

Towards a Foundational API for Resilient Distributed Systems Design

Matteo Francia University of Bologna Cesena, Italy matteo.francia2@studio.unibo.it	Danilo Pianini University of Bologna Cesena, Italy danilo.pianini@unibo.it	Jacob Beal Raytheon BBN Technologies Cambridge, MA, US jakebeal@ieee.org	Mirko Viroli University of Bologna Cesena, Italy mirko.viroli@unibo.it
---	---	---	---

Abstract—Engineering resilient distributed systems remains extremely challenging, particularly in mapping from collective specifications to individual device behavior. Aggregate programming aims to address this problem using a computational field abstraction to provide inherent guarantees of resilience, scalability, and safe composition. These capabilities are provided, however, by an expressive but terse set of operators too low-level for pragmatic use in complex systems development. We thus present an API to raise the level of abstraction, thereby providing an accessible and user-friendly interface for construction of complex resilient distributed systems. In particular, we capture and organize a large, heterogeneous collection of algorithms and use patterns into a unified framework, including methods for common tasks such as leader election, distance and state estimation, and gossip-based information dissemination. We demonstrate how the expressiveness of this library reduces the abstraction gap required to engineer scenarios of large-scale pervasive computing, while introducing the novel multiInstance pattern enabling an unanticipated composition of computational fields.

Index Terms—Aggregate computing, programming languages, self-organization, application programming interface

I. INTRODUCTION

Heterogeneous computational devices and networks increasingly pervade our environment, as computation has become cheap enough to embed computational devices (FPGAs, microcontrollers, etc) in nearly every aspect of our lives [1], [2]. It remains extremely challenging, however, to program distributed systems that fully take advantage of this dense computational environment, due to its scale, heterogeneity and potential complexity of interactions. The wide gap between collective behavior requirements and their resilient implementation has long been recognized and addressed for a number of specialized applications and with a variety of strategies [3], [4].

Aggregate programming [5] generalizes across many such prior efforts with a layered architecture that factors these challenges into separable sub-problems. The foundation of aggregate programming is field calculus [6], which, by the means of composition of computational fields (an evolving map from a collection of devices to data values), provides transformations from collective specification to local implementation. Atop this foundation are systems of resilient coordination operators [7]

that implicitly provide adaptation capabilities such as self-stabilizing recovery from faults and independence from details of device distribution [8]. Both field calculus and the resilient coordination operators, however, are extremely terse systems of generalized mathematical operators, and thus too low-level for pragmatic use, especially when the complexity of systems increase—much like other foundational models like λ -calculus [9] or π -calculus [10]. Thus, as with other more specialized collective-to-local programming frameworks (e.g., [11], [12], [13]), the pragmatic usability of aggregate programming depends strongly on providing libraries with an accessible and user-friendly application programming interface (API) for commonly used mechanisms and patterns, thereby raising the effective level of abstraction and reducing programming complexity.

To this end, we have extended Protelis [14], a field-calculus implementation hosted in Java, by developing the `protelis-lang`¹ library, aiming to provide a domain-general “foundational API” for aggregate programming. Additionally, and as one of the key contributions of this paper, we created new “meta-patterns” providing advanced reuse and composition techniques, including `multiInstance`, a distributed algorithm using first-class functions to compute and compose a dynamically-determined number of parallel functional processes.

The remainder of this paper is organized as follows: Section II provides additional background and description of related work; Section III provides details on the goals, coverage and structure of the library, including fine-grained description of existing and new algorithms; Section IV illustrates the expressiveness of the library through selected scenarios of large-scale pervasive computing; and Section V summarizes contributions and future work.

II. RELATED WORKS AND BACKGROUND

Many specialized approaches for programming collective behaviors have previously been developed across many different fields, one survey of which may be found in [3]. From a software engineering perspective they have tended to cluster into five main classes: making device

¹Available at <https://github.com/Protelis/Protelis>

```

P ::=  $\overline{F}$  e ;; Program
F ::= def f( $\overline{x}$ ) {e} ;; Function definition
s ::= e | let x = e | x = e ;; Statement
e ::= w |  $\overline{s}$  ;; Basic expression
| f( $\overline{e}$ ) | e.m( $\overline{e}$ ) | e.apply( $\overline{e}$ ) ;; Calls
| if(e){e'} else {e''} ;; Space-time branch
| nbr(e) ;; Neighbor values
| rep(x<-w) {e} ;; Persistent state

```

Fig. 1. Protelis core abstract syntax, adapted from [14].

interaction implicit (e.g., TOTA [15], MPI [16], NetLogo [17], Hood [18]), geometric and topological constructions (e.g., Origami Shape Language [19], Growing Point Language [20], ASCAPE [21]), summarizing state from space-time regions and streaming these summaries to other regions (e.g., TinyDB [22], Regiment [23], KQML [24]), automatically splitting computational behavior for cloud-style execution (e.g., MapReduce [13], BOINC [25], Sun Grid Engine [26]), and providing generalizable constructs for space-time computing (e.g., Protelis [14], Proto [27], MGS [28]). It is from this last, and particularly Proto, that field calculus and the aggregate programming approach derive, aiming at a generalization that can effectively encompass the vast majority of the above approaches.

Complementarily, from a “bottom-up” perspective, there have recently been efforts to organize and systematize algorithms and patterns producing resilient self-organizing behavior, with the aim of developing a “library of resilience.” The first such library was the catalogue of bio-inspired patterns in [4], systematizing mechanisms of information spreading, aggregation, and decay (temporal pertinence) into higher-level patterns. Aggregate programming follows this idea by proposing building blocks expressive enough to cover most mechanisms in [4], but with an additional key feature: resiliency properties such as self-stabilization (adaptation to transient changes) [7] and eventual consistency (adaptation to distribution density, topology, and changes) [8] are mathematically proved, and transfer to any composition of these blocks. As noted above, these building blocks are high general and terse, motivating the desire for a more user-friendly API layer atop them.

A. Aggregate programming with the Protelis language

The computational field calculus [6] provides a universal [29] formal foundation for the aggregate computing, and the Protelis functional programming language [14] has been developed as practical field calculus implementation.

The abstract syntax of Protelis necessary to understand the library presented in this paper is shown in Figure 1: overbar semi-formal notation is used to denote sets or sequences of syntactic elements. (i) `nbr(e)` creates a field where each device maps neighbors (including itself) to their latest available evaluation of e , (ii) `if(e){e'} else {e''}` performs an exclusive branch, partitioning the network into two space-time subregions—where e evaluates to true and false, respectively—computing e' in the former and e'' in

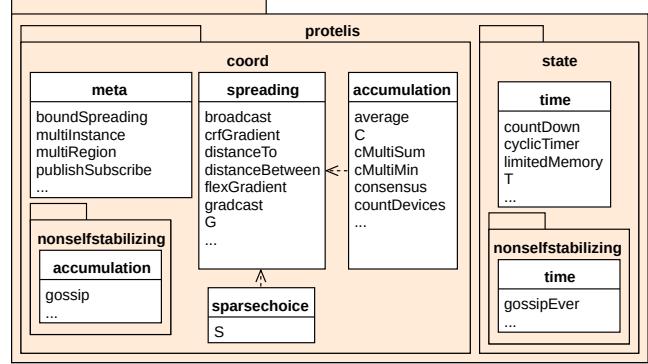


Fig. 2. Library package organization: spreading, accumulation, state and sparsechoice are built around the four resilient “building block” operators described in [7], while meta contains higher-order coordination patterns. The nonselfstabilizing sub-packages contain less resilient functions, which must thus be used more carefully.

the latter, in isolation. (iii) `rep(x<-w) {e}` defines a time-varying field that is initially w , and is updated at each round by the function binding the prior value to variable x and evaluating body e .

III. THE PROTELIS-LANG LIBRARY

While a programmer may have clear ideas about the aggregate behaviors desired from a system, the details of implementing such behaviors in the low-level operations of field calculus (or Protelis) are often quite intricate and sensitive to implementation details. This is particularly true for ensuring that behaviors are resilient and scalable, as these properties are often quite difficult to validate using either formal analysis or empirical testing [30]. In order to fulfill the promise of aggregate programming, we need a comprehensive library that provides higher-level building blocks already guaranteed to be resilient and scalable, thus insulating application programmers from these challenges.

In constructing our Protelis library, `protelis-lang`, we thus 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-stabilizing “building-block” operators [7]: G (spreading), C (aggregation), T (temporary state), and S (sparse choice). Critically, any program constructed using only these operators for coordination and state is guaranteed to be self-stabilizing and to scale well asymptotically in terms of time complexity (though details depend on specific usage). We then searched through all of the publications referenced for associated algorithms and code fragments, importing and adapting any that could be mapped onto these operators or proved equivalent, along with any other patterns and supporting functions of interest.

We then compared the contents thus identified to the two broad surveys referenced in the prior section ([3], [4]) to systematically identify and fill in gaps in coverage. Notably, comparison between building block algorithms and [4] highlighted limitations of G with respect

to the *gradient* pattern described in [4], which are further discussed in Section III-E. We also applied systematic software engineering practices to ensure the generality of the library. For example, as `broadcast` [5] uses a distance metric to generate a gradient for spreading information from a source, we provided both a metric-agnostic version (`broadcastWithMetric`) and a meta-solution (`gradcast`) where the gradient-generating algorithm of choice can be passed as an argument, making it easy to leverage more efficient specialized algorithms in the library such as `flexGradient` (see below).

This analysis thus ensures that the functions in our library are likely to cover most of the communication and coordination primitives needed for implementing any self-organizing system. Encouragingly, we found that the building-block operators correspond nicely with three of the bio-inspired mechanisms from [4]—diffusion, aggregation and evaporation of information—while the others in that work were explicitly ruled out of the scope of this current effort as focusing on device movement (e.g., flocking and swarming) and geometric patterning.

The result is a `protelis-lang` library containing more than 150 distinct functions grouped together accordingly to the “building block” operator they rely on, and organized as shown in Figure 2. By respecting the principles described in [7], the functions composing these modules inherit the self-stabilizing and eventual consistency features of the building blocks they rely on. Associated with these are also two additional 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 that combine and modulate other functions. Among them, `multiInstance` is one of the key contributions of this paper: this meta-pattern uses first-class field functions to compute and compose a dynamically-determined number of computational fields by running parallel instances of a function. We now discuss each module in turn.

A. Spreading

The `protelis:coord:spreading` module is based on the information spreading operator G , illustrated in Figure 3(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. 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.

In this module, G is exploited to build functions based on information moving towards the edges of a spatial region, such as `broadcast`, which spreads a copy of the information held by the source region. Other functions include `distanceTo`, which estimates distance of each device from a source region and `distanceBetween`,

which provides every device with an estimate of the shortest distance between two regions.

The module also includes two alternatives to G that perform better for some applications, as the self-stabilization rate of G is inversely bounded by the distance between the closest pair of neighbors and their communication speed [31]. One alternative, `crfGradient`, is a distance measure that self-repairs rapidly but is sensitive to repeated small perturbations [31]; the other, `flexGradient`, is a distance measure that tolerates small distortions in return for smoother change over time [32]. As these are equivalent to particular uses of G , they may safely substitute for it [7].

B. Accumulation

The `protelis:coord:accumulation` module is based on the information accumulation operator C , illustrated in Figure 3(b). This operator is the complement of G , resiliently aggregating information: C aggregates a field of values along a spanning tree defined by a potential gradient to a source device that receives the reduction of all values in the field into a single summary value.

Even small perturbations, however, can cause loss or duplication of data, so the module also includes specialized alternatives that use multiple paths down the potential field rather than just one: `cMultiIdempotent`, `cMultiDivisible`, their specializations `cMultiMin`, `cMultiSum`, etc.

Applications of C (often combined with G) provide various collective state estimations, 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.

C. Symmetry breaking

The `protelis:coord:sparsechoice` module is currently only the symmetry breaking operator S , illustrated in Figure 3(c). 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 $O(grain)$.

D. State

The `protelis:state` module is based on the temporary state operator T , illustrated in Figure 3(d). This operator essentially implements a flexible timer, which progresses from some initial state to a “zero” state at a potentially time-varying rate. T is applied to manage time and memory. For example, `countDown` implements a timer counting seconds down to zero, `cyclicFunction` executes a function periodically, and `limitedMemory` holds a value for a specified number of seconds. Other 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.

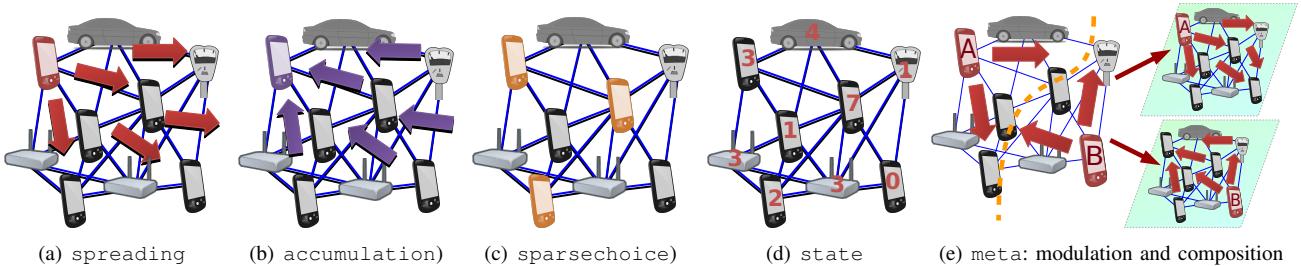


Fig. 3. The main modules of *protelis-lang* are based on: a) spreading (G), diffusing possibly changing information from sources; b) accumulation (C), aggregating information from many locations; c) sparse choice (S), breaking symmetry; d) temporary state (T), adaptive retention of information over time; and e) meta algorithms to modulate and compose other algorithms, such as the dynamic parallel computation of `multiInstance` (a, b, c, d adapted from [30]).

```

1 /* Run an instance of f for every source */
2 def multiInstance(srcs, f, null) {
3   alignedMap(
4     nbr(srcs.map(self, id -> {id, null})),
5     (key, field) -> { true },
6     (key, field) -> { f.apply(key) },
7     null)
8 }
```

Listing 1. `multiInstance` pattern from `meta`.

E. Meta patterns

In organizing the prior modules, we noticed several sets of function variants following shared patterns. Following the “strategy” design pattern [33], we have factored out these common mechanisms (using higher-order Protelis functions) and collected them in the `protelis:coord:meta` module. This module thus defines a family of general purpose patterns, which enhance code reusability by allowing useful functionality to be implemented by higher-order composition rather than a combinatorial collection of variants.

Listing 1 shows the implementation of the novel `multiInstance` meta-pattern, which leverages Protelis’ `alignedMap` construct (field alignment based on arbitrary keys) to run multiple copies of a process in parallel, one for each identified “source,” and aggregate their outputs.

This pattern offers a two-fold benefit: on the one hand, it can be leveraged to improve system stability and resilience: for instance, time replication [34] can be implemented as a specialized `multiInstance`. On the other hand, by computing multiple instances of a function, devices share information coming from multiple sources, augmenting their perception of the aggregate system as a whole. The `multiInstance` meta-pattern thus allows reuse and composition of existing algorithms, extending their application scope. For instance, as we introduced in the spreading module, G creates spatial network partitions in which devices can only interact with the closest source. By running multiple instances of G, `multiInstance` overcomes this limitation, and allows devices to merge information from all the available sources. As such, the gradient pattern from [4] is one of the possible specializations of `multiInstance`.

Yet other patterns focus on modulating scope in space

and time, such as `multiRegion`, which generalizes Boolean branching to instead use a generic discriminator with potentially many values, and `boundSpreading`, which allows a function to run only where certain conditions hold.

F. Non-self-stabilizing 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-stabilizing. Self-stabilization is extremely valuable for ensuring resilience, but there are many cases in which non-self-stabilizing coordination or state mechanisms still have a role to play. Since these are inherently dangerous to constructing a resilient system, however, we have segregated them into their own `nonselfstabilizing` modules to ensure that they are not used without explicit knowledge that one is doing so.

Finally, we note that one active area of investigation is the transformation of non-self-stabilizing algorithms into self-stabilizing algorithms that retain many of their properties. In the specific case of the `gossip` pattern, for example, it has recently been demonstrated that `gossip` can be transformed into a self-stabilizing variant by running several instances overlapping in time [34], and this functionality can be implemented with `protelis-lang` by application of the `timeReplicated` pattern from `meta`.

IV. EXPERIMENTS AND PERFORMANCE

We now show how functions and meta-patterns from `protelis-lang` can be readily combined to implement two complex application scenarios in the context of mass public events, while reducing the abstraction gap of the aggregate program. For each scenario, we compare two Protelis implementations² by counting their lines of code (LoC) as a well understood indicator of the effort required to engineer an aggregate program: an implementation that leverages the novel `multiInstance` meta-pattern and other library functions (L), and “building blocks only” (BB) implementation of approximately the same size, but which is not allowed to use any library code. This size restriction

²Code available: <https://bitbucket.org/mfrancia/2017-ecas-experiments>

provides an approximation of what can be accomplished with in the two conditions (with and without library) with a similar level of programming effort, and results in the BB implementation being necessarily much simpler and less sophisticated in its handling of our test applications.

To compare the performance of our two implementations for each test scenario, we used Alchemist [35] to simulate a network of 1000 devices dispersed through the streets of Cesena, Italy, comprising both mobile (800) and infrastructural (200) devices, all with a 150 meter communication radius. Note, however, that the goal of this evaluation is not to seek optimality or directly compare with prior systems (to the best of our knowledge, no existing system has the desired resilience properties), but rather to demonstrate how our library makes it easier to obtain resilient behaviors.

A. Scenario: “Meet the Celebrity”

In this scenario, a celebrity is a featured attendee of an event and appears at a sequence of meeting locations in the city. Because of potential for danger in overcrowded situations, security personnel allow only a limited number of people at each meeting place. Attendees can coordinate however, with an app on their personal devices that tries to maximize how many people get to see the celebrity while minimizing distance travelled and avoiding over-crowding.

In the building-block-only implementation, the G operator estimates distances and spreads information from meeting places, but each device receives information only from the closest source. As a consequence, people are allocated to the closest meeting place, which can result in some becoming overcrowded while others are underutilized.

By contrast, the library-based alternative leverages the `multiInstance` pattern to count how many people are either arrived already or expected to arrive in each meeting point. This information is then shared to all devices, where the crowd management algorithm considers both the meeting places with available room reachable before the celebrity passes by, and the meeting places for which the sum of arrived and arriving people is smaller than the overcrowding threshold. Attendees are then steered towards the closest timely meeting place that is not already overcrowded (Figure 5). From Figure 4, we can see that with few more application (App) LoC, and a much larger library code, the library implementation overcomes G ’s partitioning and enables capabilities the building-block-only solution does not have.

B. Scenario: Resource Allocation

In this scenario, we design a service assisting in allocating emergency responders (e.g., medical teams) to a set of people experiencing medical emergencies, while maximizing the coverage of all needs. As emergencies can occur anywhere, and a good allocation of resources to needs is not always obvious, we consider the allocation of 15 emergency responders to assist 4 attendees in need,

LoC	Total	BuildingBlocks	Library	App
meetTheVip-BB	21	13	0	8
meetTheVip-L	240	79	142	19
resourceAllocation-BB	69	40	0	29
resourceAllocation-L	245	79	149	17

Fig. 4. Library code (L) enables more complicated total implementations than building-block-only code (BB) with a similar amount of application lines of code (LoC).

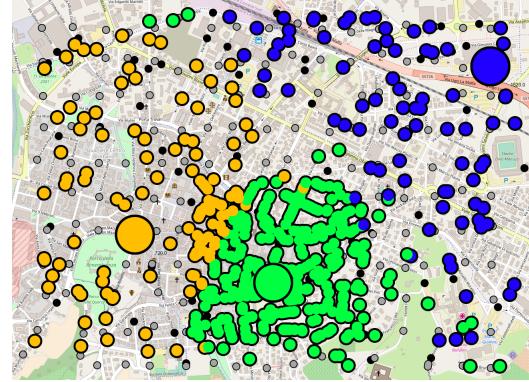


Fig. 5. Screenshot of “meet the celebrity” simulation: meeting opportunities (large circles) are assigned to as many people as safely possible (matching colors). Unassigned people are black dots, structural devices are grey.

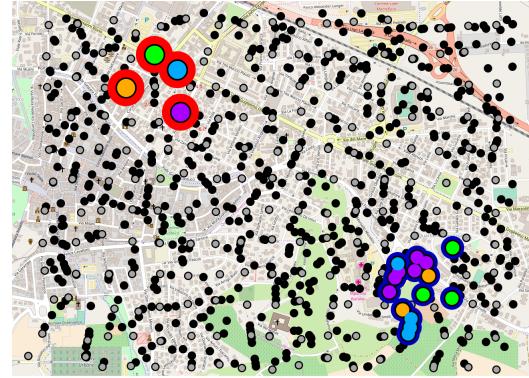


Fig. 6. Screenshot of “resource allocation” simulation: emergency responders (blue) are allocated and guided to people in need (red); allocation is indicated by inner circle color. Other devices are black (mobile) and grey (structural) dots.

requiring respectively 2, 3, 4, and 6 aid units, clustered in the opposite corners of the city centre (Figure 6).

Due to the partitioning limitations of G , the building-block-only implementation leads to a poor resource assignment as responders can only get allocated to the closest need. The library-based solution, on the other hand, leverages `multiInstance` to coordinate responders with the following strategy: (i) responders are allocated to the emergency with the least allocated resources until its needs are satisfied; (ii) if two emergencies have the same amount of resources, responders are allocated to the closest one. As shown in Figure 4, the large library codebase effectively reduces the required application code and thus simplifies the engineering phase of fully resilient systems, also achieving

additional capabilities.

V. CONCLUSION

We have presented a library, *protelis-lang*, implementing an API for resilient system design. This library bridges a critical gap in the aggregate programming framework, between 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. This library also introduces novel meta-patterns, most notably `multiInstance`, providing additional useful new functionality, as demonstrated in our experiments.

This prototype, of course, leaves much room for improvements, refinements and extensions. There are important classes of functionality that were not in the scope of this effort, such as movement coordination and security issues, to be addressed either through expansion of this library or construction of complementary modules. Finally, the *protelis-lang* library serves as a foundation for future construction of domain-specific APIs customized for particular application areas, such as emergency service coordination, home automation, or public event management.

VI. ACKNOWLEDGEMENTS

Partial support provided by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001117C0049. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. This document does not contain technology or technical data controlled under either U.S. International Traffic in Arms Regulation or U.S. Export Administration Regulations.

REFERENCES

- [1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.
- [2] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.
- [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.
- [4] 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.
- [5] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *IEEE Computer*, 48(9):22–30, 2015.
- [6] Ferruccio Damiani, Mirko Viroli, and Jacob Beal. A type-sound calculus of computational fields. *Science of Computer Programming*, 117:17 – 44, 2016.
- [7] Mirko Viroli, Jacob Beal, Ferruccio Damiani, and Danilo Pianini. Efficient engineering of complex self-organising systems by self-stabilising fields. In *IEEE SASO*, pages 81–90, 2015.
- [8] Jacob Beal, Mirko Viroli, Danilo Pianini, and Ferruccio Damiani. Self-adaptation to device distribution changes. In *IEEE Self-Adaptive and Self-Organizing Systems (SASO)*, pages 60–69, 2016.
- [9] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.
- [10] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I. *Information and Computation*, 100(1):1–40, September 1992.
- [11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.
- [12] Cheng Chu, Sang Kyun Kim, Yi-An Lin, YuanYuan Yu, Gary Bradski, Andrew Y Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. *Advances in neural information processing systems*, 19:281, 2007.
- [13] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Comm. of ACM*, 51(1):107–113, 2008.
- [14] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In *ACM Symposium on Applied Computing*, pages 1846–1853, 2015.
- [15] 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.
- [16] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.2*, September 2009.
- [17] E. Sklar. Netlogo, a multi-agent simulation environment. *Artificial life*, 13(3):303–311, 2007.
- [18] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *Int'l Conf. on Mobile systems, applications, and services*, 2004.
- [19] Radhika Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT, 2001.
- [20] Daniel Coore. *Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. PhD thesis, MIT, 1999.
- [21] 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*, 99(Suppl 3):7304, 2002.
- [22] Samuel R. Madden, Robert Szewczyk, Michael J. Franklin, and David Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. In *Workshop on Mobile Computing and Systems Applications*, 2002.
- [23] Ryan Newton and Matt Welsh. Region streams: Functional macro-programming for sensor networks. In *Workshop on Data Management for Sensor Networks (DMSN)*, pages 78–87, August 2004.
- [24] Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. KQML as an agent communication language. In *Conference on Information and Knowledge Management*, pages 456–463, 1994.
- [25] David P Anderson. Boinc: A system for public-resource computing and storage. In *Workshop on Grid Computing*, pages 4–10, 2004.
- [26] Wolfgang Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Cluster Computing and the Grid*, pages 35–36, 2001.
- [27] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.
- [28] 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.
- [29] Jacob Beal, Mirko Viroli, and Ferruccio Damiani. Towards a unified model of spatial computing. In *7th Spatial Computing Workshop (SCW 2014), AAMAS*, 2014.
- [30] Jacob Beal and Mirko Viroli. Space-time programming. *Phil. Trans. R. Soc. A*, 373(2046):20140220, 2015.
- [31] 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.
- [32] 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.
- [33] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.
- [34] Danilo Pianini, Jacob Beal, and Mirko Viroli. Improving gossip dynamics through overlapping replicates. In *Coordination Models and Languages*, pages 192–207, 2016.
- [35] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7(3):202–215, 2013.