Aggregate Programming: From Foundations to Applications

Jacob Beal[1(✉)] and Mirko Viroli[2]

[1] Raytheon BBN Technologies, Cambridge, MA 02138, USA
jakebeal@bbn.com
[2] Alma Mater Studiorum–Università di Bologna, Cesena, Italy

Abstract. We live in a world with an ever-increasing density of computing devices, pervading every aspect of our environment. Programming these devices is challenging, due to their large numbers, potential for frequent and complex network interactions with other nearby devices, and the open and evolving nature of their capabilities and applications. Aggregate programming addresses these challenges by raising the level of abstraction, so that a programmer can operate in terms of collections of interacting devices. In particular, field calculus provides a safe and extensible model for encapsulation, modulation, and composition of services. On this foundation, a set of resilient "building block" operators support development of APIs that can provide resilience and scalability guarantees for any service developed using them. We illustrate the power of this approach by discussion of several recent applications, including crowd safety at mass public events, disaster relief operations, construction of resilient enterprise systems, and network security.

Keywords: Aggregate programming · Pervasive computing · Field calculus · Distributed systems · Domain-specific languages

1 Introduction

For some time now, our world has been undergoing a dramatic transition in how we relate to computing, as the number of computing devices rises and more and more of these devices become embedded into our environment (Fig. 1). In the past, it was reasonable to use a programming model that focused on the individual computing device, and its relationship with one or more users. Now, however, it is typically the case that many computing devices are involved in the provision of any given service, and that each machine may participate in many overlapping instances of such collective services. Moreover, the increasing mobility and wireless capabilities of some computing devices (e.g., wearable devices, smart phones, car systems, drones, electronic tags, etc.), means that many devices have the opportunity to accomplish part or all of their assigned tasks through peer-to-peer local interactions, rather than by going through fixed infrastructure such as cellular wireless or the Internet, thereby lowering latency

M. Bernardo et al. (Eds.): SFM 2016, LNCS 9700, pp. 233–260, 2016.
DOI: 10.1007/978-3-319-34096-8_8

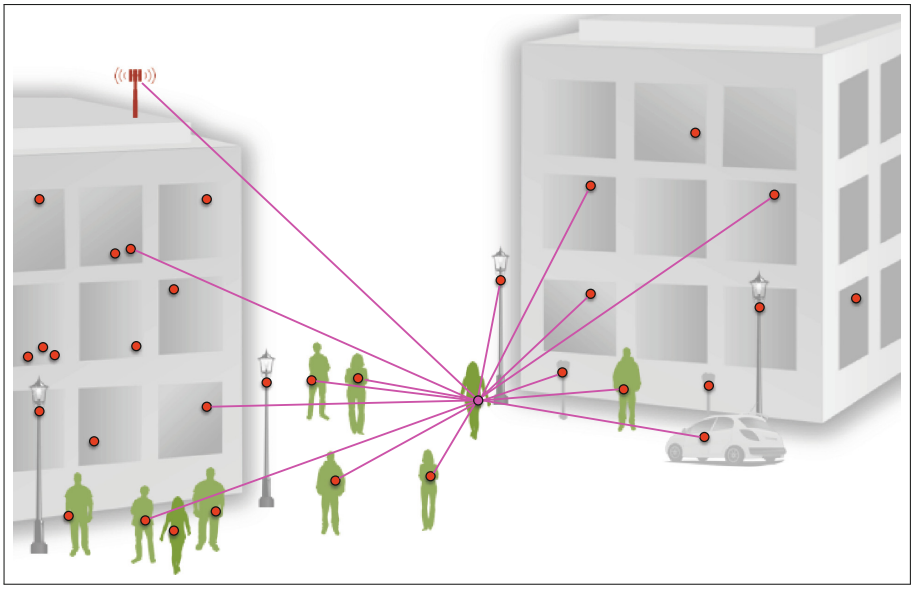


Fig. 1. Our world is increasingly filled with large numbers of computing devices, embedded into the environment and with many opportunities for local interaction as well as for more traditional location-agnostic interactions over fixed network infrastructure. Figure adapted from [8].

and increasing resilience to issues with inadequate or unavailable infrastructure, e.g., during civic emergencies or mass public events.

To effectively program such systems, we need to be able to reliably engineer collective aggregate behaviors. Ordinary programming approaches typically focus on individual devices, entangling application design with various aspects of distributed system design (e.g., efficient and reliable communication, robust coordination, composition of capabilities, etc.), as well as confronting the programmer with the notoriously difficult and generally intractable "local-to-global" problem of generating a specified emergent collective behavior from the interactions of individual devices. These problems tend to limit our ability to make use of the potential of the modern computing environment, as complex distributed services developed using device-centric programming paradigms tend to suffer from design problems, lack of modularity and reusability, deployment difficulties, and serious test and maintenance issues.

Aggregate programming provides an alternate approach, which simplifies the design, creation, and maintenance of complex distributed systems by raising the abstraction level from individual devices to potentially large aggregations of devices. This survey presents an introduction to aggregate programming and a survey of key points on the current state of the art, updating and synthesizing several prior surveys [7,8,10,11]. Aggregate programming has roots in many different communities, all of which have encountered their own versions

of the aggregate programming problem and which have between them developed a vast profusion of domain-specific programming models to address it, which are briefly surveyed in Sect. 2. Recently, however, there have been a number of unifying results regarding field-based computational models, which are reviewed in Sect. 3. These results lay the foundation for a more principled approach, in which general mechanisms for roboust and adaptive coordination are composed and refined to build domain-specific APIs, following the layered engineering approach reviewed in Sect. 4. Ultimately, this can provide distributed systems engineers with a simple interface for development of safe, resilient, and scalable distributed applications, some examples of which are presented in Sect. 5 before turning to discussion of future directions in Sect. 6.

2 Background and General Approach

In many ways, aggregate programming is not a new idea: the importance of raising the abstraction level for distributed programming has been recognized previously in a number of different fields, motivating work toward aggregate programming across a variety of domains, including biology, reconfigurable computing, high-performance computing, sensor networks, agent-based systems, and robotics, as surveyed in [7].

Despite the wide degree of heterogeneity in applications and context across these antecedents, the common problems in organizing aggregates have led such approaches to cluster around a few main strategies: making device interaction implicit (e.g., TOTA [31], MPI [32], NetLogo [41], Hood [47]), providing means to compose geometric and topological constructions (e.g., Origami Shape Language [33], Growing Point Language [17], ASCAPE [28]), providing means for summarizing from space-time regions of the environment and streaming these summaries to other regions (e.g., TinyDB [30], Regiment [34], KQML [23]), automatically splitting computational behaviour for cloud-style execution (e.g., MapReduce [21], BOINC [2], Sun Grid Engine [25]), and providing generalizable constructs for space-time computing (e.g., Protelis [37], Proto [5], MGS [26]).

These many prior efforts have also evidenced some commonalities in their strengths and weaknesses, which suggest that, when programming large-scale distributed systems, it is useful to conform to the following three principles: *(i)* mechanisms for robust coordination should be hidden "under-the-hood" where programmers are not *required* to interact with them, *(ii)* composition of modules and subsystems must be simple, transparent, and with consequences that can be readily predicted, and *(iii)* large-scale distributed systems typically comprise a number of different subsystems, which need to use different coordination mechanisms for different regions and times.

From these observations and the commonalities amongst the various prior approaches has come the generalized approach that we discuss in this paper, based on field calculus [19,20,46] and its practical instantiation in Protelis [37], which takes the following view of distributed systems engineering:

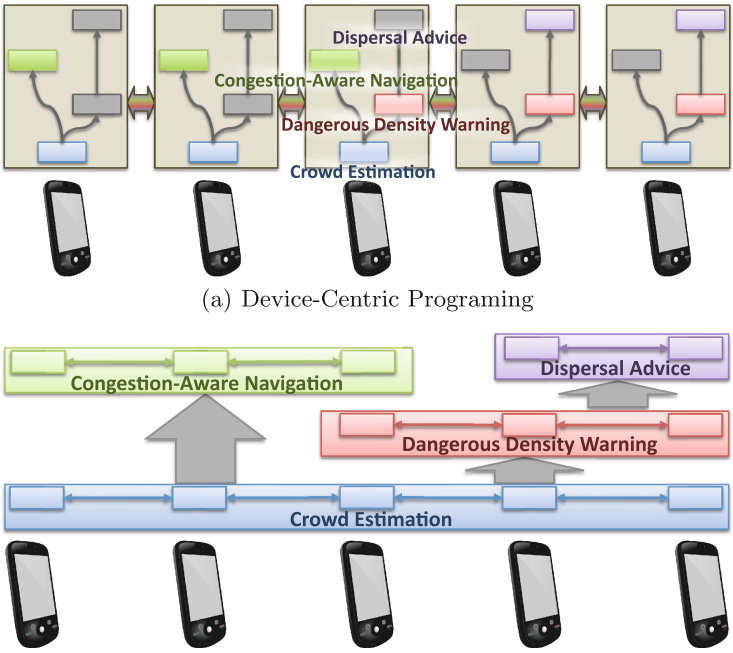


Fig. 2. Comparison of device-centric programming of distributed algorithms (a) versus aggregate programming (b): device-centric programming designs a distributed system in terms of the (often complex) behaviors and interactions of individual devices; with aggregate programming, simpler algorithmic building blocks can be scoped and composed directly for the aggregate. Figure adapted from [8,11].

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 manipulation of data constructs with extent across that region (where regions may be defined either regarding network structure or regarding continuous space and time); and
3. these manipulations are actually executed by the individual devices in the region, through local operations and interactions with (spatial or network) neighbors.

2.1 Example: Distributed Crowd Management

Consider, for example, the architecture of a crowd-safety service, such as might be distributed on the cell phones of people attending a very large public event, such as a marathon or a major city festival, as in the scenarios described in [3,36]. Figure 2 compares a traditional device-centric architecture versus an aggregate

programming approach to building a crowd-safety service with four functionalities: estimation of crowd density and distribution based on interaction between phones and observation of the local wireless environment, alerting of people in or near dangerously large and dense regions of the crowd (where there is risk of trampling or panic), providing advice for people in the interior of such regions on how to move to help disperse the dangers, and crowd-aware navigation that can help other people move around the event while simultaneously avoiding dangerous areas.

With traditional device-centric approaches (Fig. 2(a)), the programmer needs to simultaneously reason about composition of services within a device, protocols for local device interactions, and also about how the desired complex global behavior will be produced from such local interactions. With aggregate programming, on the other hand, the system can be readily approached in terms of a set of distributed modules. A programmer can then compose these modules incrementally to form complete applications simply by specifying where they should execute and how information should flow between them (Fig. 2(b)). Here, for example, crowd estimation produces as output a distributed data structure—a "computational field" [19,20]—mapping from location to crowd density, which is then an input for both crowd-aware navigation and for the alerting service. These then produce their own distributed data structures, respectively vectors for recommended travel and a map of warnings (which is in turn an input for producing dispersal advice). The details of protocol and implementation can then be automatically generated from such compositions of data structures and services. Aggregate programming thus promotes the construction of more complex, reusable, resilient, and composable distributed systems by separating the question of which services should be executed and where, from the implementation details of those services and their coordination.

2.2 Aggregate Programming Layers

Figure 3 shows how aggregate programming can hide the complexity of the underlying distributed network environment and the problems of distributed coordination with a sequence of abstraction layers. At the foundation of this approach is *field calculus* [19,20], a core set of constructs modeling device behavior and interaction, which is terse enough to enable mathematical proof of equivalence between aggregate specifications and local implementations, yet expressive enough to be universal. The notion of "computational field," adapted from physics, makes this particularly well suited for environments with devices embedded in space and communicating with others in close physical proximity, though it is more generally suitable for any sparsely connected network. Upon this foundation, we can identify key coordination "building blocks" with desirable resilience properties, each being a simple and generalized basis element generating a broad set of programs with desirable resilience properties. Finally, common patterns for using and composing these building blocks can be captured to produce both general and domain-specific APIs for common application needs like sensing, decision, and action, together forming a collective behavior API

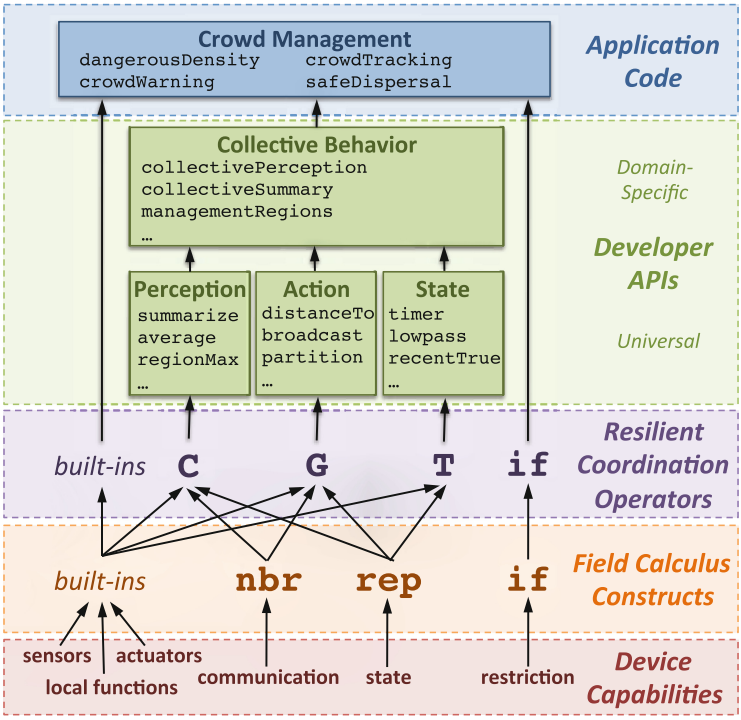


Fig. 3. Aggregate programming takes a layered approach to distributed systems development: the software and hardware capabilities of particular devices are abstracted by using them to implement a small universal calculus of aggregate-level field calculus constructs. This calculus is then used to implement a limited set of "building block" coordination operations with provable resilience properties, which are in turn wrapped and combined together to produce user-friendly APIs, both general and domain-specific, for developing distributed systems. Figure adapted from [8].

suitable for transparent implementation of complex networked services and applications [8,9,45].

Engineering distributed systems with this approach can thus allow construction of complicated resilient distributed systems with rather simple specifications, as we will see in the application examples in Sect. 5. From such a terse specification, the full complexity of the system is then automatically elaborated: from the set of resilient coordination operators that were chosen to be used, to the various ways in which resilience is actually achieved via particular building blocks or their functional equivalents, then how the aggregate specification implements each of those building blocks and maps to actions by individual devices, and finally how particular devices in the potentially heterogeneous network environment actually implement capabilities like sensing, communication, and localization.

Here, we discuss the incarnation of this approach using Protelis, a field calculus implementation with Java-like syntax and support for first-class aggregate functions. For full details on Protelis, see its presentation in [37].

3 Field Calculus

The *field calculus* [19,20,46] is an attempt to capture a set of essential features that appear across many different aggregate programing approaches. In particular, this "core calculus" approach captures these features in the syntax and semantics of a tiny programming language, expressive enough to be universal [12] yet small enough to be tractable for mathematical analysis. With regards to the overall view presented in Fig. 3, field calculus forms the second lowest layer, which is also the point where aggregate programming interfaces with the highly heterogeneous world of device infrastructure and non-aggregate software services (together comprising the lowest layer).

At its core is the notion of *computational field*, a widely-used space-time programming concept [7] inspired by the notion of fields in physics. In physics, a field is a function that maps each point in some space-time domain to a scalar value, such as the temperatures in a room, or a vector value, such as the currents in the ocean. Computational fields generalize this notion to allow the values to be arbitrary computational objects on either continuous or discrete domains, such as a set of messages to be delivered at each device in a network, or XML descriptors for a set of inventory items to be stocked on the shelves of a store.

Such spatially-extended fields, with values potentially dynamically changing over time, are then the basic "aggregate" units of values that may be distributed across many devices in the network. More precisely, a *field value* ϕ is a function $\phi : D \to \mathcal{L}$ that maps each device δ in domain D to a local value ℓ in range $\mathcal{L}$. Similarly, a *field evolution* is a dynamically changing field value, i.e., a function mapping each point in time to a field value (evolution is used here in the physics sense of "time evolution"). A field computation, then, is any function that takes field evolutions as input (e.g., from sensors or device information) and produces another field evolution as its output, from which field values are snapshots. For example, given an input of a Boolean field mapping a set of "source" devices to *true*, an output field containing the estimated distance from each device to the nearest source device can be constructed by iterative spreading and aggregation of information, such that the output always rapidly adjusts to the correct values for the current input and network structure. The field calculus [19,20] succinctly captures the essence of field computations, much as λ-calculus [14] does for functional computation and FJ [27] does for object-oriented programming.

3.1 Syntax of Field Calculus

Figure 4 presents field calculus syntax. Following the convention of [27], overbar notation denotes metavariables over sequences, e.g., $\overline{\mathsf{e}}$ is shorthand for the sequence of expressions $\mathsf{e}_1, \mathsf{e}_2, \ldots \mathsf{e}_n$ $(n \geq 0)$. A local value ℓ represents the

ℓ	::= $\texttt{c}\langle\overline{\ell}\rangle \mid \lambda$	local value
λ	::= $\texttt{o} \mid \texttt{f} \mid (\texttt{fun}\ (\overline{\texttt{x}})\ \texttt{e})$	function value
$\texttt{e}$	::= $\ell \mid \texttt{x} \mid (\texttt{e}\ \overline{\texttt{e}})$	expression
	$\mid (\texttt{rep}\ \texttt{x}\ \texttt{w}\ \texttt{e})$	
	$\mid (\texttt{nbr}\ \texttt{e})$	
	$\mid (\texttt{if}\ \texttt{e}\ \texttt{e}\ \texttt{e})$	
$\texttt{w}$	::= $\texttt{x} \mid \ell$	variable or local value
$\texttt{F}$	::= $(\texttt{def}\ \texttt{f}(\overline{\texttt{x}})\ \texttt{e})$	function declaration
$\texttt{P}$	::= $\overline{\texttt{F}}\ \texttt{e}$	program

Fig. 4. Syntax of (higher-order) field calculus, as presented in [20].

value of a field at a given device, which can be any *data value* $\texttt{c}\langle\ell_1, \cdots, \ell_m\rangle$ (written $\texttt{c}$ when $m = 0$), such as Booleans `true` and `false`, numbers, strings, or structured values like $\texttt{Pair}\langle 3, \texttt{Pair}\langle\texttt{false}, 5\rangle\rangle$ or $\texttt{Cons}\langle 2, \texttt{Cons}\langle 4, \texttt{Null}\rangle\rangle$. Such a value may also be a *function value* λ, i.e. a built-in operator `o`, a user-defined function `f`, or an anonymous function $(\texttt{fun}\ (\overline{\texttt{x}})\ \texttt{e})$, where $\overline{\texttt{x}}$ are arguments and `e` is the body, in which we assume no free variables exist. Finally, a device δ can also hold a *neighboring field value* ϕ, which maps each neighbor of δ to a local value ℓ (neighboring field values cannot be expressed directly, only appearing dynamically during computations such as with operator `nbr`, so they do not appear in the syntax).

The main entities of the calculus are expressions, each of which defines a field. An expression can be a local value ℓ, representing a field mapping its entire domain to value ℓ, a variable `x` used as function parameter or state variable, or a composite created using the following constructs:

- *Built-in operator call:* $(\texttt{e}\ \texttt{e}_1 \cdots \texttt{e}_n)$, where `e` evaluates to a "point-wise" built-in operator `o`, involving neither state nor communication, e.g. mathematical functions like addition, comparison, and sine, or an environment-dependent function such as reading a temperature sensor or the 0-ary `nbr-range` operator returning a neighboring field mapping each neighbor to an estimate of its current distance from δ. Expression $(\texttt{o}\ \texttt{e}_1 \cdots \texttt{e}_n)$ produces a field mapping each δ to the result of applying `o` to the values at δ of its $n \geq 0$ arguments $\texttt{e}_1, \ldots, \texttt{e}_n$.
- *User-defined function call:* $(\texttt{e}\ \texttt{e}_1 \cdots \texttt{e}_n)$, where `e` evaluates to a user-defined function `f`, with corresponding declaration $(\texttt{def}\ \texttt{f}(\texttt{x}_1\ \ldots\ \texttt{x}_n)\ \texttt{e})$. Evaluating $(\texttt{f}\ \texttt{e}_1 \cdots \texttt{e}_n)$ provides a standard (possibly recursive) call-by-value abstraction.
- *Anonymous function call:* $(\texttt{e}\ \texttt{e}_1 \cdots \texttt{e}_n)$, has the same semantics as user-defined function calls, except that `e` evaluates to an anonymous function $(\texttt{fun}\ (\texttt{x}_1 \cdots \texttt{x}_n)\ \texttt{e})$.
- *Time evolution:* $(\texttt{rep}\ \texttt{x}\ \texttt{w}\ \texttt{e})$ is a "repeat" construct for dynamically changing fields, using a model in which each device evaluates expressions repeatedly in asynchronous rounds. State variable `x` initialises to the value of `w`, then

updates at each step by computing `e` against the prior value of `x`. For instance, `(rep x 0 (+ x 1))` counts how many rounds have been computed at each device.
- *Neighboring field construction:* (`nbr e`) captures device-to-device interaction, returning a field ϕ that maps each device δ to a neighboring field value, which in turn maps each neighbor to its most recent available value of `e` (realizable e.g., via periodic broadcast). Such neighboring field values can then be manipulated and summarized with built-in operators, e.g., `(min-hood (nbr e))` maps each device to the minimum value of `e` amongst its neighbors.
- *Domain restriction:* (`if` $\mathtt{e}_0$ $\mathtt{e}_1$ $\mathtt{e}_2$) is a branching construct, computing $\mathtt{e}_1$ in the restricted domain where $\mathtt{e}_0$ is true, and $\mathtt{e}_2$ in its complement.

To better illustrate the various constructs of field calculus, consider the following code, which estimates distance to devices where `source` is `true`, while avoiding devices where `obstacle` is `true`:

```
(def distance-avoiding-obstacle (source obstacle)
  (if obstacle infinity
    (rep d infinity (mux source 0
       (min-hood+ (+ (nbr-range) (nbr d)))))))
```

coloring field calculus keywords red, built-in functions green, and user-defined functions blue. In the region outside the obstacle (with the partition conducted by `if`), a distance estimate `d` (established by `rep`) is computed using built-in "multiplexing" selector `mux` to set sources to 0 and other devices to an updated distance estimate computed using the triangle inequality, taking the minimum value obtained by adding the distance to each neighbor to its estimate of `d` (obtained by `nbr`). In particular, `min-hood+` takes the minimum of all neighbors' values (excluding the device itself), and `mux` multiplexes between its second and third inputs, returning the second if the first is true and the third otherwise.

3.2 Local Semantics and Properties

Any field calculus program can be equivalently interpreted either as an aggregate-level computation on fields or as an equivalent "compiled" version implemented as a set of single-device operations and message passing. The full semantics may be found in [19,20], but the key ideas are simple enough to sketch briefly here.

Each field calculus program `P` consists of a set of user-defined function definitions and a single main expression $\mathtt{e}_0$. Given a network of interconnected devices D that runs a program `P`, "device δ fires" means that device $\delta \in D$ evaluates $\mathtt{e}_0$. The output of a device computation is a *value-tree*: an ordered tree of values tracking the result of computing each sub-expression encountered during evaluation of $\mathtt{e}_0$. Each expression evaluation on device δ is performed against the most recently received value-trees of its neighbors, and the produced value-tree is conversely made available to δ's neighbors (e.g., via broadcast in compressed form) for their next firing: (`nbr e`) uses the most recent value of `e` at the same position

in its neighbors' value-trees, (`rep x w e`) uses the value of `x` from the previous round, and (`if` e_0 e_1 e_2) completely erases the non-taken branch in the value-tree (allowing interactions through construct `nbr` with only neighbors that took the same branch, called "aligned neighbors"). A complete formal description of this semantics is presented in [19,20].

A type system based on the Hindley-Milner type system [18] can then be built for this calculus [19], which has two kinds of types: local types (for local values) and field types (for field values), associating to each local value a type `L`, and type `field(L)` to a neighboring field of elements of type `L`, and correspondingly a type `T` to any expression. This type system can then be used to detect semantic errors in a program (e.g., first expression of a call not evaluating to a function, incorrect argument types for a call, first argument of `if` not a Boolean), ensuring that these localized versions of field calculus programs are guaranteed to observe correct domain alignment and to terminate locally if the aggregate form terminates, i.e., to faithfully implement the desired equivalence relation.

The syntax and semantics of field calculus thus form a provably sound foundation for aggregate programming, ensuring that distributed services expressed in field calculus can be safely and predictably composed and modulated. At the same time, the small set of constructs also aids in portability, infrastructure independence, and interaction with non-aggregate services: the field calculus abstraction can be supported on any device or infrastructure where these simple constructs can be implemented, including heterogeneous mixtures of devices with different sensor, actuator, computation, and communication capabilities, so long as the devices have some means of interacting. Likewise, non-aggregate software services, such as other local applications or cloud services, are often complementary to aggregate services and can be connected with aggregate services simply by importing their APIs into the aggregate programming environment [37].

4 From Theory to Pragmatic System Engineering

Field calculus may be universal, but it is also too low level to be readily used for building complex distributed services. First, like other core calculi, in order to be compact enough to be readily manipulated for mathematical results, field calculus is extremely terse and generalized, as well as lacking any of the "syntactic sugar" features that make a language more usable for programming. Second, because it is universal, field calculus can express any program, including many that have none of the safety or resilience properties that we desire in distributed systems.

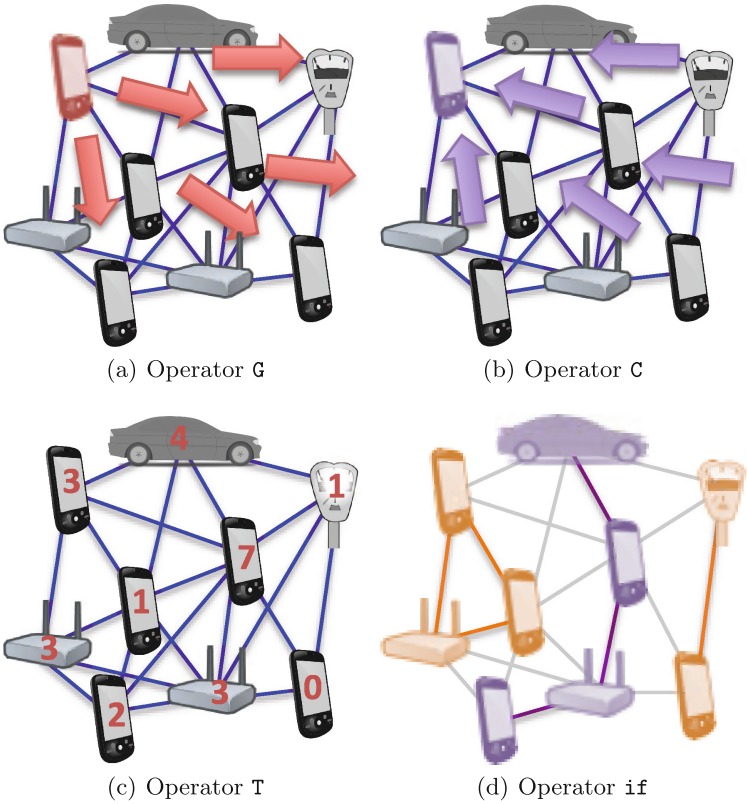
To make aggregate programming practically usable as an approach, we must further raise the level of abstraction. This is done first by implementing field calculus into a full programming language, Protelis [37], which makes it more usable via syntactic sugar and methods for interfacing with other existing libraries and frameworks. Protelis contains a complete implementation of the field calculus, hosted in Java via the Xtext language generator [22], with syntax transformed to an equivalent Java-like syntax with a number of useful syntactic sugar features such as variable definition, and taking advantage of Java's reflection mechanisms

to make it easy to import a large variety of useful libraries and APIs for use in Protelis. All further code will thus be given in Protelis, rather than field calculus.

The level of abstraction is then raised by identifying a composable system resilient "building block" operators, which provide core functions of coordination as well as resilience and safety guarantees. Finally, these building blocks are composed into both general and domain-specific APIs, which may further exploit optimized equivalents of particular operators for improved performance in more restricted use cases.

4.1 Building Blocks for Resilient Coordination

We first begin to raise the level of abstraction from field calculus toward an effective programming environment for resilient distributed systems by identifying a system of highly general and guaranteed composable "building block" operators for the construction of resilient coordination applications. This new



(a) Operator `G` (b) Operator `C`

(c) Operator `T` (d) Operator `if`

Fig. 5. Illustration of four "building block" operators for construction of resilient distributed services: information-spreading (`G`), information aggregation (`C`), aggregation over time (`T`), and partition into non-interacting subspaces (`if`).

layer (the middle layer in Fig. 3) is formed by careful selection of coordination mechanisms that are all *(i)* self-stabilizing, meaning that they can reactively adjust to changes in input values or the structure of the network, *(ii)* scalable to potentially very large networks, and *(iii)* preserve these resilience properties when the building blocks are composed together to form more complex coordination services. Critically, this means that it can be proven that any service built using only these "building blocks" will implicitly inherit such resilience [45].

One such set of operators has been identified already [8,45]: a set of three highly generalized coordination operators, `G`, `C` and `T`, along with field calculus' if and built-ins (Fig. 5). Each of these building blocks captures a family of frequently used strategies for achieving flexible and resilient decentralized behavior, hiding the complexity of using the low-level constructs of field calculus. The three building blocks are defined as:

- `G(source,init,metric,accumulate)` is a "spreading" operation generalizing distance measurement, broadcast, and projection, which takes four fields as inputs: `source` (a Boolean indicator field), `init` (initial values for the output field), `metric` (a function providing a map from each neighbor to a distance), and `accumulate` (a commutative and associative two-input function over values). It may be thought of as executing two tasks: first, computing a field of shortest-path distances from the `source` region according to the supplied function `metric`, and second, propagating values up the gradient of the distance field away from source, beginning with value `initial` and accumulating along the gradient with `accumulate`. For instance, if `metric` is physical distance, `initial` is 0, and `accumulate` is addition, then G creates a field mapping each device to its shortest distance to a source.
- `C(potential,accumulate,local,null)` is an operation that is complementary to G: it accumulates information down the gradient of a supplied `potential` field. This operator takes four fields as inputs: `potential` (a numerical field), `accumulate` (a commutative and associative two-input function over values), `local` (values to be accumulated), and `null` (an idempotent value for `accumulate`). At each device, the idempotent `null` is combined with the `local` value and any values from neighbors with higher values of the `potential` field, using function `accumulate` to produce a cumulative value at each device. For instance, if `potential` is exactly a distance gradient computing with G in a given region R, `accumulate` is addition, and `null` is 0, then C collects the sum of values of `local` in region R.
- `T(initial,floor,decay)` deals with time, whereas G and C deal with space. Since time is one-dimensional, however, there is no distinction between spreading and collecting, and thus only a single operator. This operator takes three fields as inputs: `initial` (initial values for the resulting field), `floor` (corresponding final values), and `decay` (a one-input strictly decreasing function over values). Starting with `initial` at each node, that value gets decreased by function `decay` until eventually reaching the `floor` value, thus implementing a flexible count-down, where the rate of the count-down may change over time. For instance, if `initial` is a pair of a value v and a

timeout t, `floor` is a pair of the blank value `null` and 0, and `decay` takes a pair, removing the elapsed time since previous computation from second component of the pair and turning the first component to `null` if the first reached 0, then T implements a limited-time memory of v.

```
def G(source, initial, metric, accumulate) {
  rep(dv <- [Infinity, initial]) {
    mux(source) {
      [0, initial]
    } else {
      minHood([nbr(dv.get(0)) + metric.apply(),
               accumulate.apply(nbr(dv.get(1)))])
    }
  }.get(1)
}
```

Fig. 6. Protelis implementation of operator G

Although there are only a few operators identified in [45], they are so general as to cover, individually or in combination, a large number of the common coordination patterns used in design of resilient systems. More importantly, when appropriately implemented in field calculus (e.g., Fig. 6), it has been shown that this system of operators, plus if and built-in operators, are elements of a "self-stabilizing language" where every program is a guaranteed to be self-stabilizing [45]. This means that distributed systems built from these operators enjoy a number of resilience properties: **stabilization:** if the input fields eventually reach a fixed state, the same happens for the output field; **resilience:** if some messages get lost during system evolution, or some node temporarily fails, this will not affect the final result; and **adaptability:** if input fields or network topology changes, the output field automatically adapts and changes correspondingly. These operators and their compositions are all also scalable for operation on potentially very large networks. Furthermore, this system of resilient operators can be readily expanded, simply by proving that any additional operators are also members of the self-stabilizing language, thereby proving that such an additional operator has the same resilience properties and can be safely composed with all previously identified operators.

4.2 Pragmatic Distributed Systems Engineering APIs

Building block operators are for the most part still too abstract and generalized to meet the pragmatic needs of typical applications programmers. To better meet these needs, various applications and combinations of "building block" operators can be captured into libraries, thereby forming a pragmatic and user-friendly API while still retaining all of the same resilience properties. Such libraries,

both general and domain-specific, form the penultimate layer in Fig. 3, upon which application code (the highest layer) is actually written.

For example, a number of useful functions related to information diffusion and distributed action can be constructed from various configurations of operator `G` (along with built-ins). One such common computation is estimating distance from a set of source devices, which we have previously discussed as part of the field calculus example in Sect. 3. Implemented as an application of `G`, it may be expressed in Protelis as:

```
def distanceTo(source) {
  G(source, 0, () -> {nbrRange}, (v) -> {v + nbrRange})
}
```

Applying `G` in a different way implements another common coordination action, broadcasting a value across the network from a source:

```
def broadcast(source, value) {
  G(source, value, () -> { nbrRange }, (v) -> {v})
}
```

Other `G`-based operations include construction of a Voronoi partition and a "path forecast" that marks paths that cross an obstacle or region of interest.

Similarly, functions related to collective perception of information can be implemented using operator `C`, such as accumulating a summary of all the values of a variable in a region to a given sink device:

```
def summarize(sink, accumulate, local, null) {
  C(distanceTo(sink), accumulate, local, null)
}
```

or computing the variable's average or maximum value in that region. Likewise, state and memory operations may be implemented using operator `T`, such as holding a value until a specified timeout:

```
def limitedMemory(value, timeout) {
  T([timeout, value], [0, false],
    (t) -> {[t.get(0) - dt, t.get(1)]}).get(1)
}
```

These general API functions can then be combined together, just as in any other programming environment, to create higher level general libraries and more domain-specific libraries. For example, a common "higher-level" general operation is to share a summary throughout a region, which can be implemented by applying `broadcast` to the output of `summarize`. Likewise, in the domain of spatially-embedded services like the crowd-safety application discussed above, a useful building block is to organize an environment into dynamically defined "management regions," which can be implemented by combining state and partition functions.

A mixture of such libraries at various levels of specificity and abstraction thus forms the actual programming environment that a typical developer would use for engineering the distributed coordination aspects of a resilient distributed system, while implementing the purely local or cloud-based aspects of the service using more standard programming tools for those aspects of the system. Because the APIs are ultimately built on the foundations of resilient operators and the field calculus, however, it is guaranteed that any service developed in this manner also implicitly obtains the properties of resilience and safe composition from the lower layers of the abstraction hierarchy.

4.3 Improving Performance by Equivalent Substitutions

Finally, just as the performance of more conventional programs can be improved by changing the implementation of key libraries (e.g., changing a generic hash table implementation to one better balanced for an application's expected table size and access patterns), the performance of aggregate programs can be improved by substituting the generic building block operators by more specialized variants with better performance in particular contexts and patterns of use [45].

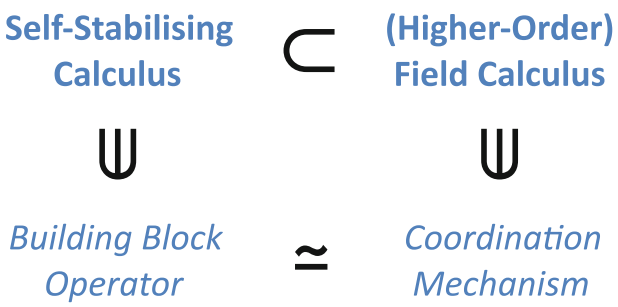


Fig. 7. Although field calculus can express any coordination mechanism, many useful mechanisms are difficult or impossible to express within a sublanguage that is known to be self-stabilizing. Any coordination mechanism that is asymptotically equivalent to a mechanism in the self-stabilizing subset, however, can be safely substituted without compromising safety or resilience guarantees. Figure adapted from [45].

Specifically, these substitutions make use of the mathematical relationship shown in Fig. 7: due to the functional composition model and modular proof used in establishing the self-stabilizing calculus, any coordination mechanism that can be guaranteed to self-stabilize to the same result as a building block operator can be substituted for that building block without affecting the self-stabilization of the overall program, including its final output. This allows creation of a "substitution library" of high-performance alternatives that can be used in certain circumstances and in those circumstances are more efficient or have more desirable dynamics. More formally:

Definition 1 (Substitutable Function). *Given functions* λ, λ' *with same type,* λ *is substitutable for* λ' *iff for any self-stabilizing list of expressions* $\overline{e}$*,* $(\lambda\ \overline{e})$ *always self-stabilizes to the same value as* $(\lambda'\ \overline{e})$*.*

The basic idea is that the property of self-stabilization specifies only the values after a function converges, so as long as two functions have the same converged values, they can be swapped without affecting any of the resilience properties based on self-stabilization. A building block operator with undesirable dynamical properties can thus be replaced by a more specialized coordination mechanism that improves overall performance without impairing resilience.

Three examples of substitution, given in [45], are:

- Distance estimation via G may converge extremely slowly when the network contains some devices that are close together [6]. Much faster alternatives exist, however, such as CRF-Gradient [6] and Flex-Gradient [4], and are known to self-stabilize to the same values as G distance estimation.
- Value collection with C is fragile: since it collects values over a spanning tree, even small perturbations can cause loss or duplication of values, with major transient impact on its results. When the accumulation is idempotent (e.g., logical AND) or separable (e.g., addition), this can be mitigated by accumulating across multiple paths.
- Low-pass filtering a signal is often useful for reducing noise. One common method, an exponential backoff filter, is substitutable with tracking a value via T, meaning that low-pass filters of this sort can be freely incorporated into programs without affecting their resilience.

When used in an application, such substitutions can markedly improve application performance. For example, consider an extremely simple distributed service for live estimation of crowd opinions of acts at a festival, implemented using

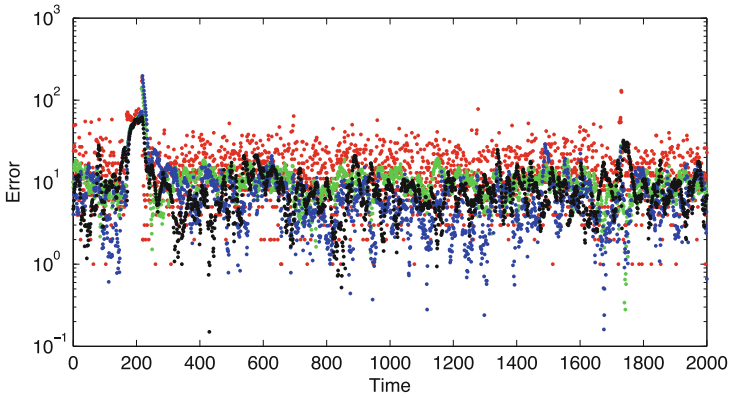


Fig. 8. Example crowd opinion feedback application is incrementally improved from its baseline performance (red) by first replacing T with an exponential filter (green), then C with multi-path summation (blue), and finally G with Flex-Gradient (black). Figure adapted from [45]. (Color figure online)

G to set up a potential field partitioning space into zones of influence for each act, C to sum a binary field of feedback, and T to track values:

```
(def add-range (v) (+ v (nbr-range)))

(def opinion-feedback (acts feedback)
  (T-filter
     (C (G acts 0 nbr-range add-range) sum feedback 0)))
```

In simulations of this scenario from [45], each incremental substitution of a generic function with a more optimized function improves the accuracy of the application: Fig. 8 shows how this application's performance can then be incrementally improved by first replacing T with an exponential filter, then C with multi-path summation, and finally G with Flex-Gradient.

Likewise, optimizations at lower layers of the framework have the potential improve the efficiency of field calculus implementations and the efficiency and simplicity of the implementation on particular devices and the interface with other applications and services. This layered approach to aggregate programming may thus serve as a framework for developing an efficient software ecosystem for engineering complex distributed systems, analogous to existing ecosystems for web or cloud development.

5 Application Examples

With the aid of appropriate domain-specific APIs, aggregate programming can greatly simplify the development and composition of distributed applications across a wide variety of domains. These can involve embedded devices and applications that are explicitly tied to space, but also can apply to more traditional location-agnostic computer networks. This section illustrates the breadth of possible applications by presenting examples across four domains: crowd safety at mass public events, UAV planning and control, construction of resilient enterprise systems, and network security.

5.1 Crowd Safety at Mass Events

One example, explored in [8], of an environment where aggregate programming is particularly applicable is at mass public events, such as marathons, outdoor concerts, festivals, and other civic activities. In these highly crowded environments, the combination of high densities of people and large spatial extent can often locally overwhelm the available infrastructure, causing cell phones to drop calls, data communications to become unreliable, etc. The physical environment is often overwhelmed as well, and the movement of high numbers of people in crowded and constrained environments can pose challenging emergent safety issues: in critically overcrowded environments, even the smallest incident can create a panic or stampede in which many people are injured or killed [43].

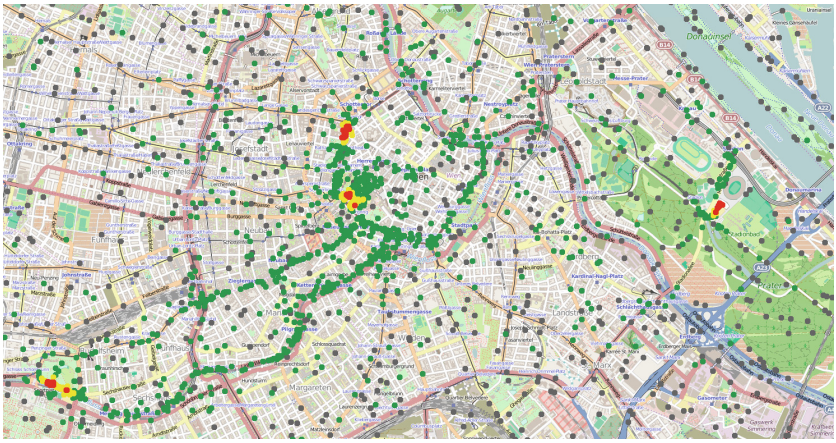


Fig. 9. A crowd safety service, restricted to run on personal devices (colored) in a simulation of approximately 2500 personal and embedded devices at the 2013 Vienna marathon, detects regions of potentially dangerous crowd density (red) and disseminates warnings to nearby devices (yellow). Figure adapted from [8]. (Color figure online)

Between smart-phones and other personal devices, however, the effective density of deployed infrastructure is much higher, since more people means more personal devices. Aggregate programming can be used to coordinate these devices, without the need for centrally deployed infrastructure, to provide services such as for crowd safety, to help identify and diffuse potentially dangerous situations. For example, crowding levels can be conservatively estimated by first estimating the local density of people as $\rho = \frac{|nbrs|}{p \cdot \pi r^2 \cdot w}$, where $|nbrs|$ counts neighbors within range r, p estimates the proportion of people with a participating device running the app and w estimates fraction of walkable space in the local urban environment, then comparing this estimate with "level of service" (LoS) ratings [24], taking LoS D (>1.08 people/m^2) to indicate a crowd and LoS E (>2.17 people/m^2) in a large group (e.g., $300+$ people) to indicate potentially dangerous density. Potential crowding danger can thus be detected and warnings disseminated robustly with just a few lines of Protelis code dynamically deployed and executed on individual devices [20,37]:

```
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 }
}

def crowdTracking(p, r, t) {
  let crowdRgn = recentTrue(densityEst(p, r)>1.08, t);
  if(crowdRgn) { dangerousDensity(p, r) } else { none };
}

def crowdWarning(p, r, warn, t) {
  distanceTo(crowdTracking(p,r,t) == high) < warn
}
```

Figure 9 shows an ALCHEMIST [38] simulation of such a crowd safety service running in an environment of pervasively deployed devices: 1479 mobile personal devices, each following a smart-phone position trace collected at the 2013 Vienna marathon, as discussed in [3,36], and 1000 stationary devices, all communicating via once per second asynchronous local broadcasts with 100 meters range, with all devices participating in infrastructure services but the crowd safety service restricted to run only on the mobile personal devices. Building this program via aggregate programming ensures that it is resilience and adaptive despite its very short length, allowing it to effectively estimate crowding and distribute warnings while executing on a large number of highly mobile devices.

5.2 Humanitarian Assistance and Disaster Relief Operations

Humanitarian assistance and disaster relief operations are another example of an environment where distributed coordination is particularly valuable, due to existing infrastructure being damaged or overwhelmed. With appropriate mechanisms for distributed coordination, however, "tactical cloud" resources can substitute for fixed infrastructure in support of assistance and relief operations. For example, [40,44] present an architecture of "edge nodes" equivalent to a 1/2-rack of servers in sturdy self-contained travel cases, which can be effectively mounted and operated even in small vehicles such as HMMVWs or towed trailers. Continuing advances in computing capability mean that such edge nodes actually offer a startling amount of capability: 10 such units can be equivalent to an entire cargo-container portable data center. The challenge is how to effectively coordinate and operate mission critical services across such devices, particularly given that the communications network between nodes has limited capacity and changes frequently as nodes are moved around and also given that individual edge nodes may be taken offline at any time due to evolving mission requirements, failures, accidents, or hostile action. Aggregate programming can simplify the development of resilient services for the tactical cloud environment, whereas existing methods tend to push application development toward a "star" topology where edge nodes interact mostly indirectly by means of their communications with a larger infrastructure cloud.

Consider, for example, a representative service example of assisting in the search for missing persons following a major disaster such as tsunami. This is a good example of a distributable mission application, since it involves data at several different scales: missing person queries (e.g., providing a photo of a missing loved one) and responses (e.g., a brief fragment of a video log showing a missing person) are fairly lightweight and can be spread between servers fairly easily, while video logs (e.g., from helmet- and vehicle-mounted cameras) are quite large and are best to search locally.

An implementation of this coordination service requires less than 30 lines of Protelis [37] code: this implementation distributes missing person queries, has them satisfied by video logged by other teams, then forwards that information back toward the team where the query originated. Figure 10 shows a screenshot

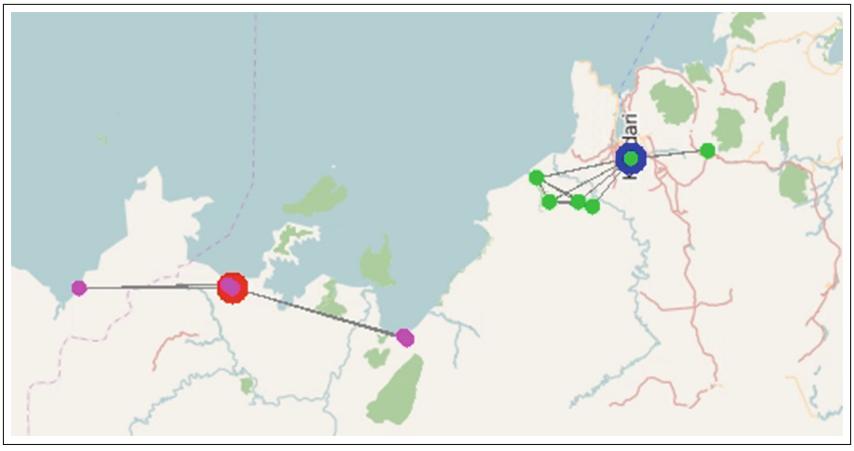


Fig. 10. Simulation of tactical cloud coordination in a humanitarian response scenario: tactical cloud nodes in survey team vehicles collectively help families find missing persons following a natural disaster: a query lodged with one team is opportunistically disseminated from its cloud node (red), to be compared against the video logs stored locally in each team's node. The desired information is located at a distant node (blue), then opportunistically forwarded to other nodes (green) until it can reach either the original source or some other node where the response can be received, thereby satisfying the query. (Color figure online)

from simulation of this scenario in the ALCHEMIST simulator [38]. In this scenario, a group of eleven survey teams are deployed amphibiously, then move around through the affected area, carrying out their survey mission over the course of several days. Each team hosts a half-rack server as part of their equipment, coordinating across tactical networks to collectively form a distributed cloud host for mission applications, such as searching video logs for missing persons and collating survey data. The distributed service implementation opportunistically disseminates queries, such that they end up moving implicitly by a combination of forwarding and taking advantage of vehicle motions to ferry data across gaps when there is no available connectivity. At each tactical cloud node, the query is executed against its video logs, and any matches are forwarded by the same opportunistic dissemination and marked off as resolved once the results of the service have been delivered to the person who requested assistance.

5.3 Resilient Enterprise Systems

Aggregate programming can also be applied to networks that are not closely tied to space, such as enterprise service networks, as in the work on distributed recovery of enterprise services presented in [15]. Management of small- to medium-scale enterprise systems is a pressing current problem (Fig. 11), since these systems are often quite complex, yet typically managed much more primitively than either individual machines (which are simpler and more uniform) or

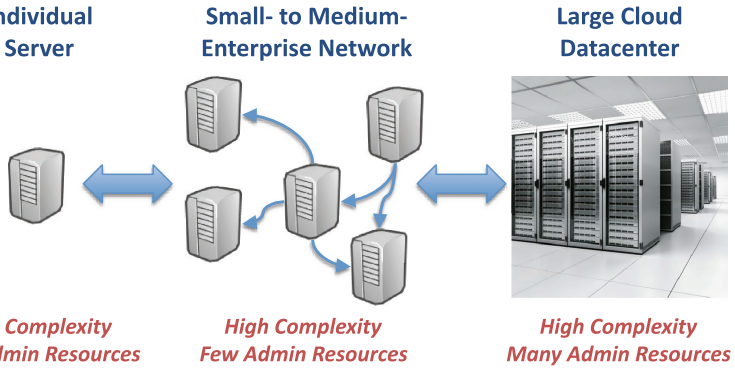


Fig. 11. Small- to medium-sized enterprises often have complex networks with many services and servers, but are not large enough to have significant administrative resources to devote to customization or to benefit from economies of scale. Figure adapted from [15].

large-scale datacenters (which can invest in high-scale or custom solutions). As a result, small and medium enterprises tend to have poor resilience and to suffer much more disruptive and extensive outages than large enterprises [1].

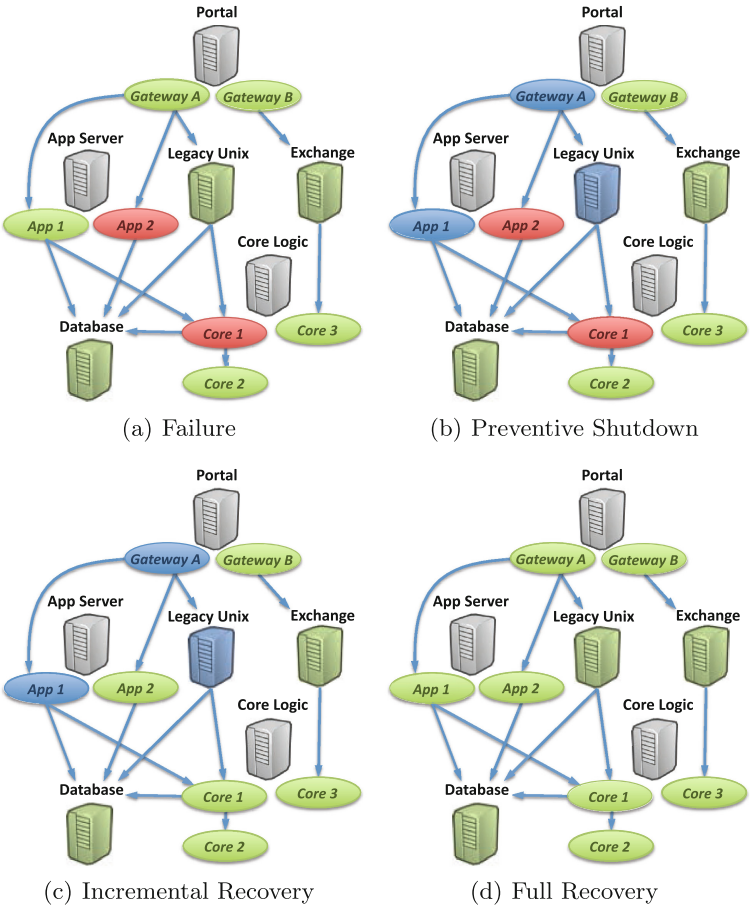
In [15], aggregate programming is used to address the common problem of safely and rapidly recovering from failures in a network of interdependent services, for which typical industry practice is to shut the entire system down and then restart services one at a time in a "known safe" order. The solution presented in [15], Dependency-Directed Recovery (DDR), uses Protelis [37] to implement a lightweight network of daemon processes that monitor service state, detecting dependencies (e.g., via methods such as in [13,29,39,42]) and controlling services to proactively bring down only those services with failed dependencies, then restart them in near-optimal time (Fig. 12). This system is realized with management daemons implemented Java, each hosting a Protelis VM executing the following simple coordination code:

```
// Collect state of monitored service from service manager daemon
let status = self.getEnvironmentVariable("serviceStatus");
let serviceID = self.getEnvironmentVariable("serviceID");
let depends = self.getEnvironmentVariable("dependencies");
let serviceDown = status=="hung" || status=="stop";

// Compute whether service can safely be run (i.e. dependencies are satisfied)
let liveSet = if(serviceDown) { [] } else { [serviceID] };
let nbrsLive = unionHood(nbr(liveSet));
let liveDependencies = nbrsLive.intersection(depends);
let safeToRun = liveDependencies.equals(depends);

// Act based on service state and whether it is safe to run
if(!safeToRun) {
```

```
  if(!serviceDown) {
    self.stopService() // Take service down to avoid misbehavior
  } else { false } // Wait for dependencies to recover before restarting
} else {
  if(serviceDown) {
    self.restartService() // Safe to restart
  } else { false } // Everything fine; no action needed
}
```



(a) Failure (b) Preventive Shutdown

(c) Incremental Recovery (d) Full Recovery

Fig. 12. Example of dependency-directed recovery in a service network, showing status *run* as green, *stop* as blue, and *hung* as red. Following failure of some set of services (a), other services that depend on them shut themselves down (b). As failed services restart, services that depend on them restart incrementally (c), until the entire service network has recovered (f). Figure adapted from [15].

With this program, any failure leads to a sequence of shutdowns, following dependency chains from failed services to the services that depend on them. Complementarily, when a service's dependencies start running again, that service restarts, becoming part of a wave of restarts propagating in parallel up the partial order established by dependencies.

Analysis of this system shows that it should produce distributed recovery in near-optimal time, slowed only be communication delays and the update period of the daemons. Experimental validation in emulated service networks of up to 20 services verifies this analysis, as well as showing a dramatic reduction in downtime compared to fixed-order restart, and allowing many services to continue running uninterrupted even while recovery is proceeding.

5.4 Network Security

For a final example, consider the value of effective and resilient coordination in network security. Improvements in virtualization technology have made it possible to trace and record the state evolution of an entire service or server, which can allow checkpointing of key points in process history, so that if attacks or faults are later detected the process can be "rewound" to a known-safe state and re-run with a dynamic patch or with the bad interaction edited out of the flow [16,35]. Executing such mechanisms, however, requires that interactions be able to be tightly monitored and ordered, which is often quite difficult and costly for networked services.

Taking an aggregate programming perspective, however, we may recognize that when interactions between services can be monitored, as in many networked services, a partial order of events based on the sending and receiving of messages can be substituted for the total order otherwise required for checkpointing or rewind and replay. To enable this, each service in the network takes local checkpoints every time that it sends a message or processes a message that it has received. A send/receive pair between two interacting services may then be interpreted as a directed graph edge, from send to receive, and the union of these directed edges with directed edges expressing the local order of local checkpoints on each server forms an distributed acyclic directed graph that can be safely interpreted as a partial order over events. A distributed checkpoint can then be computed emergently using distributed graph inference to compute the closure of graph succession on a set of local events (e.g., a set of faults or attacks), rewind executed by coordinated deletion of this subgraph, and replay executed by re-executing the incoming edges to the subgraph. Critically, this does not require any sort of synchronization between services, as well as allowing recovery to take place asynchronously, with any service not affected by possible contamination able to run uninterrupted and other services being able to run again as soon as they themselves are free of possible contamination.

Using aggregate programming to implement this partial order approach, coordination for rewind and replay can be implemented in less than 100 lines of Protelis [37]. Figure 13 shows an example screenshot from a rewind and replay system running on a network of emulated services, in the process of editing out

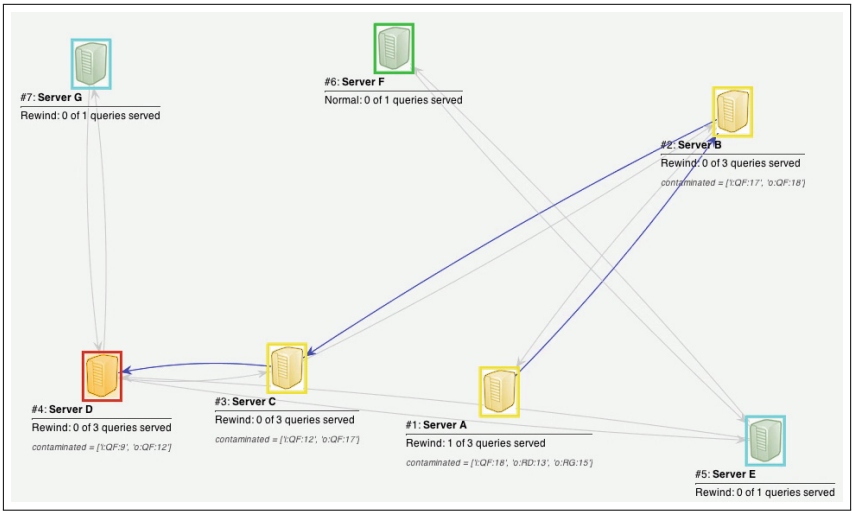


Fig. 13. Screenshot of distributed rewind and replay isolating and eliminating contamination (yellow machines) from an attack on a service network: following detection of an injected attack on a service (red box), potentially contaminated services (yellow box) suspend, trace potential contamination, and begin rewinding potentially contaminated interactions. Meanwhile, adjacent unaffected services (blue box) temporarily suspend operations to prevent spread of contamination while non-adjacent services (green box) continue to operate normally. (Color figure online)

an injected attack. Following detection of an injected attack on a service (e.g., via [16,35]), potentially contaminated services suspend, trace potential contamination, and begin rewinding potentially contaminated interactions. Meanwhile, adjacent unaffected services temporarily suspend operations as a "firebreak" against further spread of contamination, while non-adjacent services continue to operate normally.

6 Summary and Future Directions

This review has presented a summary of the aggregate programming approach to distributed systems engineering, including a review of its theoretical foundations in field calculus, how resilience can be guaranteed through composable "building blocks," and how these can be combined and refined to make effective APIs for engineering distributed applications across a wide range of domains. Overall, the aggregate programming approach offers the potential for complex distributed services to be specified succinctly and treated as coherent objects that can be safely encapsulated, modulated, and composed together, toward the ultimate goal of making distributed systems engineering as routine as ordinary single-device programming.

From this present state, four key directions for future work are:

- Further development of the theoretical foundations of aggregate programming, particularly with regards to mobile devices and the relationship between continuous environments and discrete networks of devices.
- Expansion of resilience results, including expansion of the set of building blocks and extension to a broader range of resilience properties, particularly regarding dynamical properties and feedback systems.
- Pragmatic improvements to the infrastructure and integration of aggregate programming, including expansion of libraries and APIs to more capabilities and more domains, integration with other pragmatic concerns such as security, optimizing usage of energy and other resources, and development of "operating system" layers for aggregate and hybrid aggregate/cloud architectures, as well as improvements to Protelis or other aggregate programming implementations.
- Developing applications of aggregate programming for a variety of problem domains, and transition of these applications into useful real-world deployments.

Our world is increasingly a world of computational aggregates, and methods such as these are the only way that we are likely to be able to keep engineering tractable, safe, and resilient in the increasingly complex interweaving of the informational and physical worlds, and our increasing dependence upon such distributed systems in the infrastructure of our civilization.

Acknowledgment. This work has been partially supported by the EU FP7 project "SAPERE - Self-aware Pervasive Service Ecosystems" under contract No. 256873 (Viroli), by the Italian PRIN 2010/2011 project "CINA: Compositionality, Interaction, Negotiation, Autonomicity" (Viroli), and by the United States Air Force and the Defense Advanced Research Projects Agency under Contract No. FA8750-10-C-0242 (Beal). 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 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. Approved for public release; distribution is unlimited.

References

1. Aberdeen Group: Why mid-sized enterprises should consider using disaster recovery-as-a-service, April 2012. http://www.aberdeen.com/Aberdeen-Library/7873/AI-disaster-recovery-downtime.aspx, Retrieved 13 July 2015
2. Anderson, D.P.: Boinc: a system for public-resource computing and storage. In: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing, pp. 4–10. IEEE (2004)

3. Anzengruber, B., Pianini, D., Nieminen, J., Ferscha, A.: Predicting social density in mass events to prevent crowd disasters. In: Jatowt, A., Lim, E.-P., Ding, Y., Miura, A., Tezuka, T., Dias, G., Tanaka, K., Flanagin, A., Dai, B.T. (eds.) SocInfo 2013. LNCS, vol. 8238, pp. 206–215. Springer, Heidelberg (2013). http://dx.doi.org/10.1007/978-3-319-03260-3_18
4. Beal, J.: Flexible self-healing gradients. In: ACM Symposium on Applied Computing, pp. 1197–1201. ACM, New York, March 2009
5. Beal, J., Bachrach, J.: Infrastructure for engineered emergence in sensor/actuator networks. IEEE Intell. Syst. **21**, 10–19 (2006)
6. Beal, J., Bachrach, J., Vickery, D., Tobenkin, M.: Fast self-healing gradients. In: Proceedings of ACM SAC 2008, pp. 1969–1975. ACM (2008)
7. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: languages for spatial computing. In: Mernik, M. (ed.) Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, Chap. 16, pp. 436–501. IGI Global (2013). A longer version available at: http://arxiv.org/abs/1202.5509
8. Beal, J., Pianini, D., Viroli, M.: Aggregate programming for the internet of things. IEEE Comput. **48**(9), 22–30 (2015). http://jakebeal.com/Publications/Computer-AggregateProgramming-2015.pdf
9. Beal, J., Viroli, M.: Building blocks for aggregate programming of self-organising applications. In: Eighth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASOW 2014, London, United Kingdom, 8–12 September, 2014, pp. 8–13 (2014). http://dx.doi.org/10.1109/SASOW.2014.6
10. Beal, J., Viroli, M.: Formal foundations of sensor network applications. SIGSPATIAL Spec. **7**(2), 36–42 (2015)
11. Beal, J., Viroli, M.: Space-time programming. Philos. Trans. R. Soc. Part A **73**, 20140220 (2015)
12. Beal, J., Viroli, M., Damiani, F.: Towards a unified model of spatial computing. In: 7th Spatial Computing Workshop (SCW 2014), AAMAS 2014, Paris, France, May 2014
13. Chen, X., Zhang, M., Mao, Z.M., Bahl, P.: Automating network application dependency discovery: experiences, limitations, and new solutions. In: OSDI, vol. 8, pp. 117–130 (2008)
14. Church, A.: A set of postulates for the foundation of logic. Ann. Math. **33**(2), 346–366 (1932)
15. Clark, S.S., Beal, J., Pal, P.: Distributed recovery for enterprise services. In: 2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), pp. 111–120, September 2015
16. Clark, S.S., Paulos, A., Benyo, B., Pal, P., Schantz, R.: Empirical evaluation of the a3 environment: evaluating defenses against zero-day attacks. In: 2015 10th International Conference on Availability, Reliability and Security (ARES), pp. 80–89. IEEE (2015)
17. Coore, D.: Botanical Computing: A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer. Ph.D. thesis, MIT (1999)
18. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Symposium on Principles of Programming Languages, POPL 1982, pp. 207–212. ACM (1982). http://doi.acm.org/10.1145/582153.582176
19. Damiani, F., Viroli, M., Beal, J.: A type-sound calculus of computational fields. Sci. Comput. Program. **117**, 17–44 (2016)

20. Damiani, F., Viroli, M., Pianini, D., Beal, J.: Code mobility meets self-organisation: a higher-order calculus of computational fields. In: Graf, S., Viswanathan, M. (eds.) FORTE 2015. LNCS, vol. 9039, pp. 113–128. Springer, Heidelberg (2015). http://dx.doi.org/10.1007/978-3-319-19195-9_8
21. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
22. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: OOPSLA, pp. 307–309. ACM (2010)
23. Finin, T., Fritzson, R., McKay, D., McEntire, R.: Kqml as an agent communication language. In: Proceedings of the Third International Conference on Information and Knowledge Management, CIKM 1994, pp. 456–463. ACM, New York (1994). http://doi.acm.org/10.1145/191246.191322
24. Fruin, J.: Pedestrian and Planning Design. Metropolitan Association of Urban Designers and Environmental Planners (1971)
25. Gentzsch, W.: Sun grid engine: towards creating a compute power grid. In: Proceedings of the First IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 35–36. IEEE (2001)
26. Giavitto, J.L., Godin, C., Michel, O., Prusinkiewicz, P.: Computational models for integrative and developmental biology. Technical report 72–2002, Univerite d'Evry, LaMI (2002)
27. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. **23**(3), 396–450 (2001)
28. Inchiosa, M., Parker, M.: Overcoming design and development challenges in agent-based modeling using ascape. Proc. Nat. Acad. Sci. U.S.A. **99**(Suppl. 3), 7304 (2002)
29. Lou, J.G., Fu, Q., Wang, Y., Li, J.: Mining dependency in distributed systems through unstructured logs analysis. ACM SIGOPS Operating Syst. Rev. **44**(1), 91–96 (2010)
30. Madden, S.R., Szewczyk, R., Franklin, M.J., Culler, D.: Supporting aggregate queries over ad-hoc wireless sensor networks. In: Workshop on Mobile Computing and Systems Applications (2002)
31. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: the tota approach. ACM Trans. Softw. Eng. Methodologies **18**(4), 1–56 (2009)
32. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 2.2, September 2009
33. Nagpal, R.: Programmable Self-Assembly: Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics. Ph.D. thesis, MIT (2001)
34. Newton, R., Welsh, M.: Region streams: functional macroprogramming for sensor networks. In: First International Workshop on Data Management for Sensor Networks (DMSN), pp. 78–87, August 2004
35. Paulos, A., Pal, P., Schantz, R., Benyo, B., Johnson, D., Hibler, M., Eide, E.: Isolation of malicious external inputs in a security focused adaptive execution environment. In: 2013 Eighth International Conference on Availability, Reliability and Security (ARES), pp. 82–91. IEEE (2013)
36. Pianini, D., Viroli, M., Zambonelli, F., Ferscha, A.: HPC from a self-organisation perspective: the case of crowd steering at the urban scale. In: 2014 International Conference on High Performance Computing Simulation (HPCS), pp. 460–467, July 2014

37. Pianini, D., Beal, J., Viroli, M.: Practical aggregate programming with protelis. In: ACM Symposium on Applied Computing (SAC 2015) (2015)
38. Pianini, D., Montagna, S., Viroli, M.: Chemical-oriented simulation of computational systems with Alchemist. J. Simul. **7**, 202–215 (2013). http://www.palgrave-journals.com/jos/journal/vaop/full/jos201227a.html
39. Popa, L., Chun, B.G., Stoica, I., Chandrashekar, J., Taft, N.: Macroscope: end-point approach to networked application dependency discovery. In: Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies, pp. 229–240. ACM (2009)
40. Simanta, S., Lewis, G.A., Morris, E.J., Ha, K., Satyanarayanan, M.: Cloud computing at the tactical edge. Technical report CMU/SEI-2012-TN-015, Carnegie Mellon University (2012)
41. Sklar, E.: Netlogo, a multi-agent simulation environment. Artif. Life **13**(3), 303–311 (2007)
42. Łgorzata Steinder, M., Sethi, A.S.: A survey of fault localization techniques in computer networks. Sci. Comput. Program. **53**(2), 165–194 (2004)
43. Still, G.K.: Introduction to Crowd Science. CRC Press, Boca Raton (2014)
44. Suggs, C.: Technical framework for cloud computing at the tactical edge. Technical report, US Navy Program Executive Office Command, Control, Communications, Computers and Intelligence (PEO C4I) (2013)
45. Viroli, M., Beal, J., Damiani, F., Pianini, D.: Efficient engineering of complex self-organizing systems by self-stabilising fields. In: IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO), pp. 81–90. IEEE, September 2015
46. Viroli, M., Damiani, F., Beal, J.: A calculus of computational fields. In: Canal, C., Villari, M. (eds.) ESOCC 2013. CCIS, vol. 393, pp. 114–128. Springer, Heidelberg (2013)
47. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services. ACM Press (2004)