(d), breaks symmetry through mutual inhibition, in which devices compete against others to become leaders, generating a random Voronoi partition with a characteristic `grain` size (Listing 2.5).

`S`: implementation details

Each device is identified by a tuple containing a random value and its universal identifier. `S` breaks symmetry creating random partitions whose leaders are the devices identified by the minimum random value within a `grain` range. In `distanceCompetition` (Figure 2.7), devices compete against the others, and propose themselves as leaders in case they are farther than `grain` from the nearest `leader`. Otherwise, devices compute the minimum value of the field of leaders' identifiers in which devices more distant than `grain/2` from the closest leader are excluded from the competition (their identifiers are replaced with the tuple `[Infinity, Infinity]`). `breakUsingUids` (hence `S`) then returns `true` if `distanceCompetition` elects the current device as a leader, `false` otherwise. As result, `S` elects a set of leaders such that no device is more distant than `grain` from a leader, and no leaders are closer than `grain/2`.

2.5 Towards an aggregate library

The layered architecture of aggregate programming allows a hierarchical composition of best-effort practices, encapsulating even complex coordination mechanisms in a single higher-level component (e.g., a function). Building blocks represent the foundation of more complex APIs (Figure 1.1) that considerably reduce the abstraction gap between application requirements and system design. Functions such as `distanceTo` (Listing 2.2), `average`, `summarize` (Listing 2.3), etc., can be leveraged as higher-level "primitives" of new libraries for aggregate programming.

This approach is suitable for large-scale scenarios such as crowd applications that involve people at mass public events (marathon, city festival, etc.). Personal and infrastructural devices communicate and coordinate with the others, and their opportunistic interaction smoothly supports services such as: crowd detection, crowd-aware navigation, dispersal advice, etc. A distributed service is encapsulated and managed as a single function. Program-

```
/* Estimating crowd density */
def dangerousDensity(p, r) {
  let mr = managementRegions(r*2, () -> { nbrRange });
  let danger = average(mr, densityEst(p, r)) > 2.17
    && summarize(mr, sum, 1 / p, 0) > 300;
  if(danger) {
    high
  } else {
    low
  }
}
/* Check if an area has been exposed at overcrowd risk in a given time */
def crowdTracking(p, r, t) {
  let crowdRgn = recentTrue(densityEst(p, r) > 1.08, t);
  if(crowdRgn) {
    dangerousDensity(p, r)
  } else {
    none
  };
}
/* Disseminate warning signals */
def crowdWarning(p, r, warn, t) {
  distanceTo(crowdTracking(p,r,t) == high) < warn
}
```

Listing 2.6: Example of small crowd application that supports crowd estimation and warning dissemination in Protelis. Adapted from [2].

mers compose modules to build the desired application specifying where each service should be executed and how information flows between them. For instance, a crowd estimation service maps information about the location of a device to a crowd density computational field. This serves as an input for crowd-aware navigation, which outputs vectors of recommended travel and warnings that are in turn an input for producing dispersal advice.

Due to the high concentration of people in constrained areas, dangerous overcrowding issues emerge possibly leading to injuries [2]. Estimating crowd density, performing an overcrowding risk assessment and disseminating warnings are achieved with few lines of Protelis code (Listing 2.6). The proposed program is resilient and adaptive, enabling it to effectively estimate crowd density (`none, low, high`) and distribute warnings while executing on numerous mobile devices. Functions introduced before are composed here to show how a crowd-management library can be built on top them.

Chapter 3

`Protelis-Lang` library

"*There is no code without project, no project without problem analysis, no problem analysis without requirements.*"

Antonio Natali

In this section, we engineer `protelis-lang` on the bases of a multi-stage process including requirements analysis, problem analysis, and design. Outputs from each stage are qualitatively represented in Figure 3.1.

3.1 Requirements

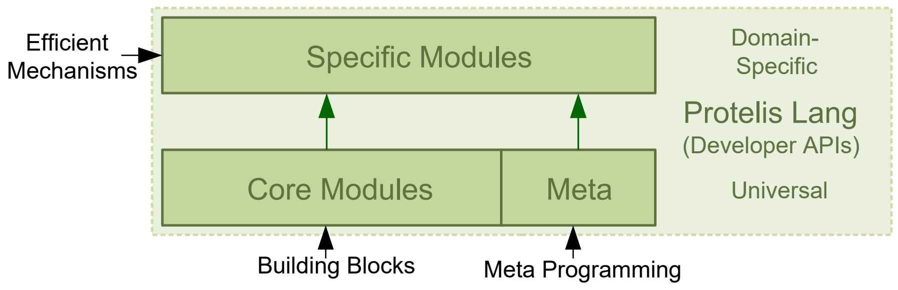
While a programmer may have clear ideas about the aggregate behaviours desired from a system, the details required to implement such behaviours in the low-level operations of field calculus (or their corresponding Protelis syntax) are often quite intricate and sensitive to details of their implementation. This is particularly true for ensuring that behaviours are resilient and scalable, as these properties are often quite difficult to validate using either formal analysis or empirical testing [17]. In order to fulfil the promise of aggregate programming, we need a comprehensive library that provides higher-level building blocks that are already guaranteed to be resilient and scalable, thus insulating application programmers from these challenges.

The terseness of field calculus allows a programmer to design fragile programs and requires one to "manually" address coordination and state-tracking challenges. Building block algorithms partially reduce the complexity of writing aggregate programs, still, being general functions they require specialisation. Though building blocks hide resilient best practices "under the hood," avoiding directly use of field calculus constructs, there is a considerable gap to bridge with respect to application requirements. The `protelis-lang` library is located at the "Developer APIs" layer of the aggregate programming stack (Figure 1.1), and, as such, our work should define extensible and general core modules on which more specific ones can be built.

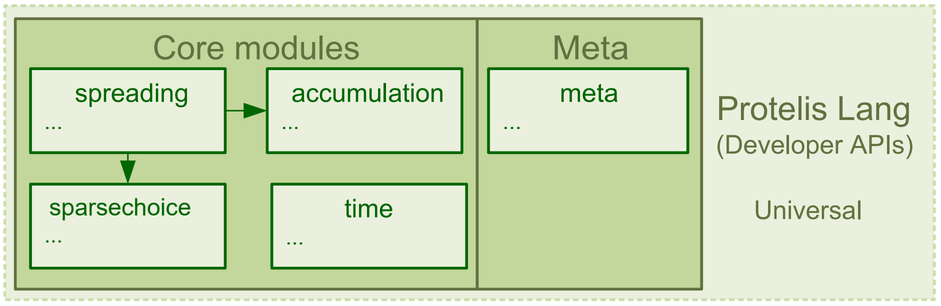
While analysing the requirements, we highlighted three key ideas: building blocks, resiliency and scalability. All of them were previously addressed in Chapter 2. As such, we defined a two-stage process to engineer our library. The first challenge is defining, organising, and formalising what has already been published. In the second stage, we defined the boundaries of `protelis-lang` through a comparison with the state-of-art literature and fill in the missing places with respect to the existing technologies and DSLs.

3.2 Analysis

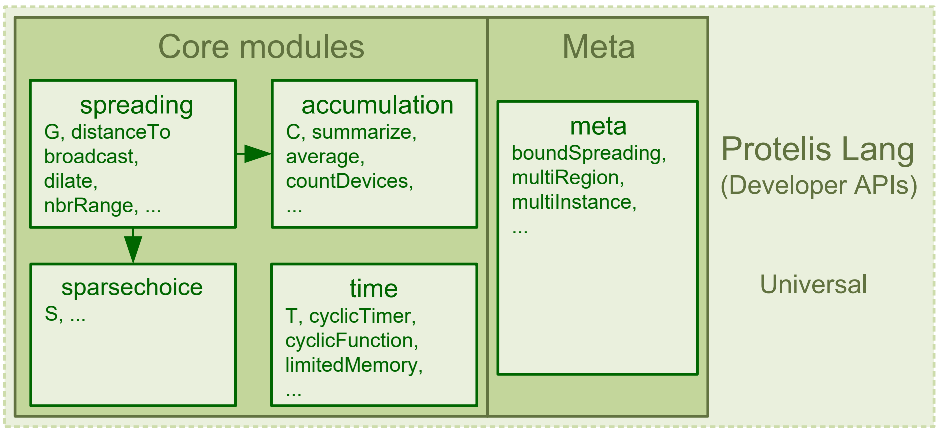
In order to construct our prototype library, we drew upon three sources in an effort to more systematically define its scope and populate its contents. At the core of the library is the system of four self-stabilising building-block operators identified in [18] and elaborated in [20]: `G` (spreading), `C` (aggregation), `T` (temporary state), and `S` (sparse choice). Critically, any program constructed using only these operators for coordination and state (or their equivalents) is guaranteed to be self-stabilising and to have good scaling properties (though timing details of course vary depending on usage details). We then searched through all of the prior publications referenced for their as-



(a) We need a comprehensive and highly extensible library providing higher-level building blocks, guaranteeing inherent resilience, and which is organised around domain-specific modules that depend on global modules (core modules and meta-programming modules).



(b) Self-organising systems are built around spreading, and accumulation of information over space and time [3, 19]. `accumulation` and `sparsechoice` depend on `spreading` as they require metrics and distance-based potentials. We added `sparsechoice` module for symmetry breaking.



(c) Modules are populated with building-block related functions.

Figure 3.1: Engineering the "Developer APIs" layer from Figure 1.1. We deployed the core modules of `protelis-lang` following three stages: (a) requirement analysis, (b) problem analysis and (c) design of `protelis-lang`. Green arrows depict inter-module dependencies.

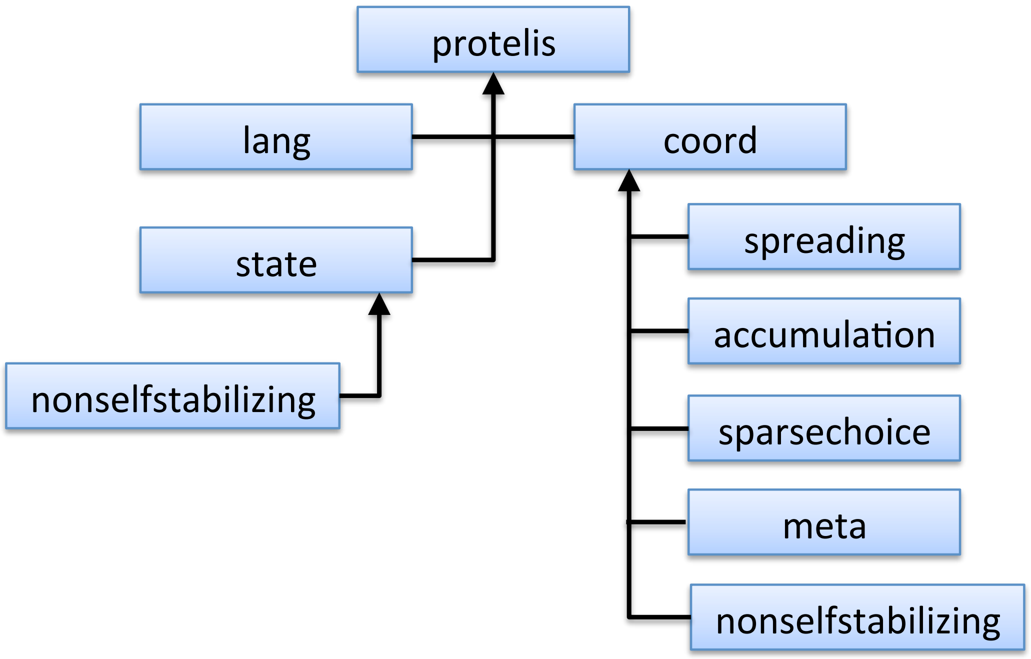


Figure 3.2: Organisation of `protelis-lang` library: `spreading`, `accumulation`, `state` and `sparsechoice` are built around the four resilient "building block" operators described in [18] and illustrated in Figure 2.6, while `meta` contains higher-order coordination patterns and `lang` contains utility functions. The `nonselfstabilizing` sub-packages expose additional less resilient building blocks that must be applied with care for that reason.

sociated algorithms and code fragments, importing and adapting those that could be mapped onto one or more of these operators or proved equivalent, along with any other patterns and supporting functions of interest.

Finally, we compared the contents thus identified in two broad surveys: the systematic analysis of space-oriented aggregate programming languages in [3] and the analysis of biologically-inspired self-organisation patterns in [19] in order to more systematically define our scope of coverage and to identify and fill any coverage gaps that we could identify.

In particular, at this time we specifically rule out of scope of the library algorithms that control device movement (e.g., flocking and swarming) or that

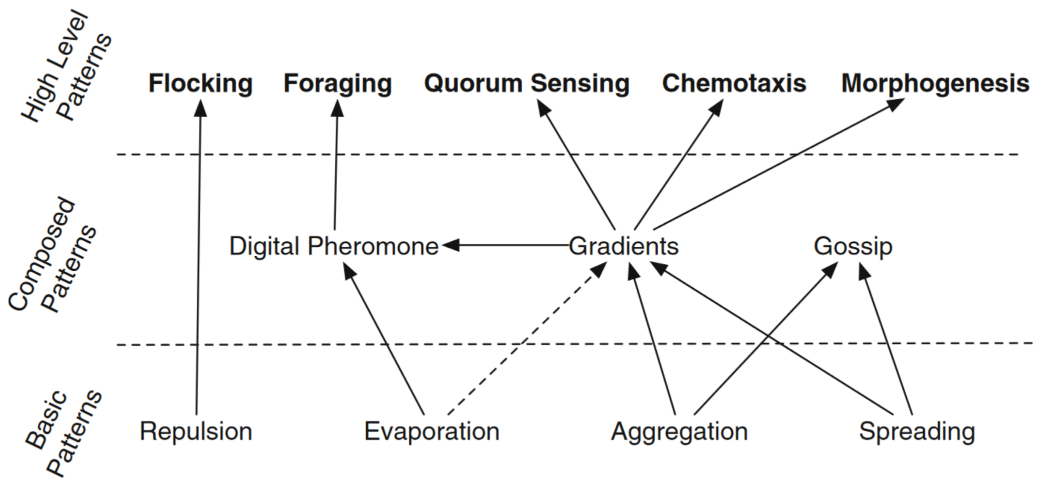


Figure 3.3: Classification of and inter-relations between bio-inspired mechanisms to ease the engineering of self-organising artificial systems. Adapted from [19].

work with coordinate information (e.g., localisation algorithms). Analysis of self-organisation patterns suggests three basic mechanisms needed to ground complex applications [19]: diffusion of information in the network as an advertisement mechanism, aggregation of distributed information as a sensing mechanism, and "evaporation" of information as a refresh mechanism. As *spreading*, *aggregation* and *evaporation* are foundational mechanisms upon which more complex patterns are built (Figure 3.3), we drew parallels with `G`, `C` and `T`. Encouragingly, we found that the building-block operators correspond nicely with these three bio-inspired mechanisms.

`G` draws from both *aggregation* and *spreading* patterns, diffusing and aggregating information ascending along a gradient. `G` is also related to the *gradient* pattern as it allows diffusion of information along with the distance from the source (Listing 3.1). However, as `G` partitions a network of devices in "isolated" spatial sub-regions, `G` does not allow the aggregation of gradients propagated by different sources. Finally, `G` represents a possible implementation of the *ant foraging pattern* in which the origin of the gradient and devices represent respectively nest and ants looking for food.

Both `C` and *aggregation* pattern merge, filter, and reduce information scattered in the system, avoiding network and memory overload. While `C` aggregates data down a potential, *aggregation* locally applies a fusion operator to process and synthesises macro information.

Gossip composes both *aggregation* and *spreading* patterns to eventually share a global agreement about local values scattered in the system. `summarize` (Listing 2.3), a feed forward composition of `C` and `G`, is a possible self-stabilising implementation of *gossip* that aggregates information in a sink and then broadcasts it back. The same composition can be leveraged to implement the *quorum sensing* pattern for taking collective decisions in systems where a minimum number of devices satisfying a certain condition is required.

As information might become outdated, functions should rely on more recent data. *Evaporation* pattern periodically reduces the relevance of data over time, and `T` can be leveraged to address this pattern (Listing 3.3).

The analogies between building blocks and the basic bio-inspired patterns suggest a possible organisation for our library around `G`, `C`, `T`, and `S`—though symmetry breaking is not addressed in [19], and prove that functionalities furnished by `protelis-lang` can potentially address a wide range of applications.

3.3 Design

While software development is immune from almost all physical laws, entropy—the degree of disorder—hits hard [32]. Well-designed code considerably affects the productivity of system designers and programmers, especially when it comes to working with state-of-art technologies. According to Dave Thomas [33]: "Clean code can be read, and enhanced by a developer other than its original author. It has unit and acceptance **tests**. It has meaningful **names**. It provides one way rather than **many ways** for doing one thing. It has minimal dependencies, which are explicitly defined, and provides a clear and **minimal API**."

Testing

The three test driven development laws guarantee that minimal tests virtually cover all the library code [34]. To manage testing complexity, we use Gradle as a tool for the automatic build of our library, JUnit as a testing framework, and we structured tests as follows:

- Test cases should be short and descriptive. Testing requires the creation of a new simulation environment, which we kept as simple as possible. In particular, testing an aggregate program, addressed in Chapter 4, requires the scenario configuration against which Protelis code is tested. They both cover only the strictly necessary specifications to test the desired feature;
- Every test function has one and only one concept assertion;
- Test names should be short and descriptive. Each function within the library must be tested and may require multiple unit and regression tests. In this case, test name should be a concatenation of "function name + feature being tested."

Naming conventions

The name of a variable or a function should be as expressive as possible, explaining what it does, and how it is used. If a name requires a comment, then the name does not reveal its intent [33]. For instance, the function definition `def ebf(s) /* exponential back-off filter */` may lead to disinformation. We prefer descriptive pronounceable names, such as `def exponentialBackoffFilter(signal)`. Indeed, the programmer, at first sight, can expect what the `exponentialBackoffFilter` function does, still, at a later stage, she should check the unit tests as they formalise the behaviour of the function.

The Protelis language does not support types and function overloading. As such, we adopted internal naming conventions and introduced a Javadoc-compliant documentation[1] describing each function signature. For instance, `distanceTo` (Listing 2.2) feeds `G` with the `self.nbrRange()` metric. As such the field of distances returned by `distanceTo` strictly depends on the back-end implementation of the Protelis `ExecutionContext`. We also wanted a generic version of this function to deal with ad-hoc metrics. Thus we introduced a new function whose name is the concatenation of "base function name + `With` + additional parameters," resulting in `distanceToWithMetric`. `distanceTo` then performs an inner call to the generic `distanceToWithMetric` that takes the desired metric as a parameter.

```
/* ----- Function signatures, type is described after each parameter ---- */
/**
 * Compute distance to a source.
 *
 * @param source bool, whether the current device is a source
 * @return       num, distance to the closest source
 */
def distanceTo(source) { ... }
/**
 * Compute distance to a source.
 *
 * @param source bool, whether the current device is a source
 * @param metric ()->num, how to estimate distances
 * @return       num, distance to the closest source
 */
def distanceToWithMetric(source, metric) { ... }
/**
 * Distance to neighbors.
 *
 * @return num, field of distances from each neighor
 */
def nbrRange() { self.nbrRange() }
/**
 * Hop to neighbors.
 *
 * @return num, field of ‘1‘s (hop to each neighor)
 */
def nbrRangeHopDistance() { 1 }

/* ----- Function invokations ---- */
distanceTo(source) /* distance based on ExecutionContext implementation */
distanceToWithMetric(source, nbrRange) /* same as above */
distanceToWithMetric(source, nbrRangeHopDistance) /* hops to the source */
```

[1]Documentation is omitted in the other listings because of space issues.

General but minimal API

Being a generally scoped library, `protelis-lang` may allow multiple solutions to certain problems. For instance, broadcasting the number of devices in a network:

```
broadcast(sink, countDevices(distanceTo(sink))); /* broadcast devices */
summarize(sink, sum, 1, 0); /* broadcast devices */
```

Still, the core functionalities should be minimal enough to avoid redundancy. Exceptions to this rule exist in case of new functions with sensible differences in the dynamic features are added to `protelis-lang` extending the substitution library (Chapter 4). For instance, we ended up having several substitutable functions, such as `G-crfGradient-flexGradient` and `C-CMultiIdempotent-CMultiDivisible`.

Functions as first-class abstractions

The statements within the body of a function should be all at the same level of abstraction. Mixing different levels within a function is always confusing for code readers [33]. While reading a Protelis program, the main function should be followed by those at the next level of abstraction, so that we can read the program descending one level of abstraction at a time as we read down the list of functions (step-down rule [33]).

We now consider two abstraction levels:

- *Field calculus constructs.*

  ```
  def distanceTo(source) {
    rep (accumulator <- Infinity) {
      mux (source) { 0 }
      else { minHood(self.nbrRange() + nbr(accumulator)) }
    }
  }
  ```

  The translation of the previous code in natural language is: `distanceTo` keeps track of an `accumulator` initialised to `Infinity`. If a device is a source, then it fires `0`, otherwise it fires the minimum value of the sum of two fields: distances to its neighbours, and neighbours' values of `accumulator`.

- *Specialising building blocks.*

```
def sum(sink, value) {
  C(distanceTo(sink), (a, b) -> { a + b }), value, 0)
}
```

  Translating this snippet into natural language is straightforward: `C` accumulates the sum of devices' `value` descending along the shortest path to the `sink`. This code does not include any field calculus constructs: the higher the abstraction level, the easier the explanation in natural language.

- *Mixing abstraction levels.*

```
def broadcastSumMixedAbstraction(sink, value) {
  rep (accumulator <- value) {
    mux(sink) { sum(sink, value) }
    else { maxHood PlusSelf(nbr(accumulator)) }
  }
}
```

  One the one hand, mixing different abstraction levels makes things hardly understandable. Each device keeps track of an `accumulator` initialised to `value`. All devices compute `sum`, but if a device is a `sink`, then it fires the actual sum, else it fires the minimum value from the field of neighbours' `accumulator`.

- *Same abstraction level.*

```
def broadcastSumSameAbstraction(sink, value) {
  broadcast(sink, sum(sink, value))
}
```

  On the other hand, using the same abstraction level dramatically simplifies the understanding of the code. Indeed, `broadcastSumSameAbstraction` and `broadcastSumMixedAbstraction` behave the same, but the former is easily understandable: `sink` devices `broadcast` the sum of the values accumulated in them.

Functions should avoid side effects and have a small number of arguments. Arguments are hard to test, as they increase the number of input combinations to be tested. Testing every combination of appropriate values can be daunting. When a function requires too many parameters (namely, more than three), it is likely that some of those arguments ought to be wrapped into a function or aggregated in a single value [33].

```
def broadcastSum4(sink, region, obstacle, value) {
   broadcast(sink,
     if (region && !obstacle) { sum(sink, value) }
     else { 0 }
   )
}
def broadcastSum3(sink, condition, value) { /* merge region and obstacle */
  broadcast(sink, sum(sink, if (condition) { value } else { 0 }))
}
def broadcastSum(sink, value) { /* merge condition and value */
  broadcast(sink, sum(sink, value))
}
```

`broadcastSum4` addresses two tasks: broadcasting information and filtering obstacles and devices which are not part of `region`. An orthogonal[2] design promotes reuse. If functions have specific, well-defined responsibilities, they can be combined with new components in ways that were not envisioned before [32]. Functions should be loosely coupled, self-contained, and with a clear scope. Modular code reduces maintenance issues, reducing development and testing, promoting reuse—the more loosely coupled the functions, the easier they are to reconfigure and reengineer, isolating "bad" code. `broadcastSum3` partially optimises `broadcastSum4`, aggregating the two boolean parameters. However, this function is still responsible for the two tasks. `broadcastSum` gets rid of the filtering functionality addressing only broadcasting of a generic `value`.

We provided multiple versions of the same functions with an increasing generality level (again, `distanceTo` and `distanceToWithMetric`), trying to keep the numbers of parameters as small as possible.

3.4 Implementation

The layered approach of aggregate programming is suitable for engineering orthogonal systems. We composed our library as a set of cooperating mod-

[2]Modular, component-based.

ules, each of which implements functionality independent of the others. Modules are organised into universal and more specific layers, that respectively provide an increasing level of abstraction.

The result is a prototype `protelis-lang` library, counting more than 150 distinct functions, organised as shown in Figure 3.2. Each of the four “building block” operators is the basis for module exploiting its pattern: `spreading`, `accumulation`, `sparsechoice`, and `time`. Associated with these are `nonselfstabilizing` modules that collect related useful patterns that must be handled with care due to their lack of resilience. At a yet higher level of abstraction, the `meta` module collects general purpose patterns for combining and modulating other functions, and finally `lang` contains a set of simple utility and glue functions used by other modules.

The remainder of this section describes each module (except the simplistic `lang`) in more detail.

3.4.1 Spreading

The `protelis:coord:spreading` module is based around the information spreading operator `G`, illustrated in Figure 2.6(a). This operator produces resilient diffusion of information away from a source region, spreading this information outward along a spanning tree built applying the triangle inequality constraint, and possibly modifying that information as it spreads. The time this takes is proportional to the diameter of the region in which `G` is executing.

Listing 3.1 shows example excerpts from `spreading`, showing how `G` is exploited to build functions based on information moving towards the edges of a spatial region. One such example is `broadcast`, which spreads a copy of information held by the source region. Others include `distanceTo`, a computation of distance of every location from a source region and `distanceBetween`, which provides every location with an estimate of the shortest distance between two regions.

The module also includes alternatives to `G` that perform better for some applications, as the self-stabilisation rate of `G` is inversely bounded by the distance between the closest pair of neighbours and their communication speed [35]. These two alternatives are more specialised but functionally equivalent, and thus may be safely substituted for `G` [20]. One alternative, `crfGradient` [35], is a distance measure that self-repairs very rapidly but is sensitive to repeated small perturbations, while the other, `flexGradient` [36] is a distance measure that tolerates small distortions in return for smoother change over time.

3.4.2 Accumulation

The `protelis:coord:accumulation` module is based around the information accumulation operator `C`, illustrated in Figure 2.6(b). This operator is the complement of `G`, addressing resilient aggregation of information across space: `C` is fed with what to aggregate, then performs that aggregation along a spanning tree down a potential gradient towards a source device that thus eventually reduces all the information scattered through a region into a single summary value. The time this takes is proportional to the diameter of the region in which `C` is executing.

Even small perturbations, however, can cause loss or duplication of data, resulting in transient disruptions that impact stabilisation. Accordingly, `protelis:coord:accumulation` provides alternatives to `C` that use multiple paths down the potential field rather than just one: as `cMultiIdempotent` and `cMultiDivisible` and their specialisations `cMultiMin`, `cMultiSum`, etc.

On top of `C`, other functions are built that use it (and often combinations with `G`) to implement collective state estimation, such as `summarize`, which shares the result of an accumulation through a region, `countDevices`, which counts the number of devices in a region, and `average`, which estimates the average of a local value across a region (Listing 3.2).

```
module protelis:coord:spreading
/* Signature of G */
def G(src, init, metric, accumulate) { ... }
/* Dynamically computes routes between regions */
def channel(source, dest, thr) {
  distanceTo(source)+distanceTo(dest)<=distanceBetween(source, dest)+thr
}
/* Check if a device is close to a source */
def closerThan(source, range) {
  distanceToWithMetric(source, nbrRange) < range
}
/* Self-healing gradient that reconfigures in O(diameter) */
def crfGradient(source, maxHop) { ... }
}
/* Forecast obstacles along a path to a source */
def directProjectionWithMetric(source, obstacle, metric) {
  G(source, obstacle, metric, (v) -> { obstacle || Gnull(v, false) })
}
/* Compute a distance to a source */
def distanceTo(source) {
  distanceToWithMetric(source, nbrRange)
}
/* Extend distanceTo to support unconventional metrics */
def distanceToWithMetric(source, metric) {
  G(source, 0, metric, (v) -> { v + metric.apply() })
}
/* Compute the distance between two regions with unconventional metric */
def distanceBetweenWithMetric(regionA, regionB, metric) {
  broadcast(regionA, distanceToWithMetric(regionB, metric))
}
/* Dilate a spatial region */
def dilate(region, width) {
  dilateWithMetric(region, nbrRange, width)
}
/* Dilate a spatial region with unconventional metric */
def dilateWithMetric(region, metric, width) {
  distanceToWithMetric(region, metric) < width
}
/* Self-healing gradient with optimized change propagation */
def flexGradient(source, epsilon, rate, range, distortion) { ... }
/* Diffuse information along with distance from source */
def gradient(src, init, metric, acc) {
  G(src, [0, init], metric, v -> {
    [addRangeWithMetric(v.get(0), metric), acc.apply(v.get(1))]
  })
}
/* Spread and aggregate information from multiple sources */
def multiGradient(sources, init, metric, accumulate) {
  multiInstance(sources,
    id->{gradient(id == getUID(), init, metric, accumulate)},
    [Infinity, init]
  )
}
/* Computing a Voronoi partition */
def voronoiPatitioningWithMetric(seed, id, metric) {
  G(seed, id, metric, identity)
}
```

Listing 3.1: Example functions from spreading.

```
module protelis:coord:accumulate
/* Signature of C */
def C(potential, accumulate, local, null) { ... }
/* Aggregate information from neighbours */
def aggregation(local, reduce) {
  hood((a, b) -> { reduce.apply(a, b) }, local, nbr(local))
}
/* Spread average information value */
def average(sink, value) { ... }
/* Apply the idempotent f to reduce info desceding a potential */
def cMultiIdempotent(potential, f, local, default) { ... }/**
/* Find the minimum value desceding along a potential */
def cMultiMin(potential, local) {
  cMultiIdempotent(potential, min, local, Infinity)
}
/* Find the maximum value desceding along a potential */
def cMultiMax(potential, local) {
  -cMultiMin(potential, -local)
}
/* Sum values descending along a potential */
def cMultiSum(potential, local) { ... }
/* Devices agree on a common value */
def consensus(init, f) {
  rep (val <- init) {  val + f.apply(sumHood(nbr(val) - val)) }
}
/* Estimate network diameter */
def diameterWithMetric(source, metric) {
  let d = distanceToWithMetric(source, metric);
  2 * rep (maxd <- 0) {max(if(d<Infinity){d}else{0},maxHood(nbr(maxd)))}
}
/* Find neighbor with the lowest potential */
def getParent(potential, f) { ... }
/* Share Laplacian consensus */
def laplacianConsensus(init, epsilon) {
  consensus(init, (v) -> {epsilon * v})
}
/* Aggregate information in source and spread it back */
def summarize(src, reduce, local, null) { ... }
/* Estimate the number of devices within a spatial region */
def countDevices(pot) {
  countDevicesWithCondition(pot, true)
}
/* Estimate the number of devices in which condition holds */
def countDevicesWithCondition(pot, condition) {
  C(pot, sum, if (condition) { 1 } else { 0 }, 0)
}
/* Estimate and broadcast the average value within a spatial region */
def average(sink, value) { ... }
/* Execute code according to the number of devices */
def quorumSensing(pot, zero, thr, under, over) {
  quorumSensingWithCondition(pot, zero, thr, true, under, over)
}
/* Execute code according to the num. of devices in which condition holds */
def quorumSensingWithCondition(pot, zero, thr, condition, under, over) {
  let c = broadcast(pot == zero, countDevicesWithCondition(pot, condition));
  if (c < thr) { under.apply(c) } else { over.apply(c) }
}
```

Listing 3.2: Example functions from `accumulation`.

3.4.3 Symmetry breaking

The `protelis:coord:sparsechoice` module currently contains only the symmetry breaking operator `S`, implemented in Listing 2.5 and illustrated in Figure 2.6(d). This algorithm breaks symmetry through mutual inhibition, in which devices compete against others to become leaders, generating a random Voronoi partition with a characteristic grain size `grain` in expected time $O(grain)$. Example applications include designating a leader device to act as a single aggregation point, breaking a space into management regions, or clustering a network.

3.4.4 State

The `protelis:state:time` module is based around the temporary state operator `T`, illustrated in Figure 2.6(c). This operator essentially implements a flexible timer, which progresses from some initial state to a "zero" state at a potentially time-varying state.

Listing 3.3 shows example excerpts from `state`, exploiting `T` for management of time and memories. For example, `countDown` tracks a timer counting seconds down to zero, `cyclicFunction` executes a function once every `length` seconds, and `limitedMemory` holds a value for a specified number of seconds. Other self-stabilising functions in the module provide related functionality not based on `T`, such as `isRisingEdge`, which checks for a rising edge in a binary signal and `exponentialBackoffFilter`, which filters a signal to smooth out noise.

3.4.5 Meta patterns

In organising the prior code, we noticed a number places in which variants of several functions we created followed a shared pattern. Following the "strategy" design pattern [37], we have factored out these common coordination mechanisms (taking advantage of the higher-order function capabilities of Protelis) and collected them in the `protelis:coord:meta` module. This

```
module protelis:state:time
/* Signature of T */
def T(initial, zero, decay) { ... }
/* Count down a period with a flexible definition of time */
def countDownWithDecay(length, dt) {
  T(length, 0, (t) -> {t - dt})
}
/* Count down a certain number of seconds */
def countDown(length) {
  countDownWithDecay(length, self.getDeltaTime())
}
/* Return true once every length time */
def cyclicTimerWithDecay(length, decay) {
  rep (left <- length) {
    if (left == 0) { length }
    else { countDownWithDecay(length, decay) }
  } == length
}
/* Periodically invoke a function */
def cyclicFunction(length, f, null) {
  cyclicFunctionWithDecay(length, self.getDeltaTime(), f, null)
}
def cyclicFunctionWithDecay(length, decay, f, null) {
  if (cyclicTimerWithDecay(length, decay)) { f.apply() }
  else { null }
}
/* Decrease relevance of information over time */
def evaporation(length, info, decay) {
  T([length, info], [0, info], t->{[t.get(0)-decay, t.get(1)]})
}
/* Filter noise signal */
def exponentialBackoffFilter(signal, a) {
  rep (old <- signal) { signal * a + old * (1 - a) }
}
/* Whether a signal has been stable for a certain time */
def isSignalStableWithDt(signal, time, dt) {
  rep (old <- [signal, 0]) {
    mux (signal == old.get(0)) { [signal, old.get(1) + dt] }
    else { [signal, 0] }
  }.get(1) >= time
}
/* Whether an event was true within timeout */
def isRecentEvent(event, timeout) {
  if (event) { true }
  else { countDown(timeout) > 0}
}

/* Hold a value until a specified timeout */
def limitedMemory(value, null, timeout, dt) { ... }
/* Wait a certain time */
def waitWithDecay(timeout, dt) {
  countDownWithDecay(timeout, dt) <= 0
}
/* Apply a function after a given time */
def waitAndApply(timeout, f, null) {
  if (wait(timeout)) { f.apply() } else { null }
}
```

Listing 3.3: Example functions from `time`.

```
module protelis:coord:meta
/* Distributed publish subscribe pattern */
def publishSubscribe(pub, sub, info, null) {
  if (C(distanceTo(pub), or, sub, false)) {
    broadcast(pub, info)
  } else { null }
}
/* Run an instance of f for every source */
def multiInstance(srcs, f, null) {
  alignedMap(
    nbr(sources.map(self, id -> {[id, null]})),
    (key, field) -> { true },
    (key, field) -> { f.apply(key) },
    null
  )
}
/* Space-time binary branching */
def boundSpreading(region, f, null) {
  if (region) { f.apply() }
  else { null }
}
/* Restricts the execution of f where filter(d) holds */
def multiRegion(d, filter, f, null) {
  if (filter.apply(discriminant)) {
    let res = alignedMap(
      nbr([[discriminant, default]]),
      (key, field) -> { filter.apply(key) },
      (key, field) -> { function.apply() }, default);
    if (res == []) { default } else { res.get(0).get(1) }
  } else { default }
}
/* Gossip active sources */
def findSources(source) {
  let k = 5; /* number of replicated replicas */
  findSourcesWithId(source, getUID(),
    roundTripTime(diameter(S(Infinity, nbrRange)))/(k-1), k);
}
def findSourcesWithId(source, id, diameter, numberOfReplicas) {
  timeReplicatedWithK(() -> {
    rep(v <- []) {
      v.union(unionHood(nbr(v))).union(if(source){[id]}else{[]})
    }
  }, [id], diameter, numberOfReplicas)
}
/* Feed f with pre-processed data */
def preProcessAndApply(input, pre, f) {
  f.apply(pre.apply(input))
}

/* Feed f with pre-processed data, and then post-process the output */
def processAndApply(input, pre, f, post) {
  post.apply(f.apply(pre.apply(input)))
}
/* Apply f, and then post-process the output */
def postProcessAndApply(input, f, post) {
  post.apply(f.apply(input))
}
```

Listing 3.4: Example functions from `meta`.

module thus defines a family of general purpose patterns that enhance code reusability by allowing a number of families of useful variant functions to be generated by higher-order composition rather than by implementing a combinatorial collection of variants.

Listing 3.4 shows example excerpts from `meta`. One such is a simple publish-subscribe pattern for asynchronous and decoupled event notification (inspired by prior work such as [38]) in which autonomous publishers diffuse event notifications to the consumers without an explicit registration process, and information is routed and delivered to the consumers following the shortest path from the producers. Others focus on restriction in space and time, allowing different sub-regions to carry on different computations and controlling the scope of a distributed computation. For instance, `multiInstance` runs multiple copies of a process in parallel, one for each identified "source," and aggregates their outputs. By sharing information, processes convey an extended perception of the system to each device. Wrapping `G` into `multiInstance` enables sharing of information between sources and running concurrent instances of `gradient` (namely, the `multiGradient` function), overstepping the implicit partitioning of `G`[3]. `boundSpreading` allows a function to run only where certain conditions hold, and `multiRegion` generalises branching to use a generic discriminator with potentially many values rather than a binary one (Figure 3.4).

3.4.6 Non-self-stabilising functions

Being universal, field calculus can express any coordination mechanism constructed from local interactions, and it is only a small fraction of such coordination mechanisms that are self-stabilising. Self-stabilisation is extremely valuable for ensuring resilience, but there are cases in which non-self-stabilising coordination or state mechanisms still have a role to play. Since these are inherently dangerous for constructing a resilient system, however, we have segregated them into their own `nonselfstabilizing` modules to

[3] *Gradient* bio-inspired pattern [19].

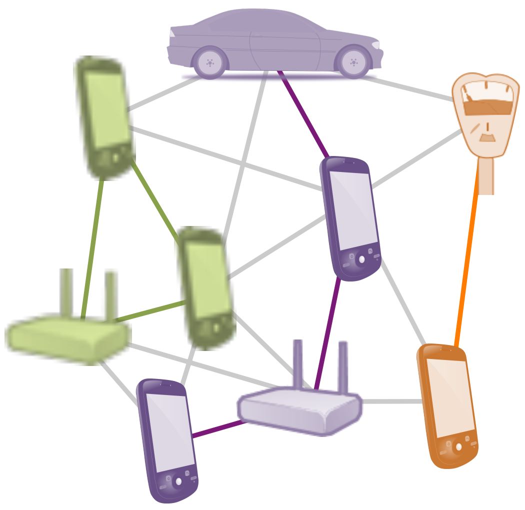


Figure 3.4: `multiRegion` splits a network into multiple subspaces (various colours), applying domain restriction with respect to a generic discriminant rather than the binary one requested by `if`. Grey links are ignored because they cross between the different subspaces.

```
module protelis:coord:nonselfstabilizing:accumulation
/* Share global agreement on scattered data */
def gossip(value, function) {
  rep(v <- value) {
    function.apply(v, hood((a,b) -> {function.apply(a,b)}, value, nbr(v)))
  }
}
/* Gossip whether the device has ever experienced a given condition */
def gossipEver(a) {
  rep (ever <- false) {
    a || anyHood(nbr(ever))
  }
}
```

Listing 3.5: Example functions from `nonselfstabilizing:accumulation`.

ensure that they are not used without knowledge that one is doing so.

One example in this package is `gossip` (Listing 3.5), which like the self-stabilising `summarize`, allows devices to eventually share an aggregated value. Since the `gossip` function spreads and accumulates information in all directions, rather than just up a gradient, it operates much more quickly and much more robustly than `summarize`. That same universal information spreading, however, means that the values produced by `gossip` always change monotonically and cannot recover from a perturbation or track a value that moves both up and down, meaning that it is not self-stabilising. The `gossip` function thus has important uses when it is used in a context where its monotonicity is not problematic, but may fail if composed in other contexts.

Finally, we note that one active area of investigation is the transformation of non-self-stabilising algorithms into self-stabilising algorithms that retain many of their properties. In the specific case of gossip, for example, it has recently been demonstrated that gossip can be transformed into a self-stabilising variant by running several instances overlapping in time [28], and this functionality can be implemented with `protelis-lang` by application of the `timeReplicated` pattern from `meta`.

3.5 Demos

The following two demos qualitatively show how `protelis-lang` can flexibly address resilient distributed applications. The former demonstration, `groupNoisyDevices`, shows how functionalities from multiple modules can be composed together to perform risk assessments in noisy scenarios, while `tsp` leverages the `multiInstance` meta-pattern to address the travelling salesman problem in aggregate systems (e.g., defining a path across some point of interests (POIs) while visiting a city).

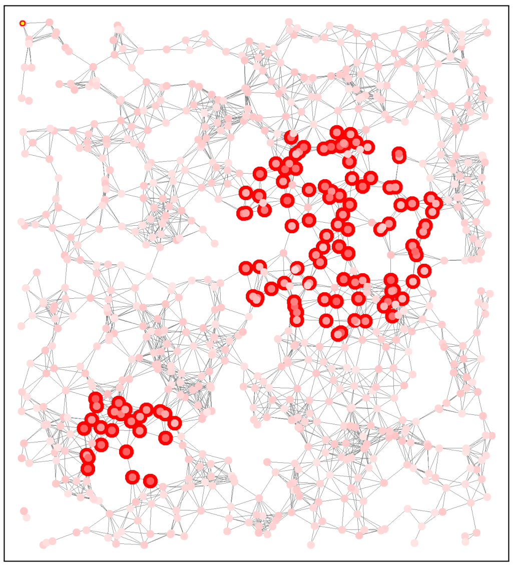
Drawing from all the modules described before, `groupNoisyDevices` (Listing 3.6) clusters misbehaving groups of neighbouring devices, electing leaders as their representatives (to which, for instance, resources can be allocated). `boundSpreading` constraints the scope of the inner anonymous function in noisy regions smoothed by `dilate`. The `dilate` function widens the boundaries of these regions, so that close areas are perceived as a single component during leader election. Hence, instead of having fragmented noisy groups, we cluster them together, potentially reducing the number of elected leaders (this aspect should be tuned according to the application domain). `S` elects representative devices responsible for performing risk assessments by aggregating the number of misbehaving devices through the `countDevices` function. Gathering the number of devices provides a raw estimation of the entities exposed to potential risks, e.g., resources can be distributed to the leaders depending on the number of devices they cluster.

The travelling salesman problem is a well known problem in combinatorial optimisation [39]. The proposed solution, based on the "nearest-neighbour" heuristic, outputs an indicative path that may diverge from the optimal one, but which is still acceptable due to the wide range of unpredictable sources of variability in real world networks[4] (e.g., movement of devices). `tsp` is a self-stabilising function that defines a dynamic path across a sequence of unanticipated points. Indeed, this solution gradually spreads information in the system, avoiding the need of a centralised knowledge at the beginning of the computation. `tsp` (Listing 3.7) leverages the `multiGradient` pattern to estimate a possible path in a fully-distributed environment. The POI closest to the starting point `A` selects the next closest POI to reach, the second POI selects the third closest not-visited point and so on. `multiGradient` runs multiple gradient instances, one for each POI, which are aggregated by each device. Functions based on the `multiInstance` pattern emphasise information reduction (basically, computing tuples out of tuples). Given a tuple of POIs, `findClosest` returns the closest one, while `getId` and `getVisitedPOIs` access to the information wrapped in a tuple.

[4]If the changes in the network topology are too fast, even finding the closest point of interest (POI) ends up being an hard task.

```
/* import accumulation, meta, sparsechoice, spreading, utils */
def groupNoisyDevices(noise, dangerThr, dilateThr, grain) {
  boundSpreading(
    dilate(noise > dangerThr, dilateThr),
    () -> {
      let leader = S(grain, nbrRange);
      [ leader, countDevices(distanceTo(leader)) ]
    },
    [false, -1]
  )
}
```

Listing 3.6: `groupNoisyDevices`


(a) Noisy devices above the security threshold are represented with a wider red circle.

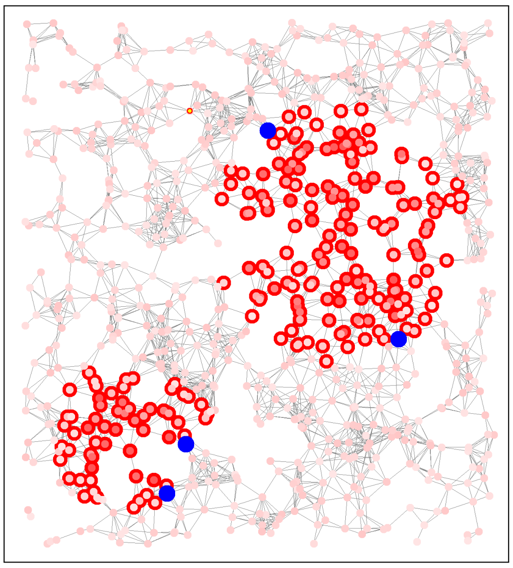

(b) `groupNoisyDevices` elects leaders (blue) to cluster misbehaving regions.

Figure 3.5: Noisy regions are smoothed with `dilate`. `boundSpreading` constrains the execution scope of `S` inside these areas, and `countDevices` provides an estimation of misbehaving devices.

```
/* import meta, spreading, utils */
def tsp(poi, A) {
  rep (v <- null) {
    let pois = multiGradient(findSources(poi), v, nbrRange, identity);
    mux (A) {
      let nearestPOI = /* choose closest POI */
        findClosest(pois.filter(self, a->{getId(a)!=getUID()}), null());
      [getId(nearestPOI), [getUID()]]
    } else {
      let p = getPrev(pois.filter(self, a->{poi&&getNext(a)==getUID()}));
      if (getId(p) != nullId()) {
        let nearestPOI = findClosest(pois.filter(self,
            (a) -> { poi /* consider only POIs ... */
              && !getVisitedPOIs(p).contains(getId(a)) /* !visited yet */
              && getId(a) != getUID() /* != myself */
            }), null);
        if (nullId() == getId(nearestPOI)) { null }
        else {[getId(nearestPOI), getVisitedPOIs(p).union([getUID()])]}
      } else { null }
    }
  }
}
```

Listing 3.7: `tsp`. Null values are replaced with a generic `null`.

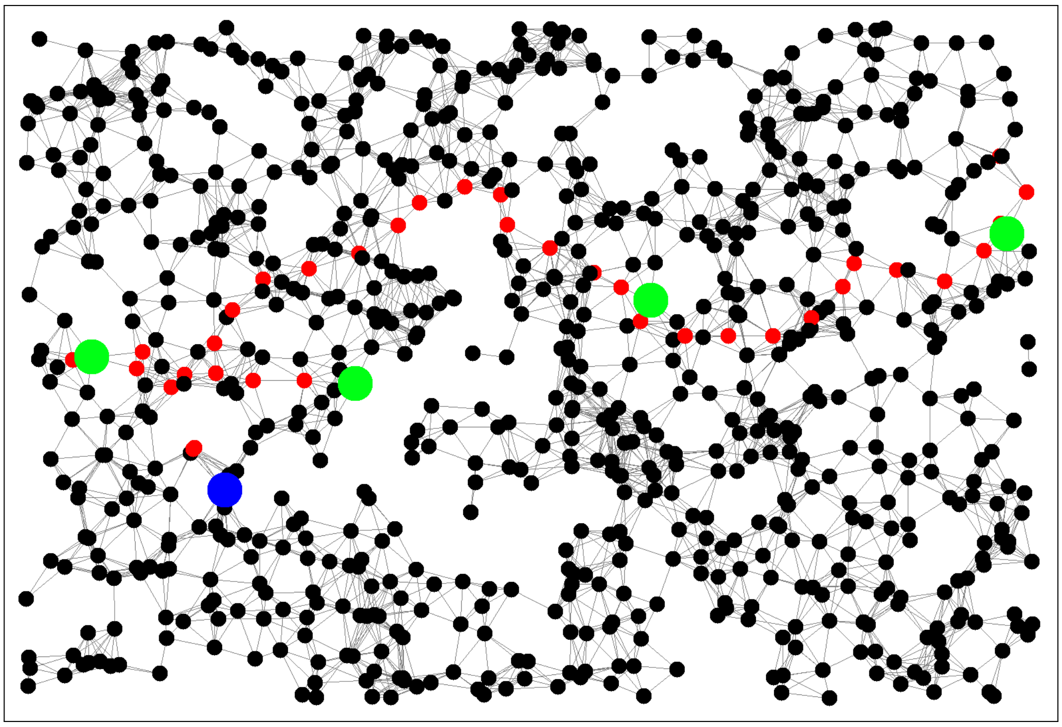


Figure 3.6: Estimate the shortest path (red), starting from `A` (blue), to cover POIs (green) in a fully distributed environment.

Chapter 4

Engineering an algorithm

A software engineering approach to algorithm development usually consists of analysis, design, implementation and empirical evaluation [40]. Engineering an aggregate program follows an iterative three-stage process (Figure 4.1) that progressively treats complex specification, resilience, and efficiency [20]:

1. *Analysis of requirements and self-organising specification.* Analysing the requirements of the distributed application, and understanding how the system should behave under which assumptions;

2. *Minimal resilient design with known coordination patterns.* Decomposing requirements into coordination patterns (e.g., information spreading and aggregation, state tracking) following an analysis of the problem, and mapping these patterns to well-known building blocks. This mapping designs and provides a minimal implementation of the distributed system;

3. *(Ad-hoc) Optimisation.* Defining which building blocks should be considered for replacement with a mechanism from the substitution library expected to provide better performance, confirming the improvement by analysis or simulation, then iterating, until no further improvement can be made.

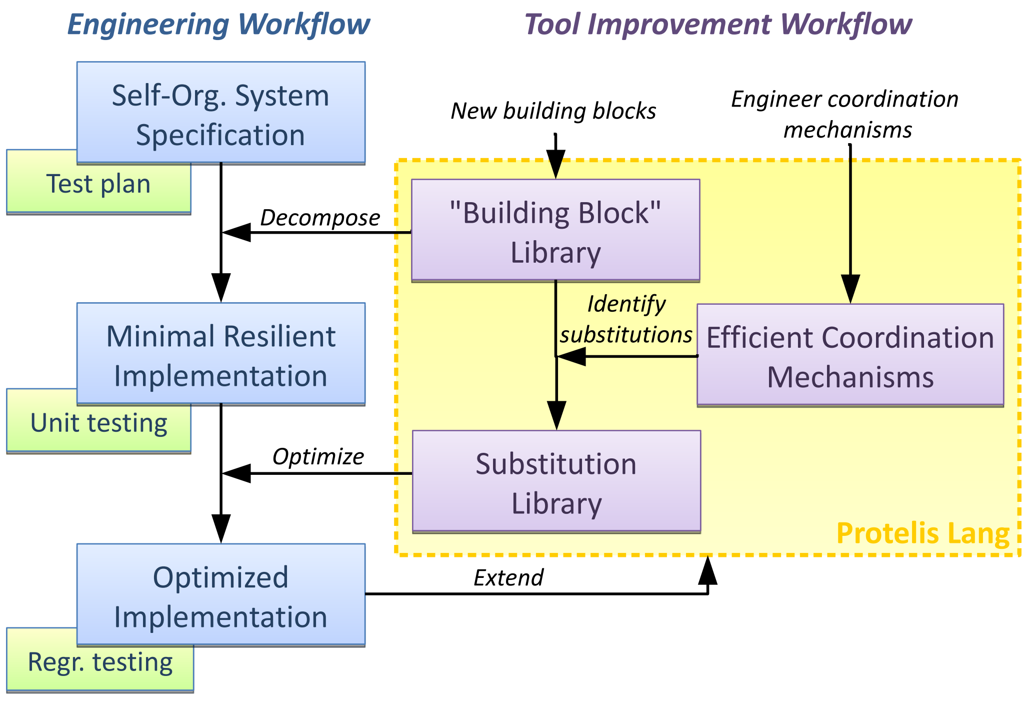


Figure 4.1: Workflow for engineering self-organising systems: an aggregate specification is decomposed into self-stabilising building blocks, then optimised by substitution of equivalent high-performance coordination mechanisms from `protelis-lang`. Adapted from [20].

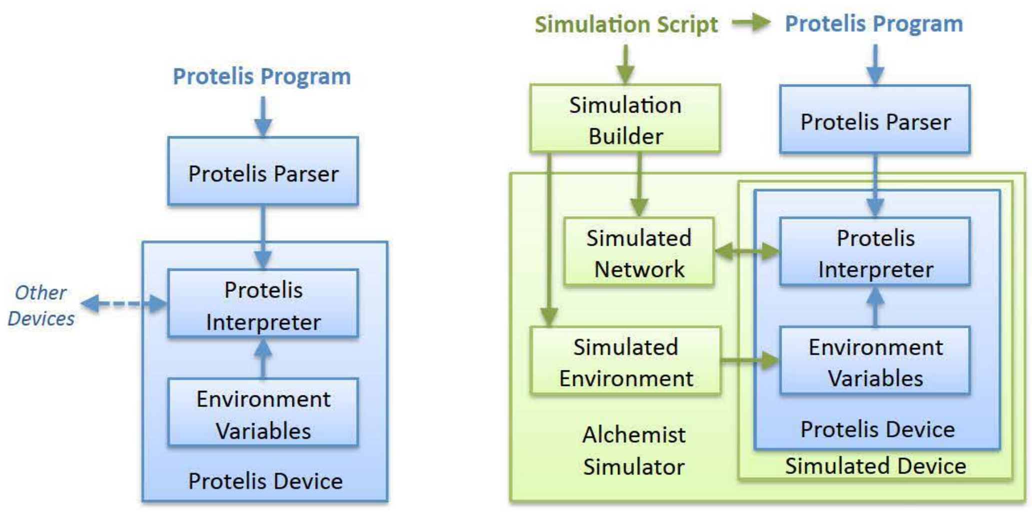


Figure 4.2: Abstract Protelis architecture: an interpreter executes a pre-parsed Protelis program at regular intervals, communicating with other devices and drawing contextual information from a store of environment variables. A minimal Alchemist incarnation handles communication between devices, and other aspects of the environment to be simulated. Adapted from [12].

4.1 Testing framework

Testing frameworks, as development tools, enhance the engineering process [41]. Testing against contracts[1] ensures that a given function honours its contract and tells whether the code meets the contract, and whether the contract means what we think it means. In stage 1, a test plan (namely, a contract) can be extracted by analysing and formalising the application requirements, and defining how the algorithm should behave over a wide range of test cases and "boundary" conditions. Unit testing, in stage 2, guarantees that the minimal implementation is not exposed to any bug and satisfies the test plan.

Unit tests load an artificial, and extremely minimal, simulation environ-

[1]Design by contract: documenting the responsibilities of software modules. A correct program does no more and no less than it claims to do [33].

ment deployed within the Alchemist simulator (Figure 4.2) [42], then check the output returned by the function being tested against known values or previous runs of the same test (regression testing in stage 3).

Having modular code to be tested reduces the impact of changes to any other component in the system. As aggregate programming addresses collective behaviours, defining what a unit is may be a "problem." Despite pointwise expressions (e.g., arithmetic operations, tuples), an aggregate program manipulates computational fields over space and time. As such, even a single specification may be compiled to complex device-to-device interaction. From now on, a function will be considered as a unit even if it may perform inner calls to other functions.

Unit tests are extended with regression tests in stage 3 to verify that the component still behaves correctly even after code optimisation or substitution. Unit and regression testing increase confidence in code maintenance, catching any defects or misbehaviours introduced due to any change.

Testing self-stabilisation requires, at least, to assert two properties: *(i)* Stabilisation test (Figure 4.3(a)), in which, given an initial state N, the network is assured to converge to N_0; *(ii)* Self-stabilisation test (Figure 4.3(b)), in which, after a perturbation of a stable state N_0, the network reacts to the change and eventually converges to N_0 again. Note that self-stabilisation implicitly asserts stabilisation.

Complete black- and white-box tests are both, often, impossible as the combinations of possible inputs may be too many to be tested extensively in a limited amount of time, thus only a very-small subset of all possible inputs is considered. Selecting the subset with the highest probability of finding errors is crucial: well-selected test cases considerably reduce the number of tests to be carried out to exclude bugs and misbehaviours with a reasonable confidence degree. The "equivalence partitioning" approach [43] partitions the input domain into a finite number of equivalence classes, so that testing a representative value of each class is equivalent to a test of any other value in the same class. If a test does not detect any error, then no other test cases in the equivalence class fail unless the partitioning is wrong.

Test cases that explore boundary conditions[2] have a higher pay-off than test cases that do not. Rather than selecting any element in an equivalence class as being representative, boundary-value analysis requires that one or more elements are selected to test the edges of the equivalence class they are part of.

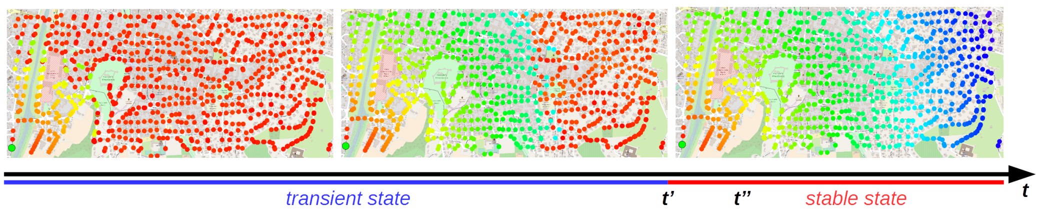
Algorithms can be tested against both equivalence classes and boundary conditions. For instance, if we partition the input space with respect to spatial displacements of devices: *(i) Class A*. A perturbed grid in which devices are almost uniformly distributed allows the testing of an algorithm against fair spatial conditions; *(ii) Boundary condition of class A*. A test case within a symmetric grid of devices verifies whether the system behaves wrong in case of structural symmetry; *(iii) Class B*. A random distribution of devices (still connected enough to avoid network segmentation) tests the algorithm in scenarios with heterogeneous densities of devices.

Equivalence partitioning and boundary analysis are delegated to the expertise of the tester, as it is often impossible to explore the space of all the possible combinations. Test case design can be further improved in terms of error guessing. Given a particular program, intuition and experience lead the designer to guess possible errors and then write tests to expose them. Bugs are then detected after the deployment of the algorithm, and as such should be addressed with new regression tests.

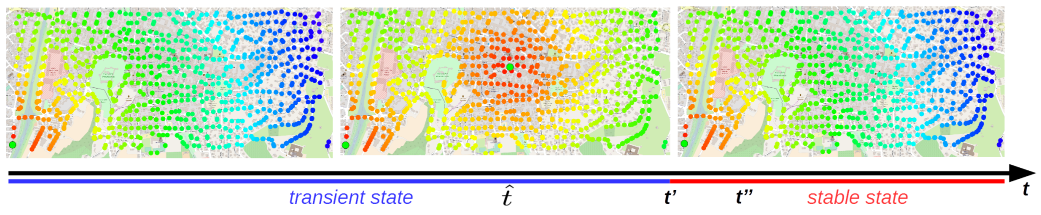
4.2 Quantitative analysis

A quantitative analysis of performance should follow the "minimal resilient implementation phase," producing an estimation of algorithm performance and optimising such implementation iterating over the suitable substitutions. The performance assessment amounts to the evaluation of stabilisation time with respect to some parameter such as: *(i)* size of the network and

[2]Boundary conditions are situations near the edges of equivalence classes.



(a) Stabilisation. `distanceTo` eventually converges to the stable state N_0. A distance-based potential is spread from the source (bottom-left corner) until stabilises at time t'.



(b) Self-stabilisation. After `distanceTo` reached N_0, the system is perturbed adding and then removing a new source at time $\hat{t}$. Being a self-stabilising function, `distanceTo` recovers from the perturbation, converging to N_0 again.

Figure 4.3: Testing stabilisation and self-stabilisation of the `distanceTo` function. Colour saturation increases with the distance to the closest source (green wide circle).

diameter—the larger the diameter, the longer is required to propagate messages across the network; *(ii)* dynamics of input—transients may vary with respect to different states of input, e.g., the rising value problem; *(iii)* depth of nested functions—nested functions affect outer functions' performance. Considering the function `G`, whose complexity is proportional to the network diameter $O(diameter)$, then having nested `G` functions raises the complexity of the program to $O(p * diameter)$ where p is the nesting order.

Performance evaluation requires ad-hoc considerations. For instance, as discussed in [2], `G` and `C` have stabilisation time that is linear in the network diameter under the assumption of stable input, "regular" metric and partially synchronous network, while `T` is strictly dependent on the decay function used as argument. In general, specific empirical analysis must be conducted.

4.3 Protelis-Lang improvement workflow

protelis-lang is a modular and extensible composition of building blocks and functions addressing both state-tracking and coordination, which can in turn be considered either to implement the application or as part of the substitution library. New building blocks, coordination mechanisms and functions can extend this library, widening its scope of application and the assets of the substitution library. Indeed, protelis-lang and the engineering process of an aggregate program intersect: once a new algorithm has been deployed and properly tested, it can be imported into the library if it introduces either new functionalities or an efficient and alternative solution to known problems (e.g., crfGradient, flexGradient and G).

These two engineering processes require standard tools for versioning and continuous integration. protelis-lang is organised following the git-flow model[3], automatically built with Gradle[4] [44] and deployed on Maven Central[5] [45] through TravisCI[6] [46].

4.4 Engineering distanceTo as a case study

This section shows how aggregate algorithms can be implemented, refined and tested, taking distanceTo—which we introduced in Chapter 2—as a case study. Considering this requirement: "creating an algorithm in which each device estimates its distance to a source region," the distanceTo algorithm is engineered here.

1. *Analysis of requirements and self-organising specification.* We can draw parallels between the requirements and bio-inspired patterns [19]. distanceTo and the gradient pattern are nicely related, as the latter diffuses information about the distance to its origin.

[3]https://github.com/nvie/gitflow
[4]https://gradle.org
[5]https://mvnrepository.com/
[6]https://travis-ci.org/Protelis

Self-organising specification: aggregate algorithms should self-stabilise in a time proportional to the network diameter. However, even small changes in the network topology can cause the disruption of a distance-based field, whose self-healing rate should take the rising value problem into account.

2. *Minimal resilient design with know coordination patterns.* The analogies between `G` and the gradient bio-inspired are shown in Chapter 2. Thus, specialising `G` may provide an acceptable minimal implementation to start working on with (Listing 2.2).

3. *(Ad-hoc) Optimisation.* In a long-lived system, both the sources and devices distribution may change over time. Self-healing gradients are subject to the rising value problem [35]—local variation in message speed constrains self-healing rate by the shortest neighbour-to-neighbour distance in the network. `crfGradient` adjusts very quickly to changes in the network, thus is good in scenarios where the transient should be as short as possible. However, fast potential changes can make many devices' values rise before converging again to their correct values. Indeed, even the smallest changes in the source or network can produce small estimate changes throughout the network, leading to high communication and energy costs. In many applications, such as routing and geometric restriction of processes, devices far from the source need only coarse estimates [36]. `flexGradient` algorithm has a tunable trade-off between precision and communication cost. Frequent small changes in the network or source thus cause frequent estimate changes only within a distance proportional to the magnitude of the change.

When fed with the same inputs, `distanceTo`, `crfGradient` and `flexGradient` self-stabilise to the same outputs. Being substitutable functions, one can replace the others depending on the application scenario [20].

4.4.1 Qualitative analysis and unit testing

Figure 4.6(a) qualitatively shows how `distanceTo` diffuses a gradient from two sources, until the system is perturbed by removing a source (Figure 4.6(b)), and the algorithm begins its self-healing phase. As `G` suffers of the rising value problem, the self-healing rate of `distanceTo` is constrained by the shortest neighbour-to-neighbour distance in the network (Figure 4.7(a)). `crfGradient`, drawn from the substitution library (namely, a subset of `protelis-lang`), has a faster self-healing rate, as this gradient can rise at an arbitrary speed (Figure 4.7(b)) [35].

The Protelis repository[7] includes the `Protelis-Test` (Figure 4.4) testing framework based on JUnit[8]. Test cases require both an Alchemist compliant YAML configuration—from which the expected result of the simulation is parsed—and the Protelis code to be tested. Each test requires a number of steps t' within which the algorithm is guaranteed to stabilise, and then an additional range t'' within which stabilisation is checked.

As both `distanceTo` and `crfGradient` are substitutable functions, their test cases are indeed similar. Listing 4.2 describes the expected result for each device UID, the structure of the discrete network (e.g., grid, rectangle, circle and composition of them), the Protelis program to be executed (Listing 4.3) and the communication range between devices. To assert their self-stabilisation, both algorithms are tested against source perturbation: after their stabilisation to N_0, one of the two sources is removed at time $\hat{t}$ (namely `50`), the algorithms start new transients which gradually self-stabilise to N_0 again. To check the stability of the results, test cases assert that the outputs have been stable to N_0 for at least t''.

Note that performance is not addressed by the standard testing procedure, as an accurate study requires ad-hoc experiments with respect to complex network dynamics.

[7]`https://github.com/Protelis/Protelis`
[8]`http://junit.org/junit4/`

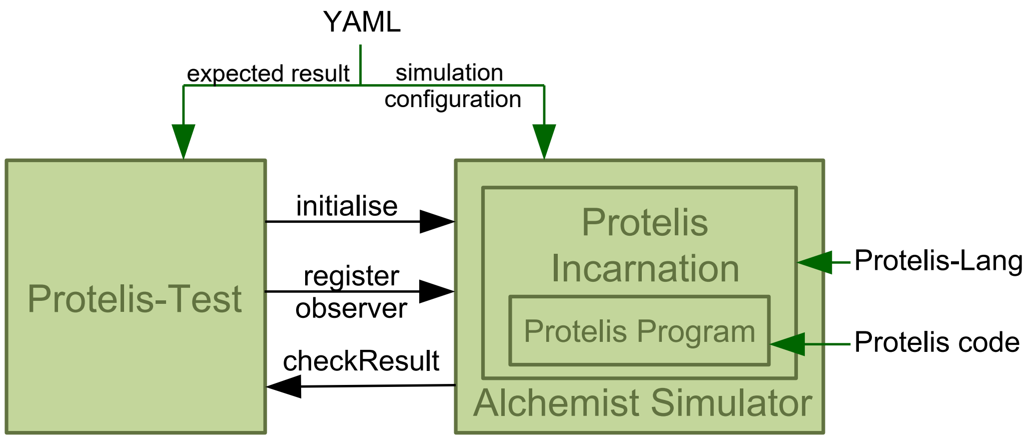


Figure 4.4: The `Protelis-Test` Java-based framework leverages the Alchemist simulator to test functions from `protelis-lang`. `Protelis-Test` initialises and registers an observer to the Alchemist engine. After each simulation step, the Alchemist Engine invokes the `checkResult` method on such observer for validating the simulation outcome. Black arrows represent control flows, while green arrows are generic resource imports.

```java
void checkResult(final long currStep, final int stabilizationSteps,
       final int stabilitySteps, final Environment<Object> env,
       final List<Pair<String, String>> expectedResult, /* ... */) {
    if (currStep >= stabilizationSteps - stabilitySteps) {
      /* Get result from simulation environment (in which Protelis
       * code is executed); compare it with 'expectedResult' (parsed
       * from YAML file); if they differ, then test fails */
    }
}
```

Listing 4.1: `checkResult` method validates the simulation result against the expected result. Recalling Figure 4.3(b): `currStep` is the current simulation step (t), `stabilizationSteps` is the number of steps after which the function is guaranteed to be stabilised (t'), and `stabilitySteps` is the number of steps within which stabilisation is asserted (t''). `expectedResult` is parsed from the YAML configuration.

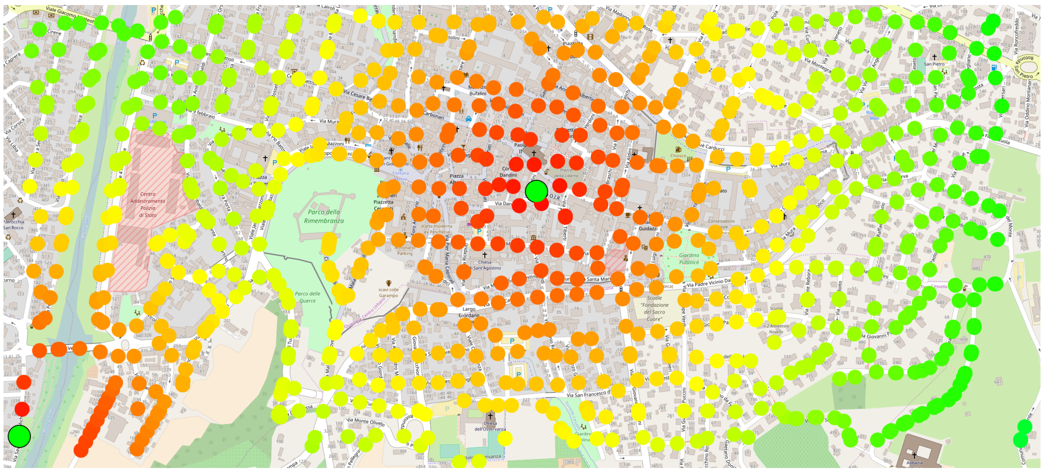
```
# Test results are defined here (namely state N0).
# result:
#  {0 0.00, 1 1.00,  2 2.00,  3 3.00,
#   4 1.00, 5 1.41,  6 2.41,  7 3.41,
#   8 2.00, 9 2.41, 10 2.82, 11 3.82}
incarnation: test
network-model:
  type: EuclideanDistance
  parameters: [1.5]
program: &program
  - time-distribution: 1
    program: distanceTo #crfGradient
displacements:
  - in:   #devices are displaced in a uniform grid
      type: Grid
      parameters: [0, 0, 3, 2, 1, 1, 0, 0]
  programs:
    - *program
```

Listing 4.2: YAML configuration of `distanceTo` and `crfGradient` tests. This configuration also contains the expected outcome of the simulation (`result`), which is parsed and tested by `Protelis-Test`.

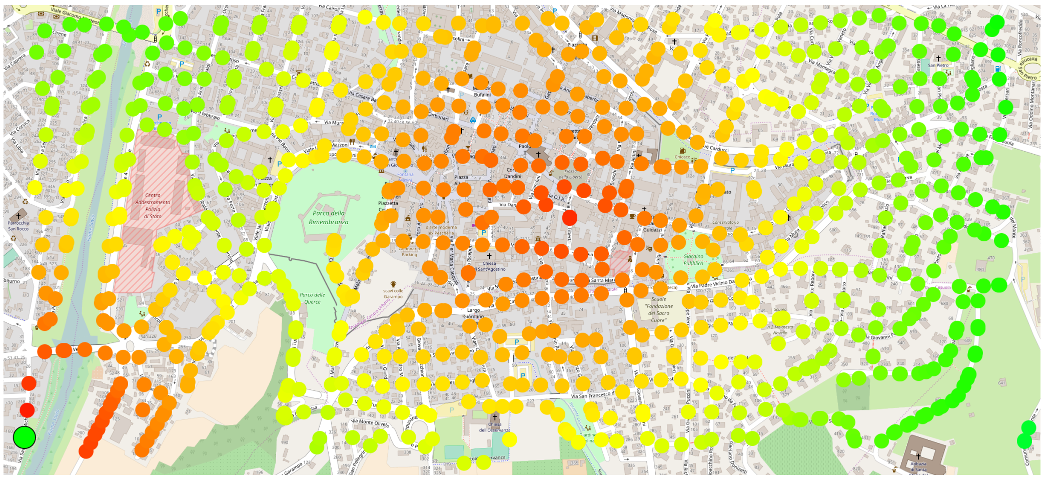
```
/* import ... */
let source = getUID() == "0"
    || getUID() == "5" /* two sources */
         && timeSinceStart() <= 50; /* one source is removed after 50 time
    units*/
distanceTo(source) /* crfGradient(source, 1.5) */
```

Listing 4.3: Protelis code to test `distanceTo` and `crfGradient`. `getUID` returns device UID as string, and `timeSinceStart` counts the time elapsed since the beginning of the code execution. To prove their self-stabilisation, both algorithms are tested against source perturbation, asserting that `distanceTo` and `crfGradient` self-stabilise to N_0.

Figure 4.5: Unit tests of `distanceTo` and `crfGradient`. As these functions are substitutable, their tests are similar. They only differ in the function being invoked.

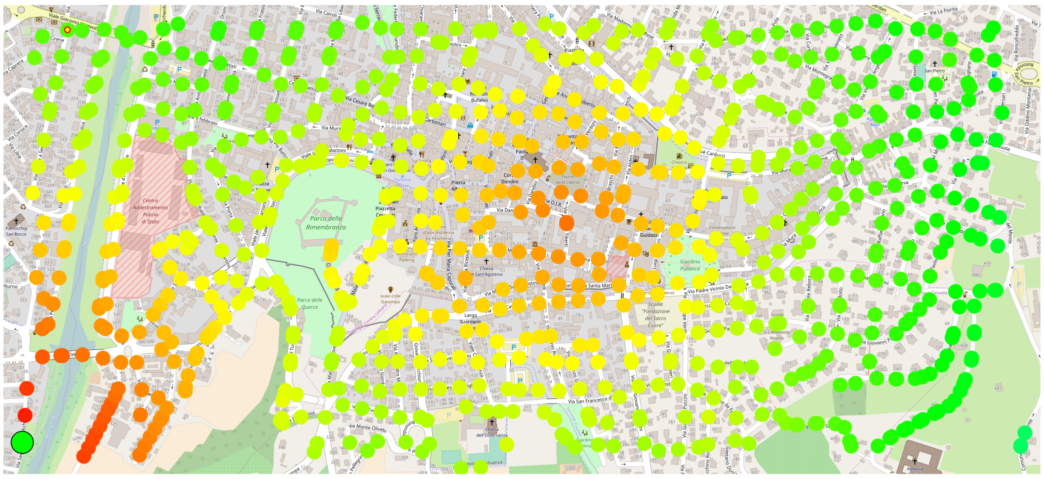


(a) Distance-based potential: a gradient is diffused from two sources (big green circles). Both `distanceTo` and `crfGradient` stabilise to N_0 when fed with a stable input. Colour saturation increases with the distance to the closest source.

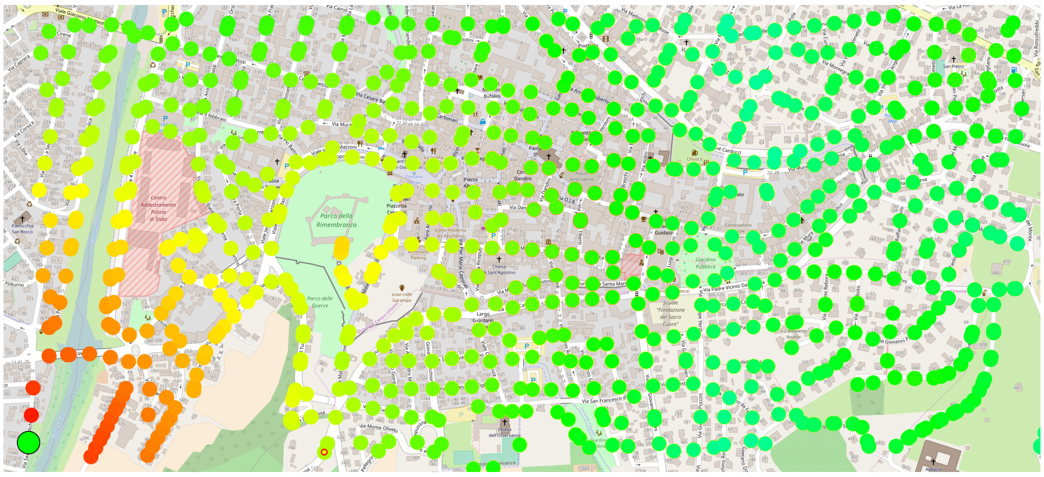


(b) At time $\hat{t}$, the system is perturbed by removing the central source (reaching state N). `distanceTo` and `crfGradient` react to this perturbation and gradually self-stabilise to N_0.

Figure 4.6: Qualitative analysis of self-stabilisation. Being substitutable functions, both `distanceTo` and `crfGradient` stabilise to N_0 following different transients.



(a) `distanceTo` potential.



(b) `crfGradient` potential.

Figure 4.7: Comparison of `distanceTo` and `crfGradient` transients at time t, where $\hat{t} < t < t'$, while converging from state N to state N_0. `distanceTo` healing ratio is slower than `crfGradient`, though both provably reconfigure in $O(diameter)$ time.

Chapter 5

Evaluation

We now demonstrate that `protelis-lang` can indeed provide benefits in the quality, flexibility, and modularity of coding resilient distributed systems. In particular, we aim to show how functions from `protelis-lang` can be easily combined in order to reduce the abstraction gap without compromising the overall performance. The workflow for this approach begins early in the design phase of an aggregate application, with a programmer identifying well-known coordination problems. Identifying these problems should then lead to selection of design patterns that can be readily factored across the algorithms and patterns included in the library.

To illustrate this workflow, we present two application scenarios in the context of public mass events, a particularly challenging pervasive scenario in that they involve mass numbers of personal mobile devices that often overwhelm available fixed wireless infrastructure, strongly limiting the degree to which cloud assets can be used as part of a distributed systems solution. For both scenarios, we execute our experiments using the Alchemist simulator [42]. Devices are scattered with an approximate uniform random distribution along the streets of the Italian city of Cesena, representing some combination of mobile and infrastructural devices participating in the system. In order to ensure a good connectivity in such of sparse environment with a varying number of devices (n), we tune the communication range such that the expected number of neighbours for every device is 10; namely, devices communicate within a unit disc of radius $r = \sqrt{(10 * m)/(\pi * n)}$, where m is the entire area considered.

For each scenario, we show: *(i)* a simple Protelis implementation based on `protelis-lang`, that only partially addresses the scenario issues; *(ii)* an extended solution, making use of the advanced `multiInstance` pattern, that provides a better solution; *(iii)* a qualitative evaluation of the results; *(iv)* a quantitative evaluation, showing the performance impact of relying on library code.

Note that the goal is not to demonstrate optimality, but rather to demonstrate how simply resilient behaviour with reasonable performance can be achieved with our library. Since `protelis-lang` extensively leverages high-order functions, however, the presented solutions could be easily improved, extended, and tuned by changing the specific implementation of the desired component, with only minor changes to the Protelis code.

5.1 Scenario 1: meeting a celebrity

In our scenario, a celebrity, e.g. a movie star, is a featured attendee of an event and moves along a scheduled path through the streets of the city. Many people want to see the celebrity, but crowd movement is constrained, can become dangerous if too many people mob to the same area at the same time, and cannot be easily centrally managed if the crowd is large enough to overwhelm fixed communication infrastructure. In our scenario, then, people wishing to meet the celebrity should rendezvous at a number of different meeting points along the path, while avoiding the creation of perilously dense areas in the process. Allocation of people must respect security constraints of meeting places, such as their maximum capacity. We approach this in two separated stages: first defining meeting points and then managing the crowd with respect to the capacity of each meeting point.

Geographic allocation of meeting points, implemented in Listing 5.1 as

```
/* import meta, sparsechoice, spreading, time */
def clustering(path, grain, d) {
  boundSpreading(path,
    () -> {
      let l = S(grain, nbrRange);
      l && isSignalStable(l, roundTripTime(d))
    }, false)
}
```

Listing 5.1: `S` elects a set of meeting points following the `path` travelled by a celebrity. `isSignalStable` ensures that a meeting point has been stable before being considered.

```
/* import accumulation, spreading, utils */
def meetTheVIP(A, mp, arrT, v, wantsToGo, goAnyway, thr){
  let t = distanceBetween(mp, A)/v;
  let d = distanceTo(mp);
  let c = countDevicesWithCondition(d, wantsToGo&&d/v<t);
  broadcast(t<arrT&&(!(c>thr)||goAnyway), getUID())
}
```

Listing 5.2: *Meet the Celebrity*: person `A` chooses the closest (in time) non-overcrowded meeting point `mp`, considering `v` as her movement speed, `arrT` as the ETA of the celebrity, and `thr` as crowd threshold.

```
/* import accumulation, meta, spreading, utils */
def meetTheVIP(mp,arrT,v,wantsToGo,goAnyway,thr,chs) {
  let t = broadcast(mp, arrT);
  let d = distanceTo(mp);
  let ids = findSources(mp);
  let mps = multiGradient(ids,
    [countDevicesWithCondition(d, d/v<t), arrT],
    nbrRange, identity);
  if(wantsToGo) {chs.apply(mps, goAnyway, thr)}
  else{/*nullUID*/}
}
```

Listing 5.3: *Meet the Celebrity* improved solution. This function is parametric with respect to the crowd management algorithm `chs`.

the function `clustering`, is the first step to avoid the creation of overcrowded regions. We select a number of areas along the path of the celebrity by restricting the scope of `S` using `boundSpreading`. `S` elects a set of meeting points through mutual inhibition, creating a Voronoi partition with a component size `grain`, such that no device is more distant than `grain` from a leader, and no leaders are closer than `grain/2`. Assuming a mean capacity of meeting points c and a uniform distribution of attendees, we can compute `grain` $= \sqrt{(c * m)/(n * \pi)}$. Since values of `S` may flicker during the transient, the `isSignalStable` state function is exploited to consider only stable values[1].

`clustering`'s output is fed as the `mp` (meeting point) parameter of `meetTheVIP` (Listing 5.2), which chooses a suitable destination for a person `A`, avoiding all places in which either the celebrity will arrive earlier, or the number of forecasted attendees reached the security threshold. The solution in Listing 5.2 leverages `distanceTo` and `distanceBetween`, though these, being based on `G`, are unable to forward information about more than a single meeting point. As a consequence, people can only get allocated to the closest meeting point, which may prevent some from meeting a celebrity who might otherwise have travelled to a farther meeting point.

Listing 5.3 presents a better solution, based on `multiGradient` (which is in turn based on `G` and `multiInstance`) to overcome the limitations of the previous algorithm, diffusing information from all the meeting points. The risk of overcrowding is reduced, since attendees can then get distributed across all the meeting points. The overcrowding risk of a meeting point is assessed by counting how many people could reach it before the celebrity passes by. Reflecting the modular approach of `protelis-lang`, `meetTheVIP` accepts the specific crowd management algorithm as a parameter (`chs`). The crowd management algorithm chosen for the scope of this dissertation thus considers: *(i)* meeting points reachable before the celebrity passes by; *(ii)* the

[1]A signal is considered stable if does not change for a time greater or equal to the round trip of a message under a weak fairness assumption on devices firing [28]

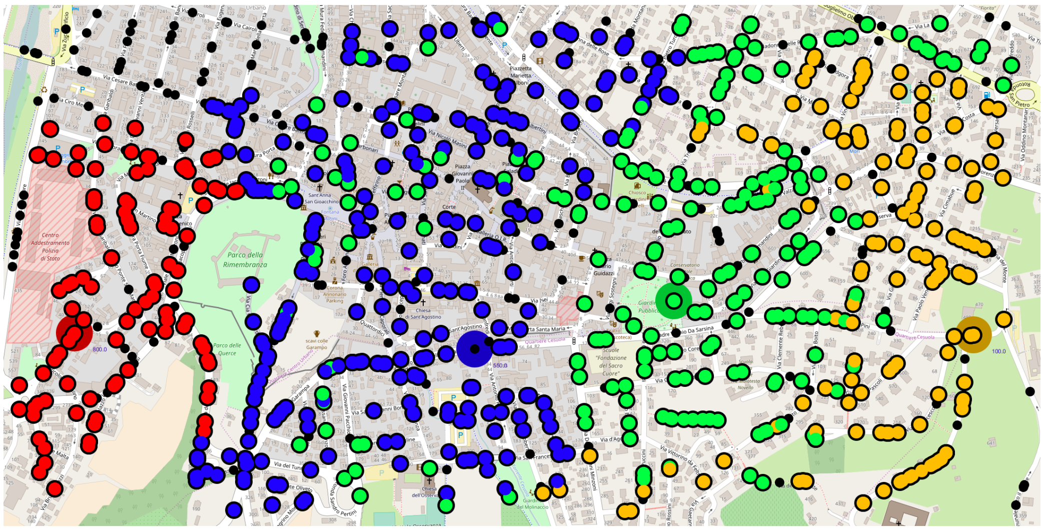
possibility for an attendee to forcibly include overcrowded areas in the range of possible POIs; *(iii)* attendees are steered towards the closest POI only if their estimated time arrival is shorter than one third of the celebrity's; *(iv)* otherwise, attendees are steered towards the less-crowded remaining meeting points. The simulation screenshot in Figure 5.1(a) shows an example of meeting places being selected and how attendees end up distributed across them.

As can be seen, a system for this scenario can be implemented with minimal glue code over the functions provided by `protelis-lang`. The modular decomposition of "Meet the Celebrity" scenario into two sub-problems allows us to separately analyse the allocation of meeting points and crowd management functionalities. The time required by `clustering` to stabilise depends on the set of leaders elected by `S` [18] and thus might not linearly scale with the diameter of the network. On the other hand Figure 5.1(b) shows that, given a set of stable meeting points, crowd management stabilises linearly ($\sqrt{n}$) with respect to the diameter of the network.

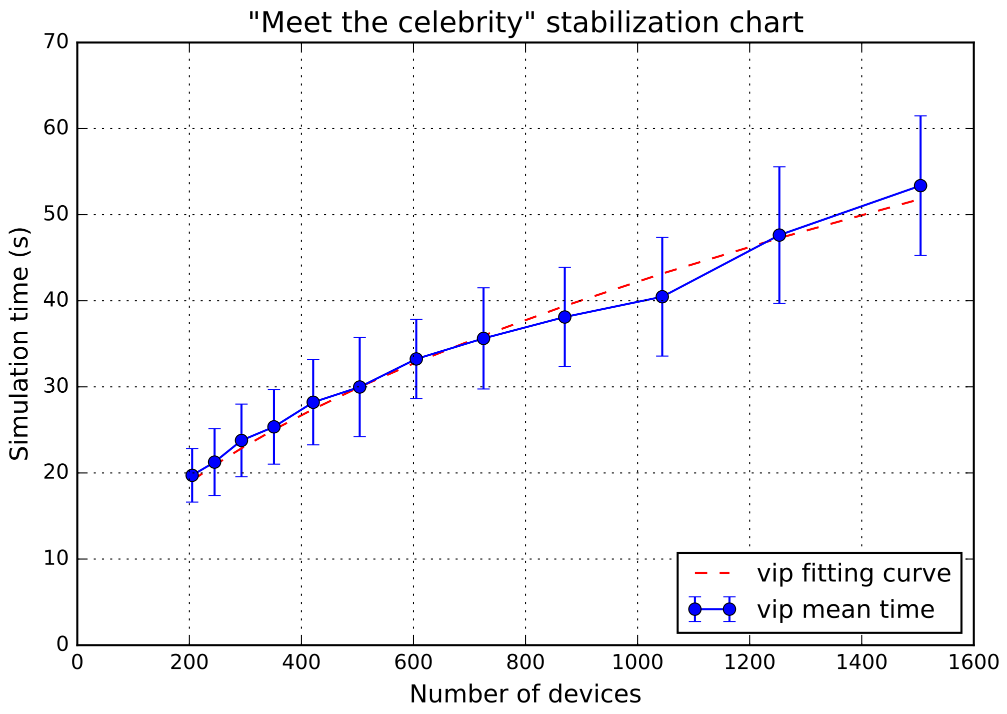
5.2 Scenario 2: resource allocation

Security services (e.g. medical teams) presence and displacement at a mass event should be arranged according to the estimated level of participation and the spatial arrangement of the event. Even if best practices are applied in placing such services, however, emergent situations may arise where the actual optimal (or close to optimal) resource allocation is non-trivial: for instance, security personnel can be allocated according to their mobility capabilities; ambulances can move farther than units on foot, and as such should move towards farther people. Complementarily, units on foot should preferentially aid the closest person. Whichever case, sharing information between health-care personnel and attendees is essential for a proper team coordination to happen.

Listing 5.4 shows a naive solution based on partitioning the network with



(a) Meeting points (wide circles) are dynamically defined across Cesena's map, allowing people to meet a celebrity. Black points are attendees not interested to meet the celebrity or else prevented by overcrowding risk. Other people share the colour of the meeting point they are being steered towards.



(b) Stabilisation time of Listing 5.3. Each point is the average of 70 simulations with random device displacements. Error bars show ±1 standard deviation.

Figure 5.1: Qualitative and quantitative analysis of "Meeting the Celebrity" scenario

```
/* import accumulation, spreading, utils */
def resourceAllocation(dngr, res, needRes, maxD, allocate) {
  rep (ress <- []) {
    let d = distanceTo(dngr);
    let resToDanger = C(d, (a, b) -> { a.union(b) },
      if (res && allocate.apply(
        gradient(dngr, [ress, needRes], nbrRange, identity), maxD)) {
          [[getUID(), d]]
      } else {[]}, []);
    if (dngr) { resToDanger } else {[]}
  }
}
```

Listing 5.4: *Resource allocation.* `dngr`: whether the person is in danger, `res` : whether the person is part of security personnel, `needRes`: estimation of resources needed, `maxD`: range covered by the resource, `allocate`: allocation criteria. An example execution is shown in Figure 5.2(a).

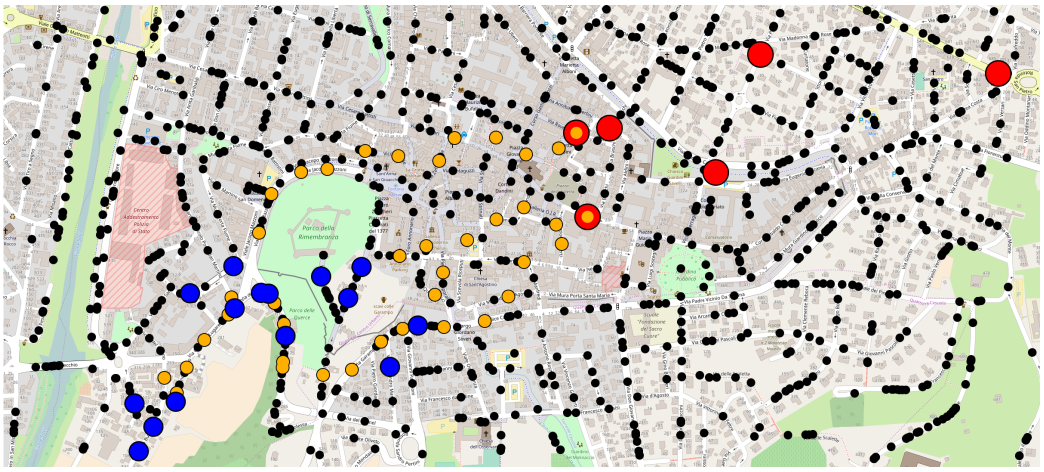
```
/* import accumulation, meta, spreading, utils */
def resourceAllocation(dngr, res, needRes, maxD, canMove, allocate) {
  let ids = findSources(dngr);
  let closest = distanceTo(dngr);
  let closeRes = countDevicesWithCondition(closest,res);
  rep (ress <- []) {
    let bcst = multiGradient(ids,
      [ress, closeRes, needRes], nbrRange, identity);
    let dst = allocate.apply(bcst,closest,maxD,canMove);
    let resToDanger = multiInstance(ids,
      id -> {
        C(distanceTo(id==getUID()), (a,b)->{a.union(b)},
          if (res && getId(dst) == id) {
            [[getUID(), getDistance(dst)]]
          } else {[]}, []);
      }, [/*nullUID*/, []]).reduce(/*id=getUID()*/);
    if (dngr) {resToDanger} else {[]}
  }
}
```

Listing 5.5: *Resource allocation* improved solution. `dngr`: whether the person is in danger, `res`: whether the person is part of security personnel, `needRes` : estimation of resources needed, `maxD`: range covered by the resource, `canMove`: predisposition to move towards people father than the closest one, `allocate`: allocation criteria. Even though `G` and `C` partitions the network in sub-regions of devices, `multiInstance` pattern runs an instance of a function for all the devices satisfying a certain condition, namely being in danger. Security personnel share information about people in need and allocate themselves according to `allocate`. An example execution is shown in Figure 5.2(b).

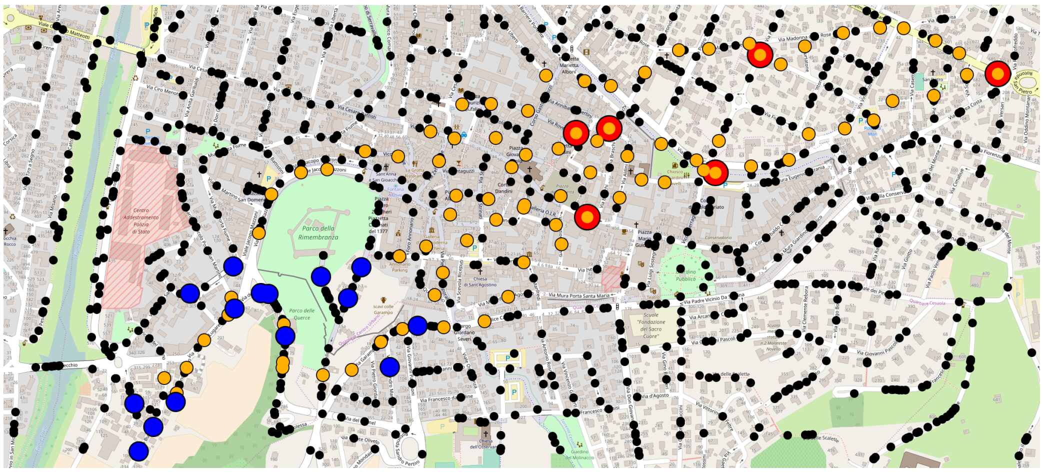
`C` and `G`, which can lead to unfair resource assignment, while an improved solution leveraging `multiInstance` is shown in Listing 5.5. An example showing this difference between the two is shown in Figure 5.2(a) and Figure 5.2(b): in the former, only two out of the six available resources allocated, while all of them are allocated in Listing 5.5. The main difference between the two proposed algorithms is to be found into the use of `multiInstance` (and `multiGradient`, which is based upon it), though the basic structure is substantially shared. As in the previous example, in the naive implementation only the closest person in need of help is considered by the medical team, de-facto preventing more sophisticated allocations.

In second proposed implementation, security personnel receive information from all the attendees in need and can be potentially allocated to any of them, performing a better resource distribution. A key part of both algorithms is the `allocate` function, which encapsulates the actual allocation criteria. The strategy we used in our experiment is the following: *(i)* every security unit may intervene within a `maxD` radius; *(ii)* only those units for which `canMove` yields `true` may assist other people than the closest one; *(iii)* units that `canMove` are randomly allocated to people in danger, attendees with no close units get a probability boost; *(iv)* if a unit is unemployed, it may participate the request of a person in need who is already being serviced. Such rules do not prevent a particularly unfortunate distribution of security personnel to be unable to optimally cover the needs of the attendance. However, since the allocation strategy is abstracted away from the core algorithm, a more sophisticated version with better heuristics could be plugged in without any other change in the program structure.

Figure 5.3 shows that both the proposed solutions scale and stabilise with $\sqrt{n}$ (with n the number of devices). The naive strategy, however, suffers from multiple disruptions and reconstructions of its fragile `C` spanning tree, hindering performance at low network diameter.



(a) Naive resource allocation scenario. As both `G` and `C` cluster the network in sub-regions, only the people in need (red dots) closest to the security personnel (blue dots) get serviced. The shortest paths connecting security personnel and attendees in need is depicted in orange.



(b) By exploiting `multiInstance` to execute overlapping instances of `G` and `C`, it is possible to allocate security personnel (blue dots) in such a way that all requesters (red dots) get served. Orange dots depict shortest paths.

Figure 5.2: Qualitative and quantitative analysis of “Resource Allocation” scenario

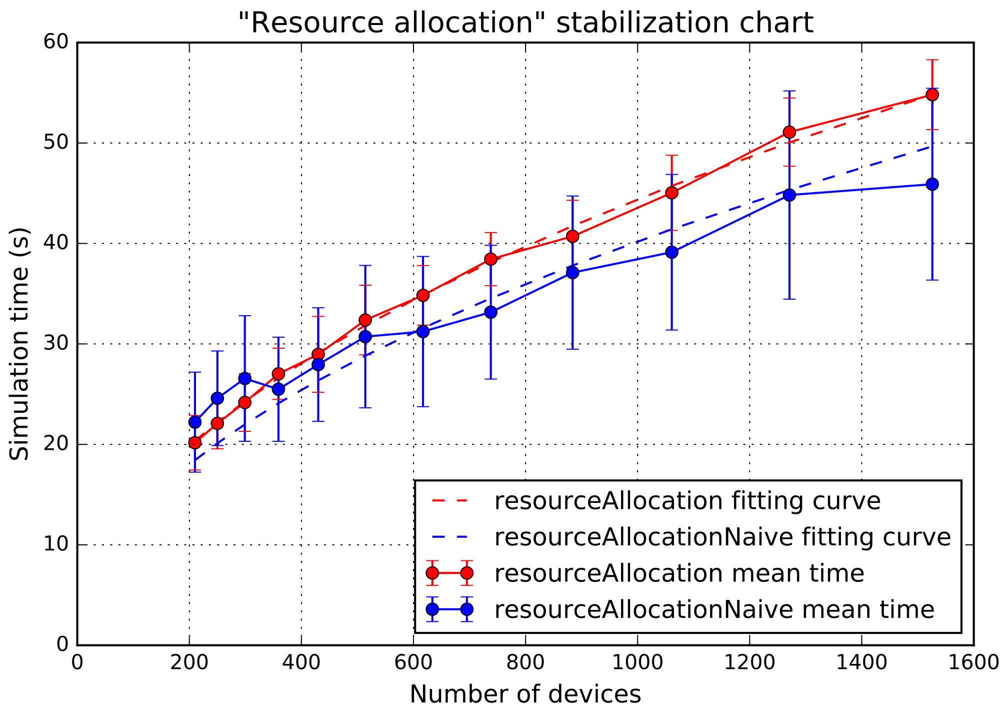


Figure 5.3: Stabilisation time of naive (blue) and improved (red) resource allocation. Each point is the average of 50 simulations with random device displacements. Error bars show ±1 standard deviation.

Chapter 6

Conclusion

In this dissertation, we have presented a Protelis library, `protelis-lang`, that implements a prototype API for resilient system design and extends the workflow for engineering aggregate algorithms with a testing framework. The `protelis-lang` library bridges a critical gap that has heretofore existed in the aggregate programming framework, between the theoretical results providing properties of resilience, scalability, safe composition, etc., and the pragmatics of exploiting and applying these properties in the construction of complex distributed systems.

Field calculus needs higher abstractions to scale with system complexity. Highly general building block algorithms have been built on top of them, allowing fully-resilient coordination of computational entities. The `protelis-lang` library is organised around these operators, providing modules for spreading, aggregation and evaporation of information, symmetry breaking through mutual inhibition, and meta-patterns which extend the application scope of existing algorithms. The efficacy, flexibility and expressiveness of this library are illustrated through scenarios of large-scale crowd application and their empirical evaluation in simulation.

This prototype, of course, leaves room for improvements and completion: just as with any other library, we expect that there are many refinements that can be made in its contents and their organisation, and that these will be most readily discovered as the library is leveraged for its intended purpose of building applications. There are also important classes of functionality that were not in the scope of this effort, such as coordinated movement algorithms, to be addressed either through expansion of this library or construction of complementary libraries. Finally, the `protelis-lang` library can also serve as a foundation for the construction of domain-specific APIs customised for particular application areas, such as emergency service coordination, home automation, or public event management.

Bibliography

[1] Mahadev Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal communications*, 8(4):10–17, 2001.

[2] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *IEEE Computer*, 48(9):22–30, 2015.

[3] 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: tt http://arxiv.org/abs/1202.5509.

[4] Jacob Beal and Mirko Viroli. Aggregate programming: From foundations to applications. In *Advanced Lectures of the 16th International School on Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems - Volume 9700*, pages 233–260, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[5] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. on Software Engineering Methodologies*, 18(4):1–56, 2009.

[6] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.2*, September 2009.

[7] E. Sklar. Netlogo, a multi-agent simulation environment. *Artificial life*, 13(3):303–311, 2007.

[8] 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.

[9] Radhika Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT, 2001.

[10] Daniel Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. PhD thesis, MIT, 1999.

[11] M.E. Inchiosa and M.T. Parker. Overcoming design and development challenges in agent-based modeling using ascape. *Proceedings of the National Academy of Sciences of the United States of America*, 99(Suppl 3):7304, 2002.

[12] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 1846–1853. ACM, 2015.

[13] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.

[14] Jean-Louis Giavitto, Christophe Godin, Olivier Michel, and Przemyslaw Prusinkiewicz. Computational models for integrative and developmental biology. Technical Report 72-2002, Univerite d'Evry, LaMI, 2002.

[15] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[16] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In *Advances in Service-Oriented and Cloud Computing*, pages 114–128. Springer, 2013.

[17] Jacob Beal and Mirko Viroli. Space–time programming. *Phil. Trans. R. Soc. A*, 373(2046):20140220, 2015.

[18] Jacob Beal and Mirko Viroli. Building blocks for aggregate programming of self-organising applications. In *Proceedings of the 2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, SASOW '14, pages 8–13, Washington, DC, USA, 2014. IEEE Computer Society.

[19] 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*, 12(1):43–67, 2013.

[20] Mirko Viroli, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Efficient engineering of complex self-organising systems by self-stabilising fields. In *Self-Adaptive and Self-Organizing Systems (SASO), 2015 IEEE 9th International Conference on*, pages 81–90. IEEE, 2015.

[21] Ferruccio Damiani, Mirko Viroli, Danilo Pianini, and Jacob Beal. Code mobility meets self-organisation: a higher-order calculus of computational fields. In *Formal Techniques for Distributed Objects, Components, and Systems*, pages 113–128. Springer, 2015.

[22] Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. Self-adaptation to device distribution changes. In Giacomo Cabri, Gauthier Picard, and Niranjan Suri, editors, *10th IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2016, Augsburg, Germany, September 12-16, 2016*, pages 60–69, 2016.

[23] Marco Mamei and Franco Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(4):15, 2009.

[24] Mirko Viroli, Danilo Pianini, and Jacob Beal. Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In *International Conference on Coordination Languages and Models*, pages 212–229. Springer, 2012.

[25] Jacob Beal, Mirko Viroli, and Ferruccio Damiani. Towards a unified model of spatial computing. In *7th Spatial Computing Workshop (SCW 2014), AAMAS*, 2014.

[26] Ferruccio Damiani and Mirko Viroli. Type-based self-stabilisation for computational fields. *arXiv preprint arXiv:1509.05659*, 2015.

[27] Mirko Viroli and Jacob Beal. Resiliency with aggregate computing: State of the art and roadmap. *arXiv preprint arXiv:1607.02231*, 2016.

[28] Danilo Pianini, Jacob Beal, and Mirko Viroli. Improving gossip dynamics through overlapping replicates. In Alberto Lluch Lafuente and José Proença, editors, *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9686 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2016.

[29] Ferruccio Damiani, Mirko Viroli, and Jacob Beal. A type-sound calculus of computational fields. *Science of Computer Programming*, 117:17 – 44, 2016.

[30] Mirko Viroli, Giorgio Audrito, Ferruccio Damiani, Danilo Pianini, and Jacob Beal. A higher-order calculus of computational fields. *CoRR*, abs/1610.08116, 2016.

[31] Jonathan Bachrach and Jacob Beal. Building spatial computers. 2007.

[32] Andrew Hunt. *The pragmatic programmer.* Pearson Education India, 2000.

[33] Robert C Martin. *Clean code: a handbook of agile software craftsmanship.* Pearson Education, 2009.

[34] Robert C Martin. Professionalism and test-driven development. *Ieee Software*, 24(3), 2007.

[35] Jacob Beal, Jonathan Bachrach, Dan Vickery, and Mark Tobenkin. Fast self-healing gradients. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 1969–1975. ACM, 2008.

[36] Jacob Beal. Flexible self-healing gradients. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1197–1201, New York, NY, USA, 2009. ACM.

[37] Erich Gamma. *Design patterns: elements of reusable object-oriented software.* Pearson Education India, 1995.

[38] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, August 2001.

[39] Mandell Bellmore and George L Nemhauser. The traveling salesman problem: a survey. *Operations Research*, 16(3):538–558, 1968.

[40] Peter Sanders. Algorithm engineering–an attempt at a definition. In *Efficient Algorithms*, pages 321–340. Springer, 2009.

[41] Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[42] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7(3):202–215, 2013.

[43] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[44] Tim Berglund and Matthew McCullough. *Building and Testing with Gradle.* " O'Reilly Media, Inc.", 2011.

[45] John Casey, Vincent Massol, Brett Porter, and Carlos Sanchez. Better builds with maven. *The How-to Guide for Maven*, 2:p61, 2008.

[46] BN Vasilescu, SB van Schuylenburg, JJHM Wulms, Alexander Serebrenik, and Mark GJ van den Brand. Continuous integration in github: experiences with travis-ci. 2014.