

Engineering multi-agent systems with aggregate computing

Danilo Pianini¹, Mirko Viroli¹, and Jacob Beal²

¹ University of Bologna, Italy
{danilo.pianini,mirko.viroli}@unibo.it
² Raytheon BBN Technologies, USA
jakebeal@bbn.com

Abstract. Recent works foster the idea of engineering distributed situated systems by taking an aggregate stance: design and development are better conducted abstracting away from individuals’ details, rather directly programming overall system behaviour. Concerns like interaction protocols, self-organisation, adaptation, and large-scaleness, are automatically hidden under the hood of the platform supporting aggregate programming. One common approach to the engineering of such systems is to provide a framework whose API exposes some aggregate-level abstractions. Another possibility is to tackle the issue at the language level, embedding aggregate abstractions directly into language primitives. This second approach is followed e.g. by MIT Proto, whose very core ideas were recently formalised into the “Field Calculus”, upon which a new practical language, Protelis, has been built. The goal of this demo is to incrementally show the potentiality of such an approach in the design of distributed, situated systems.

1 Introduction to Aggregate computing

In recent years, a number of different strands of research on self-organizing systems have come together to create a new “aggregate programming” approach to the engineering of distributed systems. Aggregate programming is motivated by a desire to avoid the notoriously intractable “local to global” problem, where the system designer must predict how to control individual devices to achieve a collective goal. The whole approach starts from the observation that the complexity of large-scale situated systems must be properly hidden “under-the-hood” of the programming model, so that composability of collective behaviour can be more easily supported and allow to better address the construction of complex systems.

Unifying a number of the proposed aggregate programming approaches is the notion of a “computational field” that maps each device in the field’s domain to a local value in its range. This concept was originally developed for spatial computers, in which communication and geometric position are closely linked, but can support effective aggregate programming of many non-spatial networks as well.

Aggregate programming is then based on the following three principles:

1. the “machine” being programmed is a region of the computational environment whose specific details are abstracted away (perhaps even to a pure spatial continuum);
2. the program is specified as a manipulation of data constructs with spatial and temporal extent across that region;
3. these manipulations are actually carried out in a robust and self-organizing manner by the aggregate of cooperating devices situated in that region, using local interactions.

A mathematical foundation for such approaches has been formalized recently with a minimal “field calculus” [5] that appears to be an effective unifying model, covering a wide range of aggregate programming models, both continuous (e.g., geometry-based) and discrete (e.g., graph-based).

The *field calculus* captures the key ingredients of aggregate neighbour-based computation into a tiny language suitable for grounding programming and reasoning about correctness – recent works addressed type soundness [3] and self-stabilisation [4].

The unifying abstraction is that of computational field, and every computation (atomic or composite) is about functionally creating fields out of fields. Hence, a program is made of an expression to be evaluated in space-time (ideally, in a continuum space-time, practically, in asynchronous rounds in each device of the network) and returning a field evolution.

2 Protelis

On this foundation, a practical language has been developed: Protelis [6]. On the one hand, it incorporates the main spatial computing features of the field calculus, hence enjoying its universality, consistency, and self-stabilization properties [2, 4]. On the other hand, it turns the field calculus into a modern specification language, improving over Proto by providing:

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

Field calculus is a theoretical construct; any practical implementation must embed a field calculus interpreter within an architecture that handles the pragmatics of communication, execution, and interfacing with hardware, operating system, and other software. At the same time, it is important that this system be readily portable across both simulation environments and real networked devices. Finally, both system development and maintainability are greatly enhanced if the exact same code is used for execution in all contexts.

For Protelis, we approach these problems following the same general pattern as was used for the Proto VM [1]. Figure 1(a) shows the abstract architecture for Protelis virtual devices. First, a parser translates a text Protelis program

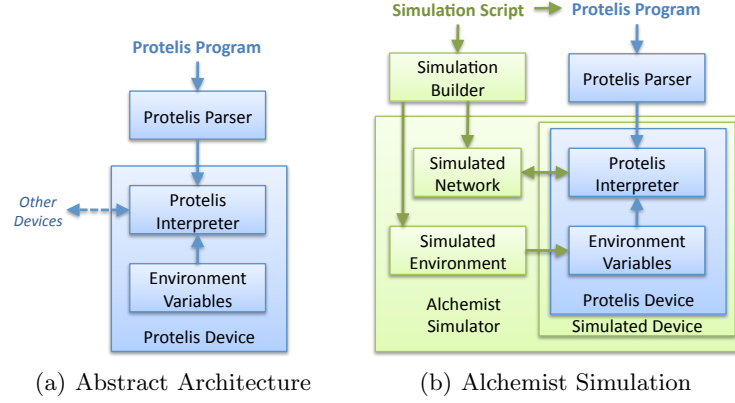


Fig. 1. In the abstract Protelis architecture (a), an interpreter executes a pre-parsed Protelis program at regular intervals, communicating with other devices and drawing contextual information from a store of environment variables. Alchemist provides machinery for hosting instances of the Protelis interpreter into simulated devices (b).

into a valid representation of field calculus semantics. This is then executed by a Protelis interpreter at regular intervals, communicating with other devices and drawing contextual information from environment variables implemented as a tuple store of $(token, value)$ pairs. This abstraction is instantiated for use on particular devices or simulations by setting when executions occur, how communication is implemented and the contents of the environment. Figure 1(b) shows the particular instantiation in Alchemist.

3 The demo

The goal of this demo is to showcase Protelis through a series of Alchemist-backed simulations, starting from simple aggregate computations and incrementally adding complexity up to the support of mobile code. The simplest, introductory example will be the computation, on a network of mobile devices, of the distance towards an area in space called “source”. This distributed data structure, called gradient, is both simple and of paramount importance, since many coordination algorithms that run on mesh networks rely on it in order to carry on more complex computation. A more complex example will be the showcase of mobile code on a real-world scenario, in which different versions of a crowd detection program are injected in random points of a network of devices, and their update will be observable. Also, there will be an example of agents computing and sharing a plan. A group of rescuers will be deployed in a urban scenario in search of hurts needing a triage. Each of them has a pre-defined, waypoint based plan. At runtime, agents will be able to choose between the original plan or another one, computed based on what each other agent has discovered about the surrounding environment.

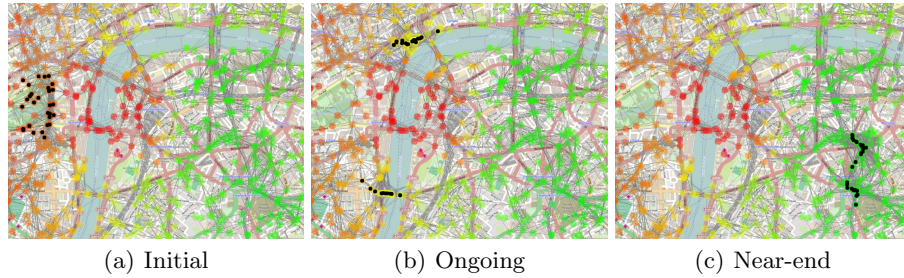


Fig. 2. Three snapshots of an example simulation where people are steering towards the most convenient path. The path emerges as a spatial distortion of a gradient.

Figure 2 shows a possible screenshot sequence, a video is available at <https://vid.me/gslm>.

References

1. J. Bachrach and J. Beal. Building spatial computers. Technical Report MIT-CSAIL-TR-2007-017, MIT, March 2007.
2. J. Beal, M. Viroli, and F. Damiani. Towards a unified model of spatial computing. In *7th Spatial Computing Workshop (SCW 2014)*, AAMAS 2014, Paris, France, May 2014.
3. F. Damiani, M. Viroli, D. Pianini, and J. Beal. Code mobility meets self-organisation: A higher-order calculus of computational fields. In S. Graf and M. Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems*, volume 9039 of *Lecture Notes in Computer Science*, pages 113–128. Springer International Publishing, 2015.
4. M. Viroli and F. Damiani. A calculus of self-stabilising computational fields. In eva Kühn and R. Pugliese, editors, *Coordination Languages and Models*, volume 8459 of *LNCS*, pages 163–178. Springer-Verlag, June 2014. Proceedings of the 16th Conference on Coordination Models and Languages (Coordination 2014), Berlin (Germany), 3-5 June. Best Paper of Discotec 2014 Federated conference.
5. M. Viroli, F. Damiani, and J. Beal. A calculus of computational fields. In C. Canal and M. Villari, editors, *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Sci.*, pages 114–128. Springer Berlin Heidelberg, 2013.
6. R. L. Wainwright, J. M. Corchado, A. Bechini, and J. Hong, editors. *Protelis: Practical Aggregate Programming*, Salamanca, Spain, 2015. ACM.