

Protelis: Practical Aggregate Programming

Danilo Pianini, Mirko Viroli
Alma Mater Studiorum – Università di Bologna, Italy
{danilo.pianini,mirko.viroli}@unibo.it

Jacob Beal
BBN Technologies, USA
jakebeal@bbn.com

ABSTRACT

The notion of a *computational field* has been proposed as a unifying abstraction for developing distributed systems, focusing on the computations and coordination of *aggregates* of devices instead of individual behavior. Prior field-based languages, however, have suffered from a number of practical limitations that have posed barriers to adoption and use. We address these limitations by introduction of Protelis, a functional language based on computational fields and embedded in Java, thereby enabling the construction of widely reusable components of aggregate systems. We demonstrate the simplicity of Protelis integration and programming through two examples: simulation of a pervasive computing scenario in the Alchemist simulator [24], and coordinated management of a network of services.

Categories and Subject Descriptors

D.3.2 [Software Engineering]: Languages—*Concurrent, distributed, and parallel languages*

General Terms

LANGUAGES

Keywords

Aggregate Programming, Computational Field, Field Calculus, Alchemist, Coordination

1. INTRODUCTION AND BACKGROUND

Coordinating the behavior of large numbers of computational devices or services is a problem that is continuing to grow in importance. The density of embedded or portable devices in our environment is continuing to increase. Phones, watches, clothes and accessories, vehicles, signs, buildings, displays: all these and much more are becoming capable of sensing, computing, and communicating. This “pervasive continuum” [31] is studied under a number

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC'15 April 13-17, 2015, Salamanca, Spain.
Copyright 2015 ACM 978-1-4503-3196-8/15/04...\$15.00.
<http://dx.doi.org/xx.xxxx/xxxxxx.xxxxxx>

of names, including pervasive computing, smart cities, and the Internet of Things. In all cases, though, it is expected to host many computations that are *(i)* context-dependent (many interactions will be opportunistic and involve devices in physical proximity, so that computations involve local data and situation); and *(ii)* self-adaptive and self-organizing (due to scale, they must spontaneously recover from faults and deal with unexpected contingencies).

At the same time, “traditional” networks are also increasing in scale and importance for enterprises both large and small. In an increasingly information-dependent and interconnected world, the rising cost of managing and maintaining such systems is driving a search for solutions that can increase the autonomy of computing systems, enabling them to act more as collective services than individual machines [12, 14].

In both of these cases, and a number of other areas facing similar challenges (e.g., large-scale sensor networks, multi-UAV control), there is a growing recognition that new engineering paradigms are needed, which will allow such systems to be effectively programmed at the aggregate level. In other words, languages and APIs that allow a distributed collection of devices to be programmed in terms of their collective behaviors, while leaving implicit many of the details of how such coordination is actually implemented.

A large number of aggregate programming approaches have been proposed, including such diverse approaches as abstract graph processing (e.g., [13]), declarative logic (e.g., [1]), map-reduce (e.g., [10]), streaming databases (e.g., [17]), and knowledge-based ensembles (e.g., [23])—for a detailed review, see [5]. Most aggregate programming approaches, however, have been too specialized for particular assumptions or applications to be able to address the complex challenges of these emerging environments.

Recently, however, a unifying model based on *computational fields* has been identified as a generalization of a wide range of existing approaches (e.g., [3, 19, 22, 25, 18, 21, 30, 22]). Formalized as the computational field calculus [27], this universal language appears to provide a theoretical foundation on which effective general aggregate programming platforms can be built.

This paper takes the next step, presenting Protelis, a practical aggregate programming language and implementation with the following features: *(i)* a functional paradigm with a familiar syntax for imperative-style function body specification, *(ii)* full interoperability with the Java type-system and API, *(iii)* complete implementation of the field calculus, and *(iv)* first-order functions to enhance flexibility and ex-

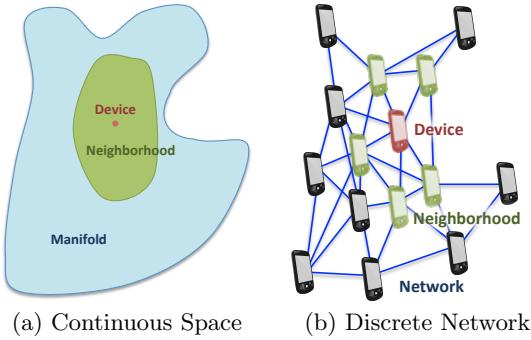


Figure 1: Computational field models originate from approximation of continuous space (a) with discrete networks of devices (b).

press code mobility patterns. We demonstrate the simplicity of Protelis integration and programming through two examples: simulation of a pervasive computing scenario in the Alchemist simulator [24], and coordinated management of a network of services.

The remainder of this paper is organized as follows: Section 2 reviews key concepts in aggregate programming and motivates the need for the new language, Section 3 describes the architectural choices for implementing Protelis, Section 4 introduces the Protelis language, Section 5 shows relevant application examples, and Section 6 provides final remarks.

2. AGGREGATE PROGRAMMING

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

Aggregate programming generally attempts to address this problem by providing composable abstractions separating these aspects: i) device-to-device communication is typically made entirely implicit, with higher-level abstractions for controlling efficiency/robustness trade-offs; ii) distributed coordination methods are encapsulated as aggregate-level operations (e.g., measuring distance from a region, spreading a value by gossip, sampling a collection of sensors at a certain resolution in space and time); and iii) the overall system is specified by composing aggregate-level operations, and this specification is then transformed into a complete distributed implementation by a suitable mapping.

Many aggregate programming applications involve collections of spatially embedded devices [5], where geometric operations and information flow provide a useful source of aggregate-level abstractions. From this, a large number of different methods have been developed (e.g., [3, 19, 22, 25, 18, 21, 30, 22]), all based on viewing the collection of devices as an approximation of a continuous field (Figure 1).

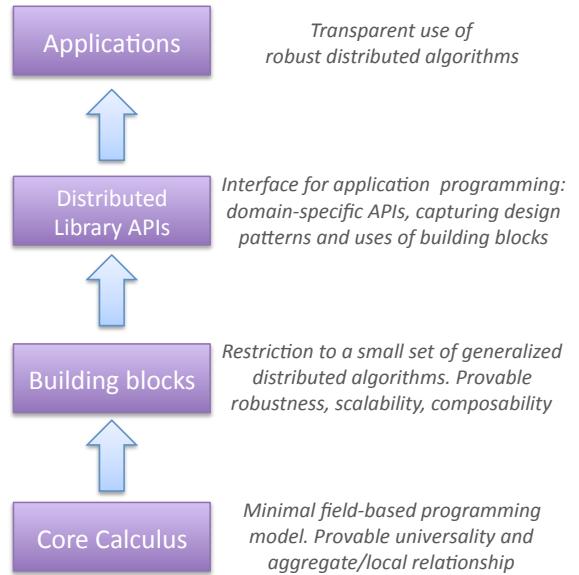


Figure 2: Layered approach for development of spatially-distributed systems via aggregate programming.

Recently, a minimal core calculus of computational fields [27], has been derived from the commonalities of these methods. The field calculus is based on the notion of a *computational field*—a map from devices comprising the system to (possibly structured) values, which is treated as unifying first-class abstraction to model system evolution and environment dynamics.

Critically, although originally derived from continuous-space concepts, the calculus does not depend on them and is applicable to any network. Field calculus is expressive enough to be a universal computing model [7] but terse enough to enable a provable mapping from aggregate specifications to equivalent local implementations.

Such a core calculus, however, is only the first step toward practical applications. In practice, effective aggregate programming for real-world distributed applications is likely to require a layered approach such as the one depicted in Figure 2. Building on the foundation of field calculus, the next layer uses the calculus to express a collection of higher-level “building block” algorithms, each a simple and generalized basis element of an “algebra” of programs with desirable resilience properties (e.g., the operators presented in [6]). On top of this, higher-level library APIs can be built, enabling simple and transparent construction of robust distributed systems.

Field calculus [27] hence provides a key theoretical and methodological foundation for aggregate programming. Its aim is to provide a universal model that is suitable for mathematical proofs of general properties about aggregate programming and the aggregate/local relationship, just as λ -calculus [9] provides for functional programming, π -calculus for parallel programming [20], or Featherweight Java [15] for Java-like object-oriented programming.

In the field calculus, everything is a field: computational fields are used to model every aspect of distributed computation, including input from sensors, network struc-

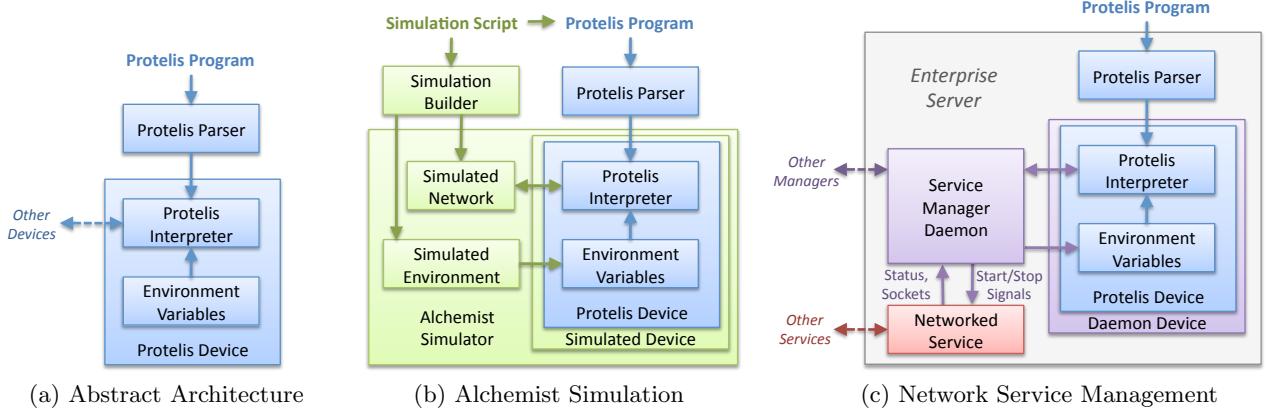


Figure 3: In the abstract Protelis architecture (a), an interpreter executes a pre-parsed Protelis program at regular intervals, communicating with other devices and drawing contextual information from a store of environment variables. This is instantiated by setting when executions occur, how communication is implemented and the contents of the environment. Two such instantiations are presented in this paper: as a simulation in the Alchemist framework (b) and as a daemon for coordinating management of networked services (c).

ture, environment interactions, distributed computations (e.g. progressive aggregation and spreading processes), and output for actuators. In particular, field calculus is constructed using five basic constructs: (i) function definition and evaluation, (ii) “built-in” operations for stateless local computation, e.g., addition, reading a sensor, (iii) a time-evolution construct which allows for stateful computation, (iv) a neighbor-value construct that creates a field of values from a device’s neighbors, and (v) a restriction operator to select which computations to perform in various regions of space and time. The specifics of these functions will be further elaborated in Section 4, when we present their implementation in Protelis.

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

The minimal syntax of field calculus has allowed its semantics, including proper coherence of device interactions, to be proven correct and consistent [27]. Additionally, despite its definition in terms of discrete semantics, field calculus is also space-time universal [7], meaning that it can approximate any field computation, either discrete or continuous, with arbitrary precision given a dense enough network of devices.

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

Like any other core calculus, however, the field calculus is less a practical programming language than a theoretical

framework. Likewise, while some of the prior approaches on which field calculus is based provide very similar semantics (most notably [3]), they all suffer from some combination of design and implementation problems that render them impractical for widespread adoption. Some, such as [28], have only minimal implementation, while others are more complete. Of the set, Proto [3] is probably the closest to a practical programming environment, but it still lacks many features expected in a modern programming language and has an implementation encumbered by a number of obsolete considerations that make it difficult to maintain and extend. Hence, we judged it best to develop a new language based on field calculus, designed from the ground up to support the development of complex applications as part of a modern software development ecosystem.

3. 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 [2]. Figure 3(a) shows the abstract architecture for Protelis virtual devices. First, a parser translates a text Protelis program into a valid representation of field calculus semantics. This is then executed by a Protelis interpreter at regular intervals, communicating with other devices and drawing contextual information from environment variables implemented as a tuple store of $(token, value)$ pairs. This abstraction is instantiated for use on particular devices or simulations by setting when executions occur, how communication is im-

plemented and the contents of the environment.

We have chosen to implement this architecture in Java. One key reason for this choice is that Java is highly portable across systems and devices. Another key reason (discussed further in the next section) is that Java’s reflection mechanisms make it easy to import a large variety of useful libraries and APIs for use in Protelis. Finally, the pragmatics of execution on embedded devices have also changed significantly since the publication of [2]: a much wider variety of low cost embedded devices are now capable of supporting Java, while at the same time improvements in Java implementations have made it much more competitive in speed and resource cost with low-level languages like C.

In particular, we have chosen to implement Protelis and its architecture via the Xtext language generator [11] and within the generic Alchemist framework [24]. Usefully, Xtext also features support for generating a language-specific Eclipse plug-in, which provides developer assistance through code highlighting, completion suggestions, and compile-time error detection.

For an initial validation, we have exercised this architecture by construction of two instantiations: one in the Alchemist framework for simulation of large-scale spatially-embedded systems; the other as a daemon for coordinating management of networked services. Figure 3(b) shows the Alchemist instantiation: simulations are configured using a simple scripting language, which specifies a Protelis program as well as the collection of devices that will execute it, communication between those devices, and other aspects of the environment to be simulated. The Alchemist event-driven simulation engine then handles execution scheduling, message delivery, and updates to the environment tuple store. Figure 3(c) shows the network service management instantiation. Here, each Protelis device lives on a separate server in an enterprise network, and is tethered to the networked service it is intended to manage by a service manager daemon. This daemon monitors the service, injecting information about its status and known dependencies into the environment and maintaining a neighborhood by opening parallel communication links to the corresponding daemons on any other servers that the monitored service communicates with. Examples using each of these implementations are shown in Section 5.

4. PROTELIS

We have designed the Protelis language as an implementation of the field calculus [27] closely related to Proto [3]. On the one hand, it incorporates the main spatial computing features of the field calculus, hence enjoying its universality, consistency, and self-stabilization properties [7, 26]. On the other hand, it turns the field calculus into a modern specification language, improving over Proto by providing i) access to a richer API through Java integration, ii) support for code mobility through first-order functions, iii) a novel syntax inspired by the more widely adopted C-family languages. Protelis is freely available and open source, and can be downloaded as part of the Alchemist distribution¹.

4.1 Syntax

We present the Protelis language in terms of its abstract syntax, provided in Figure 4 as a means to guide the discus-

$P ::= \bar{I} \bar{F} \bar{s};$;; Program
$I ::= \mathbf{import} \ m \mid \mathbf{import} \ m.*$;; Java import
$F ::= \mathbf{def} \ f(\bar{x}) \ \{\bar{s};\}$;; Function definition
$s ::= e \mid \mathbf{let} \ x = e \mid x = e$;; Statement
$w ::= x \mid 1 \mid [\bar{w}] \mid f \mid (\bar{x}) \rightarrow \{\bar{s};\}$;; Variable/Value
$e ::= w$;; Expression
$\mid b(\bar{e}) \mid f(\bar{e}) \mid e.\mathbf{apply}(\bar{e})$;; Fun/Op Calls
$\mid e.m(\bar{e}) \mid \#a(\bar{e})$;; Method Calls
$\mid \mathbf{rep}(x \leftarrow w)\{\bar{s};\}$;; Persistent state
$\mid \mathbf{if}(e)\{\bar{s};\} \mathbf{else} \{\bar{s}'\}$;; Exclusive branch
$\mid \mathbf{mux}(e)\{\bar{s};\} \mathbf{else} \{\bar{s}'\}$;; Inclusive branch
$\mid nbr\{\bar{s};\}$;; Neighborhood values

Figure 4: Protelis abstract syntax, colored to emphasize definition and application (red), functions (blue), variables (green), and special field calculus operators (purple).

sion of the language’s features. This syntax uses similar conventions to well-known core languages like FJ [15]. We let meta-variable f range over names of user-defined functions, x over names of variables and function arguments, 1 over literal values (Booleans, numbers, strings), b over names of built-in functions and operators (including the “hood” functions described in Section 4.3), m over Java method names, and $\#a$ over aliases of static Java methods. All such meta-variables are used as non-terminal symbols in Figure 4. Overbar notation \bar{y} generally means a comma-separated list y_1, \dots, y_n of elements of kind y , with the two exceptions that in \bar{F} we use no comma separator, and in \bar{s} ; semi-colon is used as separator instead.

4.2 Ordinary Language Features

One of the distinctive elements of Protelis when compared to other aggregate programming languages (particularly Proto), is the adoption of a familiar C- or Java-like syntax, which can significantly reduce barriers to adoption. Despite this syntactic similarity, Protelis is a purely functional language: a program is made of a sequence of function definitions ($F_1 \dots F_n$), modularly specifying reusable parts of system behavior, followed by a main block of statements. Following the style of C-family languages, a function’s body is a sequence of statements surrounded by curly brackets. As in the Scala programming language², however, statements can also be just expressions, and a statement sequence evaluates to the result of the last statement. Each statement is an expression to be evaluated (e), possibly in the context of the creation of a new variable ($\mathbf{let} \ x = e$) or a re-assignment ($x = e$)³. As an example, consider the following function taking four fields as parameters, after the “channel” pattern from [8]:

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

This function assumes that its input $distA$ maps each device to its distance to a region A , $distB$ maps each device to its

²<http://www.scala-lang.org>.

³Technically, “re-assignment” is actually the creation of a new variable that shadows the old.

¹<https://bitbucket.org/danysk/alchemist>

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

Atomic expressions w can be literal values (1), variables (x), tuples ($[w]$), function names (f) or lambdas ($(\bar{x}) \rightarrow \{s\}$). Structured expressions include three kinds of “calls”: (i) $b(\bar{e})$ is application to arguments \bar{e} of a built-in operation b , which could be any (infix- or prefix-style) mathematical, logical or purely algorithmic function⁴; (ii) $f(\bar{e})$ is application of a user-defined function; and (iii) $e.\text{apply}(\bar{e})$ is application of arguments to an expression e evaluating to a lambda or function name. The following shows examples of such calls:

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

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

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

4.3 Special Field Calculus Operators

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

- Construct `rep(x<-w){s;}` defines a locally-visible variable x initialized with w and updated at each computation round with the result of executing body $\{s;\}$: it provides a means to define a field evolving over time according to the update policy specified by $\{s;\}$.
- Construct `nbr{s;}` executed in a device gathers a map (actually, a field) from all neighbors (including the device itself) to their latest value from computing \bar{s} . A special set of built-in “hood” functions can then be used to summarize such maps back to ordinary expressions. For example, `minHood` finds the minimum value in the map.
- The branching constructs `mux(e){s;} else {s'()}` and `if(e){s;} else {s'()}` perform two critically different forms of branching. The `mux` construct is an inclusive “multiplexing” branch: the two fields obtained by

⁴For simplicity of presentation, we omit the syntax for infix operations and order of operations, which is closely patterned after Java.

computing \bar{s} and \bar{s}' are superimposed, using the former where e evaluates to true, and the second where e evaluates to false. Complementarily, `if` performs an exclusive branch: it partitions the network into two regions: where e evaluates to true \bar{s} is computed, and elsewhere \bar{s}' is computed instead.

The following code shows some example uses of these constructs:

```
def count() { rep(x<-0){ x + 1 } }
def maxh(field) { maxHood(nbr{field}) }
def distanceTo(source) {
  rep(d <- Infinity) {
    mux (source) { 0 }
    else { minHood(nbr{d} + nbrRange) }
  }
}
def distanceToWithObstacle(source,obstacle) {
  if (obstacle) { Infinity } else { distanceTo(source) }
}
```

Function `count` yields an evolving field, counting how many computation rounds have been executed in each device. Function `maxh` yields a field mapping each device the maximum value of `field` across its neighborhood—note a `nbr` construct should always be eventually wrapped inside a “hood” function. Function `distanceTo` nests `nbr` inside `rep` to create a chain of interactions across many hops in the network, computing minimum distance from any device to the nearest “source device” (i.e., where `source` holds `true`). It does so by a field d initially `Infinity` everywhere, and evolving as follows: d is set to 0 on sources by `mux`, and elsewhere takes the minimum across neighbors of the values obtained by adding to d the estimated distance to the current device—a triangle inequality relaxation computing a distance field also often termed *gradient* [16, 4, 25]. Finally, function `distanceToWithObstacle` shows exclusive branch at work: `distanceTo(source)` is computed in the sub-region where there is no obstacle, which causes the computation of distances to implicitly circumvent such obstacles.

5 APPLICATION EXAMPLES

To demonstrate how these features combine to offer simple programming of complex distributed algorithms across a potentially broad range of applications domains, we now present two example applications. The first aims at a pervasive computing scenario and is executed in simulation using Alchemist [24], the second aims at enterprise network management and is executed on a collection of EmuLab [29] servers.

5.1 Example: Rendezvous at a Mass Event

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

```
// Follow the gradient of a potential field down from a source
def descend(source,potential) {
  rep(path <- source) {
    let nextStep = minHood(nbr([potential, self.getId()]));
    if (nextStep.size() > 1) {
```

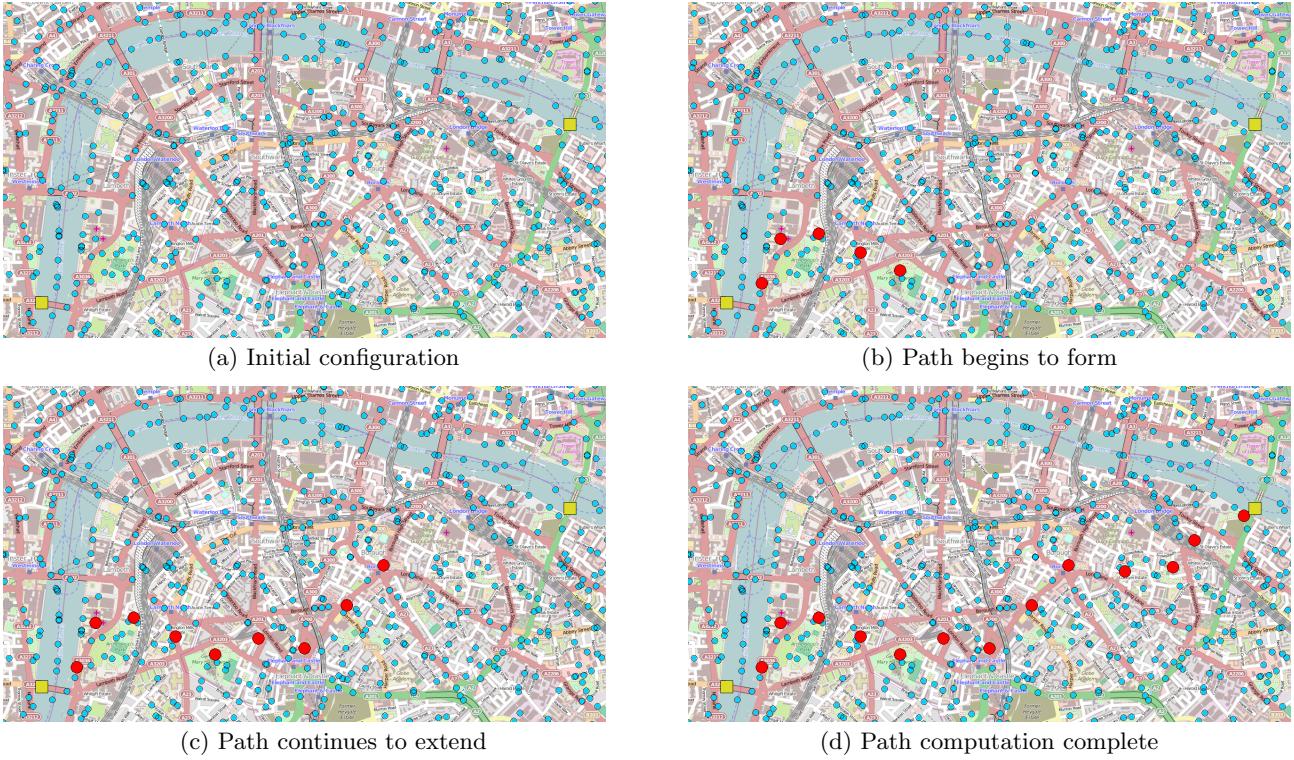


Figure 5: Example of computing a rendezvous route for two people in a crowded urban environment.

```

let candidates = nbr([nextStep.get(1), path];
  source || anyHood([self.getId(), true] == candidates)
} else {
  source
}
}

def rendezvous(person1, person2) {
  descend (person1 == owner, distanceTo(person2 == owner))
}

// Example of using rendezvous
rendezvous("Alice", "Bob");
    
```

Figure 5 shows an example of running this rendezvous process in a simulated city center. We chose London as a simulation environment, using Alchemist's capability for importing OpenStreetMap data. We displaced 1000 devices randomly across the city streets (represented by pale blue dots), with a communication range of 475 meters (this range chosen to ensure no network segmentation). We then picked two devices whose owners want to meet: one device on Lambeth Bridge (lower left of the image) and one device on Tower Bridge (upper right), each marked with a yellow square. To mark the devices for Protelis, we injected their environments with a property `owner`, assigning the strings "`Alice`" and "`Bob`" as values for the first and the second device respectively.

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

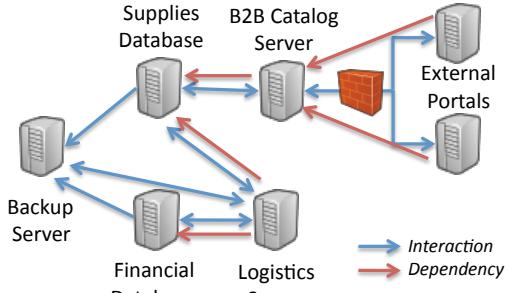
tion, given a device and a potential field, builds a path of devices connecting the former with the source of the latter. The strategy is to mark the device we want to connect to the potential field's source as part of the path, and then, in every device, compute which of the neighbors is closest to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

As Figure 5 shows, once the simulation starts, a chain of devices is rapidly identified (red dots), marking a sequence of way-points for both device owners to walk in order to meet in the middle. Note also that, due to the ongoing nature of the computation, if one of the device owners moves in a different direction instead, the path will automatically adjust so that it continues to recommend the best path for rendezvous.

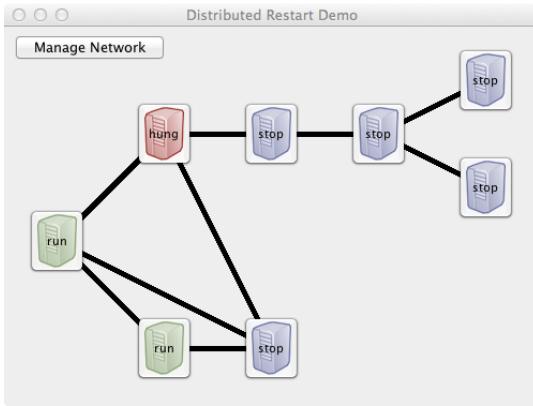
5.2 Example: Network Service Management

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

Thus, responding to a service failure often requires a coor-



(a) Example Dependent Services Scenario



(b) Example of Coordinated Restart Execution

Figure 6: (a) An example scenario of an enterprise network for a small manufacturing and supply company. (b) Example of execution on a network of 8 EmuLab [29] machines: the supplies database has crashed (red), and so all dependent services have shut themselves down (blue), while other services continue to run normally (green).

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

Figure 6(a) shows an example scenario of an enterprise network for a small manufacturing and supply company, with dependencies between two key databases and the internal and external servers running web applications. This scenario was implemented on a network of EmuLab [29] servers. The services were emulated as simple query-response networking programs in Java that entered a “hung” state either upon being externally triggered to crash or after their queries began to consistently fail.

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

in order to maintain the set of neighbors for Protelis, and allows an external monitoring application to attach and receive status reports.

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

```

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

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

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

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

```

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

Figure 6(b) shows a typical screenshot of the network of services in operation on an EmuLab network of Ubuntu machines, one service per machine, as visualized by the monitoring application. In this screenshot, the supplies database has crashed, causing many of the other services to gracefully shut themselves down. As soon as the supplies database is restarted, however, the rest of the services automatically bring themselves up in dependency order.

6. CONCLUSIONS

This paper has presented Protelis, a new language intended to provide a practical and universal platform for aggregate programming. Protelis ensures universality and coherence between aggregate specification and local execution by building atop the field calculus introduced in [27]. At the same time, accessibility, portability, and ease of integration are ensured by embedding Protelis within Java. This enables Protelis programs to draw on the full breadth of available Java APIs and to readily integrate with a wide range of devices and applications, as illustrated by our examples of pervasive computing simulation and networked service management. This implementation of Protelis thus forms an important component of the toolchain necessary for practical application of aggregate programming principles and methods to address real-world problems. The Protelis framework continues to be actively developed: we plan to enrich it in the future by adding higher-level abstractions for aggregate programming grounded on the mechanisms discussed in this work.

7. ACKNOWLEDGMENTS

This work is partially supported by the United States Air Force and the Defense Advanced Research Projects Agency (DARPA) under Contract No. FA8750-10-C-0242. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views, opinions, and/or findings contained in this article/presentation are those of the author(s)/presenter(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)

8. REFERENCES

- [1] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai. Meld: A declarative approach to programming ensembles. In *IEEE Intelligent Robots and Systems (IROS)*, pages 2794–2800, 2007.
- [2] J. Bachrach and J. Beal. Building spatial computers. Technical Report MIT-CSAIL-TR-2007-017, MIT, March 2007.
- [3] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.
- [4] J. Beal, J. Bachrach, D. Vickery, and M. Tobenkin. Fast self-healing gradients. In *Proceedings of ACM SAC 2008*, pages 1969–1975, 2008.
- [5] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. In M. Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, pages 436–501. IGI Global, 2013.
- [6] J. Beal and M. Viroli. Building blocks for aggregate programming of self-organising applications. In *2nd Focas Workshop on Fundamentals of Collective Systems*, pages 1–6. IEEE CS, to appear, 2014.
- [7] J. Beal, M. Viroli, and F. Damiani. Towards a unified model of spatial computing. In *7th Spatial Computing Workshop*, AAMAS 2014, Paris, France, May 2014.
- [8] W. Butera. *Programming a Paintable Computer*. PhD thesis, MIT, Cambridge, MA, USA, 2002.
- [9] A. Church. A set of postulates for the foundation of logic. *Annals of mathematics*, 33(2):346–366, 1932.
- [10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
- [11] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *OOPSLA*, pages 307–309. ACM, 2010.
- [12] T. Eze, R. Anthony, C. Walshaw, and A. Soper. Autonomic computing in the first decade: trends and direction. In *8th Int'l Conf. on Autonomic and Autonomous Systems*, pages 80–85, 2012.
- [13] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *DCOSS*, pages 126–140, 2005.
- [14] F. Hu, M. Qiu, J. Li, T. Grant, D. Taylor, S. McCaleb, L. Butler, and R. Hamner. A review on cloud computing: Design challenges in architecture and security. *CIT. Journal of Computing and Information Technology*, 19(1):25–55, 2011.
- [15] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. Sys.*, 23(3), 2001.
- [16] F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Trans. Softw. Eng.*, 13(1):32–38, 1987.
- [17] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. In *ACM TODS*, 2005.
- [18] M. Mamei and F. Zambonelli. Self-maintained distributed tuples for field-based coordination in dynamic networks. *Concurrency and Computation: Practice and Experience*, 18(4):427–443, 2006.
- [19] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. on Software Engineering Methodologies*, 18(4):1–56, 2009.
- [20] R. Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, 1999.
- [21] R. Nagpal. *Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics*. PhD thesis, MIT, Cambridge, MA, USA, 2001.
- [22] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Wkshp. on Data Mgmt. for Sensor Networks*, pages 78–87, 2004.
- [23] R. D. Nicola, M. Loreti, R. Pugliese, and F. Tiezzi. A formal approach to autonomic systems programming: The scel language. *ACM Trans. Auton. Adapt. Syst.*, 9(2):7:1–7:29, July 2014.
- [24] D. Pianini, S. Montagna, and M. Viroli. Chemical-oriented simulation of computational systems with Alchemist. *Journal of Simulation*, 2013.
- [25] M. Viroli, M. Casadei, S. Montagna, and F. Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Trans. Auton. and Adap. Syst.*, 6(2):14:1 – 14:24, 2011.
- [26] M. Viroli and F. Damiani. A calculus of self-stabilising computational fields. In *Coordination 2014*, pages 163–178, June 2014.
- [27] 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 *Comm. in Comp. and Info. Sci.*, pages 114–128. Springer Berlin Heidelberg, 2013.
- [28] M. Viroli, D. Pianini, and J. Beal. Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In *Coordination 2012*, pages 212–229, June 2012.
- [29] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, pages 255–270, Boston, MA, Dec. 2002.
- [30] D. Yamins. *A Theory of Local-to-Global Algorithms for One-Dimensional Spatial Multi-Agent Systems*. PhD thesis, Harvard, December 2007.
- [31] F. Zambonelli et al. Self-aware pervasive service ecosystems. *Procedia CS*, 7:197–199, 2011.