Efficient Engineering of Complex Self-Organising Systems by Self-Stabilising Fields

Mirko Viroli*, Jacob Beal†, Ferruccio Damiani‡, and Danilo Pianini*
*Università di Bologna, Italy; Email: {mirko.viroli,danilo.pianini}@unibo.it
†Raytheon BBN Technologies, USA; Email: jakebeal@bbn.com
‡Università di Torino, Italy Email: damiani@di.unito.it

***Abstract*—Self-organising systems are notoriously difficult to engineer, particularly due to the interactions between complex specifications and the simultaneous need for efficiency and for resilience to faults and changes in execution conditions. We address this problem with an engineering methodology that separates these three aspects, allowing each to be engineered independently. Beginning with field calculus, we identify the largest known sub-language of self-stabilising programs, guaranteed to eventually attain correct behavior despite any perturbation in state or topology. Construction of complex systems is then facilitated by identifying "building block" operators expressed in this language, into which many complex specifications can be readily factored, thereby attaining resilience but possibly with improvable efficiency. Efficient implementation may then be achieved by substituting high-performance coordination mechanisms that are asymptotically equivalent to particular applications of building block operators. We illustrate this workflow by construction and simulation of example applications for evacuation alerts and for live estimation of crowd feedback at mass events.**

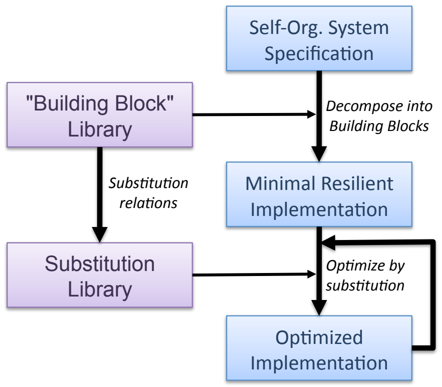


Fig. 1. Proposed workflow for self-organising systems engineering: an initial complex specification is implemented by decomposition into self-stabilising "building blocks," then optimised by substitution of equivalent high-performance coordination mechanisms.

I. Introduction

Self-organisation mechanisms support adaptivity and resilience in complex natural systems at all levels, from molecules and cells to animals, species, and entire ecosystems. A long-standing aim in computer science is to find effective engineering methods for exploiting such mechanisms to bring similar adaptivity and resilience to a wide variety of complex, large-scale applications—in smart mobility, crowd engineering, swarm robotics, etc. Practical adoption, however, poses serious challenges, as self-organisation mechanisms often trade efficiency for resilience and are often difficult to predictably compose to meet more complex specifications. Despite much prior work, e.g., in macroprogramming, spatial computing, pattern languages, etc. [1], to date no such approach has provided a comprehensive workflow for efficient engineering of complex self-organising systems.

Recent advances, however, have provided two key ingredients toward such an engineering workflow. First, *computational field calculus* [2], [3] provides a universal language for specifying self-organising distributed systems and, critically, a functional programming model for encapsulation and safely-scoped composition of self-organising systems. Second, "self-stabilising languages" have been identified [4], [5], where every program is a self-organising system with strong resilience and adaptivity guarantees.

This paper synthesises these two advances, plus a novel approach to optimisation of self-organising systems via substitution of equivalent coordination mechanisms, into a formal workflow for efficient engineering of complex self-organising systems (Figure 1). In particular, this paper explores this workflow in the context of spatially-embedded systems, though it is not limited to such systems and should apply well to most systems where device interactions form a sparse network. Section II introduces this workflow and positions it in the context of prior approaches. Following a review of field calculus in Section III, Section IV then formalises self-stabilisation for field programs and identifies the largest-to-date sub-language of such programs. Section V shows an example of how programming in this language can be simplified by means of "building block" operators for distributed action, collection, and temporal summary. Section VI then defines the class of substitutable functions and illustrates its use by identifying specialised alternatives for each identified building block operator. Finally, Section VII demonstrates our proposed workflow in action on two application scenarios, and Section VIII summarises and discusses future directions.

II. Engineering Workflow

Our proposed workflow (Figure 1) is based on a set of critical mathematical insights illustrated in Figure 2. The (higher-order) field-calculus [3] is a tiny universal language, in which any distributed computation can be

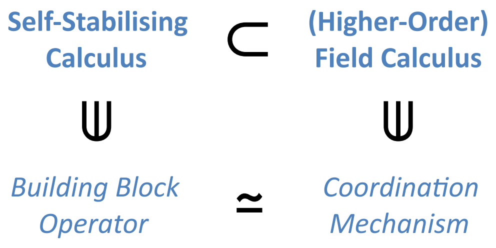


Fig. 2. Every coordination mechanism can be expressed in field calculus, but many may be difficult or impossible to express within its guaranteed self-stabilising subset. If one coordination mechanism is asymptotically equivalent to another mechanism in the self-stabilising subset, however, then it is guaranteed to be safely composable as well.

expressed, encapsulated, and safely composed (though it is best suited for use with sparse networks, i.e., those in which individual devices generally do not communicate with very large numbers of other devices). Because field calculus is universal, it can express both resilient and non-resilient computations. It can, however, be restricted to a sub-language in which all programs are guaranteed resilient in the form of self-stabilisation (discussed in detail in Section IV).

The succinctness that makies safety and self-stabilisation proofs tractable also makes such languages too low-level to be practical APIs for engineering complex self-organising systems. This can be mitigated by highly reusable "building block" operators capturing common coordination patterns [5], thus raising the abstraction level and allowing programmers to work with general-purpose functionalities or of user-friendly APIs capturing common use patterns.

These building blocks, despite their desirable resilience properties, may not be particularly efficient or have desirable dynamical properties. We thus incorporate a new insight, discussed in detail in Section VI: due to the functional composition model and modular proof used in establishing the self-stabilising calculus, any coordination mechanism that is guaranteed to self-stabilise to the same result as a building block can be substituted for that building block without affecting the self-stabilisation of the overall program, including its final output. This enables a "substitution library" of high-performance alternatives that are more efficient or have more desirable dynamics (typically specialised for particular applications of the building blocks, which are extremely generic).

Together, these insights enable a three-stage engineering workflow (Figure 1), which progressively treats complex specification, resilience, and efficiency.

1) The starting point for the workflow is a specification of the desired aggregate behavior to be implemented by the self-organising system.
2) The specification is then decomposed into coordination patterns (e.g., information spreading, information collection, state tracking) that can be mapped onto building block operators. The result is a "minimal resilient implementation" guaranteed to be self-stabilising but possibly far from optimal.
3) Each building block is then considered for replacement with a mechanism from the substitution library expected to provide better performance, confirming the improvement by analysis or simulation, then iterating, until no further improvement can be made.

A. *Relationship to Prior Work*

Many prior efforts have aimed at addressing the same problem of programming large-scale systems, and especially the challenge of mapping "macro-level" specifications for spatially situated computations into implementation in terms of individual devices and their local interactions. An in-depth survey of such work in [1] found four main classes of approach: *Device abstraction* approaches provide means to make interactions implicit, allowing an engineer to focus more directly on the macro/local problem but not providing any methodology for addressing that problem (e.g., NetLogo [6], Hood [7], TOTA [8], Gro [9], MPI [10], SAPERE [11]); *Pattern language* approaches allow definition and composition of module-like functionality in terms of geometric or topological constructions (e.g., Growing Point Language [12], Origami Shape Language [13]), while *Information movement* approaches focus on gathering information from certain regions of space-time, processing it, and delivering it to other regions (e.g., TinyDB [14] and Regiment [15]). Both pattern and information movement languages provide good workflows for self-organising systems engineering, but only within extremely restricted domains and without any means of extension to systems outside those domains. The work described in this paper build on the final class, *general-purpose aggregate languages*, which provide general mechanisms for programming aggregates, but have typically lacked sufficiently broad supporting libraries to enable efficient engineering of complex systems (e.g., field calculus [2], [3], Proto [16], Protelis [17], MGS [18]). Several recent works, however, have begun to take a more systematic approach [19], [4], [5], capturing common patterns from information movement and pattern languages in more generalisable models, now extended and synthesized in the workflow we present.

III. Computational Field Calculus

In this section we briefly and semi-formally review (higher order) field calculus—for complete details refer to [3]. The basic idea derives from an "aggregate" view of the network as a collective, in which the basic unit of data is a dynamically changing *field* of values held 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 takes field evolutions as input (e.g., from

$$\begin{array}{lcll}
\ell & ::= & \mathtt{c}\langle\overline{\ell}\rangle \;\big|\; \lambda & \text{local value}\\
\lambda & ::= & \mathtt{o} \;\big|\; \mathtt{f} \;\big|\; (\mathtt{fun}\ (\overline{\mathtt{x}})\ \mathtt{e}) & \text{function value}\\
\mathtt{e} & ::= & \ell \;\big|\; \mathtt{x} \;\big|\; (\mathtt{e}\ \overline{\mathtt{e}}) & \text{expression}\\
 & \big| & (\mathtt{rep}\ \mathtt{x}\ \mathtt{w}\ \mathtt{e}) & \\
 & \big| & (\mathtt{nbr}\ \mathtt{e}) & \\
 & \big| & (\mathtt{if}\ \mathtt{e}\ \mathtt{e}\ \mathtt{e}) & \\
\mathtt{w} & ::= & \mathtt{x} \;\big|\; \ell & \text{variable or local value}\\
\mathtt{F} & ::= & (\mathtt{def}\ \mathtt{f}(\overline{\mathtt{x}})\ \mathtt{e}) & \text{function declaration}\\
\mathtt{P} & ::= & \overline{\mathtt{F}}\ \mathtt{e} & \text{program}
\end{array}$$

Fig. 3. Syntax of (higher-order) field calculus.

sensors) and produces a field evolution as output, from which field values are snapshots. For example, given an input of a Boolean field mapping certain devices of interest to *true*, an output field of estimated distance to the nearest such device can be constructed by iterative aggregation and spreading of information, such that as the input changes the output changes to match. Note that while this model maps most intuitively onto spatially-embedded systems, since it is universal it can be used for any distributed computation (though it tends to be best suited for sparse networks, of which spatially-embedded systems are an example). The *(higher-order) field calculus* [3] succinctly captures the essence of field computations, much as λ-calculus [20] does functional computation. This simplicity then enables formal analysis of its properties and of the properties of self-organisation mechanisms expressed in field calculus.

A. Syntax of Field Calculus

Figure 3 presents field calculus syntax. Following [21], overbar notation denotes metavariables over sequences, e.g., $\overline{\mathtt{e}}$ is shorthand for the sequence of expressions $\mathtt{e}_1, \mathtt{e}_2, \ldots \mathtt{e}_n$ ($n \geq 0$). A local value ℓ represents the value of a field at a given device. It can be a *data value* $\mathtt{c}\langle\ell_1, \cdots, \ell_m\rangle$ (written $\mathtt{c}$ when $m = 0$), such as Booleans `true` and `false`, numbers, strings, or structured values like $\mathtt{Pair}\langle 3, \mathtt{Pair}\langle\mathtt{false}, 5\rangle\rangle$ or $\mathtt{Cons}\langle 2, \mathtt{Cons}\langle 4, \mathtt{Null}\rangle\rangle$, It can also be a *function value* λ: a built-in operator $\mathtt{o}$, user-defined function $\mathtt{f}$, or anonymous function $(\mathtt{fun}\ (\overline{\mathtt{x}})\ \mathtt{e})$ (in which we assume no free variables exist). Alternately, a device δ can hold a *neighbouring field value* ϕ, assigning a local value ℓ to each neighbour of δ: this is not reported in the syntax since it cannot be expressed in programs, only appearing dynamically during computations (see operator `nbr` below).

Expressions are the main entities of the calculus, modelling a whole field. An expression can be a local value ℓ, representing a constant field holding the value ℓ everywhere, a variable $\mathtt{x}$ used as function parameter or state variable (the set of free variables in an expression $\mathtt{e}$ is denoted by $\mathbf{FV}(\mathtt{e})$), or a composed expression built using the following constructs:

- *Built-in operator call:* $(\mathtt{e}\ \mathtt{e}_1 \cdots \mathtt{e}_n)$, where $\mathtt{e}$ evaluates to a ("point-wise") built-in operator $\mathtt{o}$, 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 $(\mathtt{o}\ \mathtt{e}_1 \cdots \mathtt{e}_n)$ produces a field mapping each device identifier δ to the result of applying $\mathtt{o}$ to the values at δ of its $n \geq 0$ arguments $\mathtt{e}_1, \ldots, \mathtt{e}_n$.
- *User-defined function call:* $(\mathtt{e}\ \mathtt{e}_1 \cdots \mathtt{e}_n)$, where $\mathtt{e}$ evaluates to a user-defined function $\mathtt{f}$, with corresponding declaration $(\mathtt{def}\ \ \mathtt{f}(\mathtt{x}_1 \ldots \mathtt{x}_n)\ \ \mathtt{e})$. Evaluating $(\mathtt{f}\ \ \mathtt{e}_1 \cdots \mathtt{e}_n)$ provides a standard (possibly recursive) call-by-value abstraction.
- *Anonymous function call:* $(\mathtt{e}\ \ \mathtt{e}_1 \cdots \mathtt{e}_n)$, has the same semantics as calling a user-defined function, except $\mathtt{e}$ evaluates to an anonymous function $(\mathtt{fun}\ (\mathtt{x}_1\ \cdots\ \mathtt{x}_n)\ \mathtt{e})$.
- *Time evolution:* `(rep x w e)` is a "repeat" construct for dynamically changing fields, assuming each device evaluates its main expression 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 each device has computed.
- *Neighbouring field construction:* `(nbr e)` models device-to-device interaction, by returning a field ϕ mapping each device δ to a neighbouring field value, which in turn maps each neighbour to its most recent available value of `e` (e.g., via periodic broadcast). Such neighbouring field values can then be manipulated and summarised with built-in operators, e.g., `(min-hood (nbr e))` maps each device to the minimum value of `e` amongst its neighbours.
- *Domain restriction:* $(\mathtt{if}\ \mathtt{e}_0\ \mathtt{e}_1\ \mathtt{e}_2)$ 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.

An example using the various constructs is:

```
(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. This code estimates distance to devices where `source` is `true`, avoiding devices where `obstacle` is `true`. In the region outside the obstacle (by `if`), a distance estimate `d` (established by `rep`) is computed using built-in selector `mux` to set sources to 0 and other devices by the triangle inequality, taking the minimum value obtained by adding the distance to each neighbor to its estimate of `d` (obtained by `nbr`).

B. Local Semantics and Properties

This aggregate-level model of computation over fields can be "compiled" into an equivalent system of local operations and message passing actually implementing the field calculus program on a distributed system [2], [3], as sketched in the following.

A field calculus program P is a set of user-defined function definitions and a 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 $\mathtt{e}_0$. Evaluation of an expression in device δ is performed against the most recently received value-trees of its neighbours, and the produced value-tree is conversely made available to δ's neighbours (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 neighbours' value-trees, (rep x w e) uses the value of x from the previous round, and (if $\mathtt{e}_0$ $\mathtt{e}_1$ $\mathtt{e}_2$) completely erases the non-taken branch in the value-tree (preventing interactions through construct nbr). A complete formal description of this semantics is presented in [3].

We shall assume a type system (a variant of the Hindley-Milner type system [22]) can be built for this calculus along the lines of [3], [4], which has two kinds of types: local types (for local values) and field types (for field values). This system associates to each local value a type L, and type field(L) to a neighbouring field of elements of type L, and correspondingly a type T to any expression. It can hence statically intercept 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), such that the following properties are henceforth considered to hold:

- *Type preservation:* if well-typed expression e has type T and e evaluates to v, then v has type T;
- *Domain alignment:* the domain of every field value arising in evaluating a well-typed expression on device δ consists of all aligned neighbours;
- *Termination:* any device firing is guarantee to terminate in any environmental condition[1].

IV. Self-Stabilising Calculus

In the dynamic environments typically considered by self-organising systems, a key resilience property is *self-stabilisation*, the ability of a system to recover from arbitrary changes in state. In particular, of the various notions of self-stabilisation (see survey in [23]), we use the definition from [24] as further restricted by [4]: a self-stabilising computation is one that, from any initial state, after some period without changes in the computational environment, reaches a single "correct" final configuration. As we will see, this definition covers a broad and useful class of self-organisation mechanisms (though some are excluded, such as continuously changing fields like self-synchronising pulse-coupled oscillators [25]). Self-stabilisation thus focuses on a computation's eventual behavior, rather than its transient behavior, which also enables optimisation by substitution of alternate coordination mechanisms.

[1]Termination of a round is clearly not decidable, but we shall assume—without loss of generality for the results of this paper—that a decidable subset of the termination fragment can be identified (e.g., by ruling out recursive user-defined functions or by applying standard static analysis techniques).

A. Self-stabilisation for fields

Assume a certain program P and some fixed environmental conditions $\mathcal{K}$ (i.e., the network topology and the inputs of sensors). Network state can be modelled by a field value N mapping each device $\delta \in D$ to the value-tree produced by its most recent firing, and change of these values by a transition relation $N \Rightarrow N'$, expressing the change of network state from N to N' as a sequence of devices fire. We write $N \Rightarrow_k N'$ $(k \geq 0)$ to mean a k-fair transition, i.e., that each device of the network fired at least k times, and for every h $(1 \leq h \leq k)$, its h-th firing is followed by at least $k - h + 1$ firings of all other devices, i.e., at least k complete rounds of firing occurred in the network.

We say that a network state N is *stable* if no device firing will change it, i.e., $N \Rightarrow N'$ implies $N = N'$, and network state N *self-stabilises* to stable state N_0 if through a sufficiently long fair sequence of transitions it necessarily reaches N_0 and remains there, i.e., for some $k \geq 0$, $N \Rightarrow_k N''$ implies $N'' = N_0$. Note that if a network state N self-stabilises, than it does so to a state that is unequivocally determined by the environmental conditions $\mathcal{K}$ (i.e., it does not depend on N), and can hence be interpreted as the output of computation.

Finally, we say that program P (or equivalently, its main expression) is *self-stabilising* if every state of every network running P is self-stabilising to some state under fixed environmental conditions. Note that this definition implies that field computations recover from any change on environmental conditions, since they react to them by forgetting their current state and reaching the stable state implied by such a change. Likewise, computation can reach a stable state only when environmental changes are transitory or do not affect N_0.

B. Preliminary definitions

Without risk of ambiguity, we abuse notation to use T both to name a type, as already mentioned, and also for its sets of values. We assume each local type L is associated with a total order relation $\leq_{\mathtt{L}}$ that is *locally noetherian* [4], namely, for every element $\ell \in \mathtt{L}$ there are no infinite ascending chains of elements smaller than ℓ—this typically holds for numeric data-types like Java's int, double, and BigInteger, and for any data-type expressed with a fixed number of bits. If type L has a maximal element, it is denoted $\top_{\mathtt{L}}$. We also assume each field type field(L) is associated with a partial order relation that is the element-wise extension of $\leq_{\mathtt{L}}$ to $\leq_{\mathtt{field(L)}}$ (i.e., $\phi \leq_{\mathtt{field(L)}} \phi'$ iff ϕ, ϕ' have the same domain and $\phi(\delta) \leq_{\mathtt{L}} (\delta)$).

$$\begin{array}{lcll}
\mathtt{s} & ::= & \ell \mid \mathtt{x} \mid (\mathtt{s}\ \overline{\mathtt{s}}) \mid (\mathtt{nbr}\ \mathtt{s}) \mid (\mathtt{if}\ \mathtt{s}\ \mathtt{s}\ \mathtt{s}) & \\
 & \mid & (\mathtt{rep}\ \mathtt{x}\ \mathtt{w}\ (\pi^{\mathsf{MB}}\ \mathtt{x}\ \overline{\mathtt{s}})) & \mathtt{x} \notin \mathbf{FV}(\overline{\mathtt{s}}) \\
 & \mid & (\mathtt{rep}\ \mathtt{x}\ \mathtt{w}\ (\pi^{\mathsf{F}}\ \mathtt{s}^{\mathsf{A}}\ (\mathtt{nbr}\ (\mathtt{s}\ \mathtt{x}))\ \overline{\mathtt{s}})) & \mathtt{x} \notin \mathbf{FV}(\mathtt{s}, \overline{\mathtt{s}}, \mathtt{s}^{\mathsf{A}}) \\
 & \mid & (\mathtt{rep}\ \mathtt{x}\ \mathtt{w}\ (\pi\ (\pi'\ (\mathtt{nbr}\ (\pi''\ \mathtt{x}\ \overline{\mathtt{s}}''))\ \overline{\mathtt{s}}')\ \overline{\mathtt{s}})) & \pi' \circ \pi = \pi^{\mathsf{MD}}, \pi'' \circ \pi' = \pi^{\mathsf{MBP}}, \mathtt{x} \notin \mathbf{FV}(\overline{\mathtt{s}}, \overline{\mathtt{s}}', \overline{\mathtt{s}}'')
\end{array}$$

Fig. 4. Self-stabilising calculus is a restricted sub-language of field calculus in which e is replaced by s above.

We say a function value is a *pure* function (denoted π) if its behaviour is point-wise, i.e., if it contains no `nbr`-expressions, `rep`-expressions, or built-in operators whose behaviour depends on environmental conditions, and all functions it calls are also pure. With abuse of notation, a pure function can be interpreted as a (mathematical) function over local values, writing e.g. $\pi(\overline{\mathtt{v}})$ to denote the result of applying arguments $\mathtt{v}_1, \ldots, \mathtt{v}_n$ to function π. A pure function may also be:

- *Monotonic non-decreasing* in its first argument (of type T), written π^{M}, meaning that:
$$\forall \mathtt{v}, \mathtt{v}', \overline{\mathtt{v}}:\ \mathtt{v} \leq_{\mathtt{T}} \mathtt{v}' \ \textbf{implies}\ \pi^{\mathsf{M}}(\mathtt{v}, \overline{\mathtt{v}}) \leq_{\mathtt{T}} \pi^{\mathsf{M}}(\mathtt{v}', \overline{\mathtt{v}})$$
- *Bounded* in its first argument (of type T), written π^{B}, meaning that:
$$\exists \mathtt{v}_0 : \forall \mathtt{v}, \overline{\mathtt{v}} : \quad \pi^{\mathsf{B}}(\mathtt{v}, \overline{\mathtt{v}}) \quad \leq_{\mathtt{T}} \quad \mathtt{v}_0$$
- *Double bounded* in its first argument (of type field(L)), written π^{D}, meaning that $\exists \ell : \forall \overline{\mathtt{v}}\, \forall \phi$:
$$\mathbf{min}(((\mathtt{min\text{-}hood+}\ \phi), \ell) \quad \leq_{\mathtt{L}} \quad \pi^{\mathsf{D}}(\phi, \overline{\mathtt{v}}) \quad \leq_{\mathtt{L}} \quad \ell$$
- *Progressive* in its first argument (of type L, equal to the return type), written π^{P}, meaning that
$$\forall \ell \neq \top_{\mathtt{L}} : \forall \overline{\ell} : \quad \ell <_{\mathtt{L}} \pi^{\mathsf{P}}(\ell, \overline{\mathtt{v}}).$$
This notation is used also for functions taking as first argument a local value of type L and returning a field of type field(L), meaning that for all local values ℓ and for all devices δ in the domain of $\phi = \pi^{\mathsf{P}}(\ell, \overline{\mathtt{v}})$ it holds that $\ell <_{\mathtt{L}} \phi(\delta)$.
- *Filtering* in its first argument (of type field(bool)), with second argument of type field(L)), written π^{F}, meaning that $\forall \phi_b, \phi, \overline{\mathtt{v}}$
$$\pi^{\mathsf{F}}(\phi_b, \phi, \overline{\mathtt{v}}) = \pi^{\mathsf{F}}(\phi_b|_{\phi_b^{-1}(\mathtt{true})}, \phi|_{\phi_b^{-1}(\mathtt{true})}, \overline{\mathtt{v}})$$
namely, π^{F} ignores the values of ϕ corresponding to devices where b holds `false`.

Notationally, we allow the symbol of a pure function to have multiple labels, e.g., π^{MBP} is a monotonic non-decreasing, bounded and progressive pure function.

C. Definition of Self-Stabilising Calculus

A self-stabilising sub-language of field calculus can be obtained by replacing e with *ss-expressions*, denoted s, following the syntax in Figure 4. We shall assume an ss-expression is well-typed if, in addition to the checks performed on a standard expression e, the body of any user-defined function or anonymous function it directly or indirectly calls is also an ss-expression. Symbol $\mathtt{s}^{\mathsf{A}}$ denotes an *acyclic topological relation*, namely, an ss-expression yielding a neighbouring field of Booleans to be interpreted as presence/absence of a logical directed connection with neighbours, such that these connections form a direct acyclic graph. This sub-language contains only self-stabilising programs:

Theorem 1 (Self-stabilising calculus) *Any well-typed ss-expression* s *is self-stabilising.*

Proof (sketch): The proof is by structural induction on the expression being evaluated (which, by construction does not contain free variables). It is straightforward to prove that all expressions not using the `rep`-construct self-stabilise in each device as soon as each argument self-stabilises and an additional firing occurs. Moreover, each of three `rep` patterns guarantees self-stabilisation. The case for the first pattern holds due to monotonicity and boundedness of the update function. The case for the second patten follows from acyclicity: at least one device will have no true values in $\mathtt{s}^{\mathsf{A}}$, hence it stabilises in one step; then, a device is always found that has true values in $\mathtt{s}^{\mathsf{A}}$ only relative to devices already self-stabilised, hence it eventually self-stabilises as well. The case for the third pattern adapts the technique used in the proof of Theorem 3 of [4], which proves self-stabilisation for a tiny calculus providing a spreading construct is a restricted form of this pattern. ■

The set of ss-expressions can be extended to include many more self-stabilising expressions by allowing any function λ (or π) be substituted by its body. Let E be an expression with one hole in it, and $E[\mathtt{e}]$ be the expression obtained from E by placing e in place of its hole. Let $\simeq_T$ be the smallest congruence (i.e. equivalence applicable at any level of depth) between expressions of type T that satisfies the following rules:

1) $E[\mathtt{e}] \simeq_{\mathtt{T}} ((\mathtt{fun}\ (\mathtt{x})\ E[\mathtt{x}])\ \mathtt{e})$ if $\mathtt{x} \notin \mathbf{FV}(\mathtt{e}, E)$.
2) $((\mathtt{fun}\ (\overline{\mathtt{x}})\ \mathtt{e})\ \overline{\mathtt{e}}) \simeq_T (\mathtt{f}\ \overline{\mathtt{e}})$ where $(\mathtt{def}\ \mathtt{f}\ (\overline{\mathtt{x}})\ \mathtt{e})$.

The following theorem states that self-stabilising expressions can be refactored according to congruence $\simeq_T$ without losing the self-stabilisation character, a property that will be used in next section to state self-stabilisation of building blocks.

Theorem 2 (Refactoring of self-stabilisation) *If* $\mathtt{e} \simeq_T \mathtt{e}'$*, then* e *is self-stabilising iff* $\mathtt{e}'$ *is self-stabilising.*

Proof (sketch): It can be shown: $(\mathtt{if}\ \mathtt{e}\ \mathtt{e}'\ \mathtt{e}'') \simeq_T (\mathtt{mux}\ \mathtt{e}\ \mathtt{e}'\ \mathtt{e}'')$: this is because `if` is just a `mux` on a virtual topology enacting restriction of the two regions where branches execute. Since `if` is the only construct with no call-by-value semantics, and it is $\simeq_T$-equivalent to a construct with call-by-value semantics, hence the calculus is $\simeq_T$-equivalent to a version with call-by-value semantics, in which the semantics of a function call (user-defined or anonymous) is exactly that of substituting its body after proper substitution of formal arguments with actual ones. ■

V. Examples of Building Block Operators

We now present an example of "building block" operators—in particular three highly reusable space and time operators from [5]—and show that they are members of the self-stabilising calculus based on the theorems provided in the previous section.

A. G: Spreading Information Across Space

We begin with operator G, which spreads information across space, potentially further organising and computing as it proceeds. This operator is a generalisation covering two of the most commonly used self-stabilising distributed algorithms—distance estimation (also often called "gradient") and broadcast—as well as a number of other applications, such as forecasting along paths. We define the G operator with the following field calculus expression:

```
(def G (source initial metric accumulate)
  (2nd // Return the reduction, discarding the computed distance
   (rep distance-value
    (tuple infinity initial) // Initial value
    (mux source
     (tuple 0 initial)// Source is distance zero, initial value
     (min-hood+ // Minimize lexicographically over non-self nbrs
      (tuple
       (+ (1st (nbr distance-value)) (metric))
       (accumulate (2nd (nbr distance-value)))))))))
```

where `min-hood+` takes the minimum of all neighbors' values (excluding the device itself), `mux` multiplexes between its second and third inputs, returning the second if the first is true and the third otherwise; and `tuple` creates a tuple of values, accessed by `1st` and `2nd`. The G operator may be thought of as executing two tasks, coupled together by the tuple `distance-value` in the `rep` expression. The first is computation of a field of shortest-path distances from a `source` region (indicated as a Boolean field) via the triangle inequality, where distance is computed by the supplied function `metric`. The second is accumulating values along the gradient of the distance field away from the source, using a function `accumulate` of one argument, the current accumulated value, beginning with initial value `initial`. The G operator can be configured to provide many different useful services: estimating distance, maximum-likelihood path probabilities, broadcasting values, forecasting obstacles, creating Voronoi partitions, etc. For instance, distance to source can be estimated as:

```
(def distance-to (source)
  (G source 0 nbr-range (fun (x) (+ x (nbr-range)))
```

Operator *G* can be proved self-stabilising when applied to a `metric` yielding positive values on neighbours. It follows the third `rep` pattern in Figure 4, where functions π, π', and π'' are defined as follows:

```
(def pi (x source initial)
  (mux source (tuple 0 initial) (min-hood+ x)))
(def pi' (x metric accumulate)
  (tuple (+ (1st x) (metric)) (accumulate (2nd x))))
(def pi'' (x) x)
```

B. C: Collecting Information From Across Space

The C operator is complementary to the G operator: whereas G spreads information away from the source, C accumulates information to the source. In order to maximize orthogonality with G, C assumes it is supplied with a potential field directing the accumulation of information. It may thus be defined:

```
(def C (potential accumulate local null)
 (rep v local
  (accumulate local
   (accumulate-hood accumulate
    (mux (= (nbr (find-parent potential)) (uid))
     (nbr v)
     null)))))
```

where `uid` returns a unique identifier for each device, `accumulate-hood` uses the function in its first argument to combine values from the field in its second argument, and the `find-parent` function is defined as:

```
(def find-parent (potential)
 (mux (< (1st (min-hood (nbr potential))) potential)
   (2nd (min-hood (nbr (tuple potential (uid)))))
   NaN))
```

Here `potential` is the potential field up which the values of `local` should be accumulated, combining values with `accumulate`, which must be a commutative and associative function of two arguments. In order to avoid multiply-counting devices (for those accumulations that are not idempotent), some neighbors are ignored, and their values replaced by a `null` that must not affect the accumulated value.

Combining with G (or G-derived functions), we can obtain a general "summary" operator that aggregates the values of a region to a `sink` and then spreads it throughout space, or similarly an "averaging" operator, or one that computes an integral or champions minimum or maximum value. For instance, the following function summarises the value of `local` into a `sink` device:

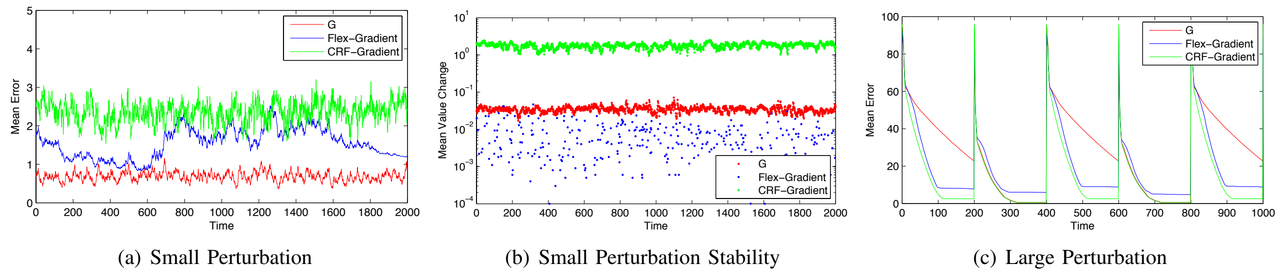
```
(def summarize (sink local)
  (C (distance-to sink) + local 0))
```

Operator *C* can be proved self-stabilising following the second `rep` pattern in Figure 4: `find-parent` by construction yields an acyclic topological relation, and the following function is a filtering one:

```
(def pi (x y accumulate null)
  (accumulate local (accumulate-hood accumulate
    (mux (= x (uid)) y null))))
```

C. T: Summarising Information Across Time

As C and G are for space, the T operator is for time. Since time is one-dimensional, however, there is no distinction between spreading and collecting, and thus there



(a) Small Perturbation (b) Small Perturbation Stability (c) Large Perturbation

Fig. 5. Examples comparing of convergence dynamics for distance estimation via G, CRF-Gradient, and Flex-Gradient: (a) with small perturbations, G produces the best estimates, but (b) Flex-Gradient has much more stable values. (c) With large perturbations, CRF-Gradient is both fast and accurate; Flex-Gradient is nearly as fast, but allows distortions to remain, and G is likely to converge slowly due to the rising value problem.

is only need for a single operator. We define the T operator as:

```
(def T (initial zero decay)
 (rep v initial (min initial (max zero (decay v)))))
```

where `decay` is a function strictly decreasing the value of its input. This operator may thus be understood as a flexible count-down toward zero, where the rate of the count-down may change over time. Example applications of T include timers, time-limited memories, and so on. Operator T is self-stabilising as it trivially adheres to the first `rep` pattern as of Figure 4, when `decay` is a decreasing function.

VI. Improving Dynamics by Substitution

The self-stabilising subset of field calculus identified in Section IV is valuable, in that it provides strong guarantees of composable resilience in distributed systems. With respect to pragmatic applications development, however, there are two critical shortcomings that need to be addressed: due to their generality, building blocks do not provide particularly good *dynamic* performance at any particular task; and many coordination mechanisms with better performance exist. Some of these are also known to self-stabilise, but may be difficult or impossible to express in the self-stabilising calculus.

We can short-circuit these issues with a notion of "substitutable functions" that extends the properties of self-stabilising calculus to a much broader class of coordination mechanisms. In particular, we consider two self-stabilising functions to be "substitutable" if the results that they eventually converge to are always identical, given the same inputs. More formally:

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

In essence, since self-stabilisation says nothing about what happens before the function converges, then as long as the converged values are the same, we can freely swap functions without affecting any properties based on self-stabilisation. A "correct" building block with undesirable dynamical properties can thus be replaced by a more specialised coordination mechanism that improves overall performance without impairing resilience.

Pragmatically, what this means is that we can extend the composability guarantees of the self-stabilising calculus to any coordination mechanism whose result is equivalent to some application of "building block" operators, per Figure 2. The remainder of this section illustrates this approach by presenting substitutable alternatives for G, C, and T, each of which is less general but offers superior performance in those cases where it can be applied.

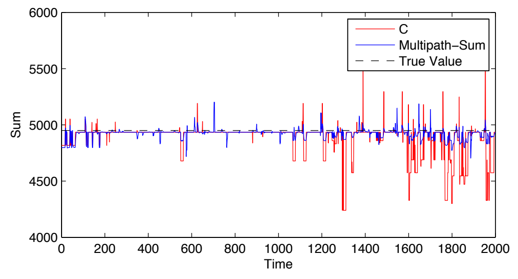
A. G vs. CRF-Gradient and Flex-Gradient

As noted above in Section V-A, one of the most common operations covered by the G operator is estimating distance to a source region. The proposed implementation of `distance-to`, however, produces rather sub-optimal convergence dynamics, and since distance estimation is such a common operation, it has been the subject of significant prior study. In particular, the G implementation of distance estimation is subject to the "rising value problem" studied in [26], which can cause extremely slow convergence when any devices are close to one another. Two good substitutable alternative algorithms are CRF-Gradient [26] and Flex-Gradient [27], both of which have already been proven to self-stabilise to the same values as G configured to estimate distance. Their field calculus code (not reported for the sake of space) is equivalent to the Proto code given in [26], [27].

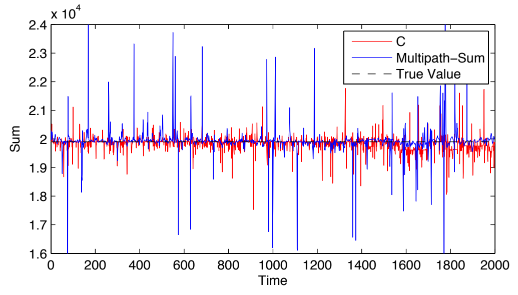
CRF-Gradient adjusts very quickly to changes in the network, at the cost of having a large minimum change size, so that even minuscule changes can make many devices' values rise before returning to their correct values. It is good for situations where it is important to get a correct value as quickly as possible. Flex-Gradient, on the other hand, prioritises smooth changes, at the cost of allowing small distortions in distance that are only slowly removed. It is good for situations where it is important for values to change smoothly and minor imprecision can be tolerated.

Figure 5 illustrates the differences of dynamics with two simulated examples. Both simulate all three algorithms in parallel[2] on 100 devices in a 200x20 meter space with

[2]All simulations of Flex-Gradient in this paper use parameters $\epsilon = 0.3$, $f = 10$, $\delta = 0.2$; see [27] for explanation.

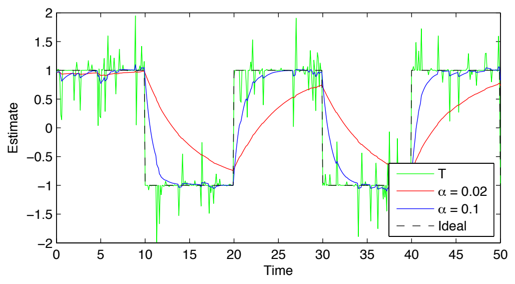


(a) Example of Small Perturbation

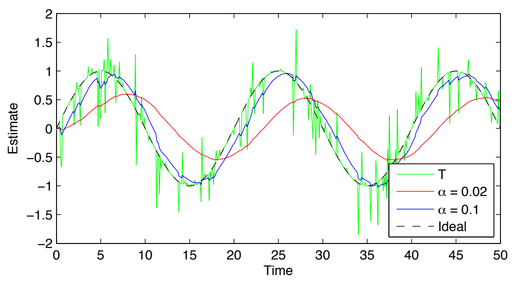


(b) Example of Large Perturbation

Fig. 6. A multi-path variant of summation for C improves dynamical stability for small perturbations, but can amplify problems with large perturbations.



(a) Example of Square Wave



(b) Example of Sine Wave

Fig. 7. When used to track a value, T is highly subject to noise; an exponential filter is a substitutable alternative that provides an adjustable trade-off between smoothness and time lag, as illustrated here with noisy square and sine waves.

20-meter unit disc communication and 1 second rounds. Error is computed as the difference between true distance to source and the current estimate. In the “small perturbation” simulation, a single source device performs a 2D random walk with steps drawn uniformly from range $[-0.1, 0.1]$ in each dimension. With constant small perturbations, G has the least error, but Flex-Gradient has much more stable values—in 83% of computational rounds not a single device changes its value. In the “large perturbation” simulation, every 200 seconds the source switches between two devices, one at $[-100, 0]$ the other at $[100, 0]$. With infrequent large perturbations, CRF-Gradient is both fast and accurate, while Flex-Gradient allows distortion to persist and G can be badly slowed by the rising value problem.

B. C vs. Multi-Path Summation/Combination

The value collection implemented by C is fragile, since C collects values over a spanning tree; even small perturbations can cause loss or duplication of values with major transient impact on its result. When the accumulation operation for C is either idempotent (e.g., logical and, or) or separable (e.g., summation), this can be mitigated by using all paths down the potential function rather than just one. For example, implemented for summation this is:

```
(def C-multisum (potential local)
  (rep v initial
    (+ initial
     (sum-hood (mux (>= potential (nbr potential)) 0
       (nbr (/ v
         (sum-hood (< (nbr potential) potential)))))))))
```

A similar pattern, substituting appropriate functions, can implement any other idempotent or separable function. Such functions are substitutable for C when the potential function has a single global minimum, and are expected to provide improved performance whenever there are likely to be many paths, at least some of which are stable.

Figure 6 illustrates the differences of dynamics with two simulated examples. Both algorithms are simulated in parallel devices in a 200x20 meter space with 20-meter unit disc communication and 1 second rounds. Potential is computed using Flex-Gradient to a single source, and the local value is a sequential identifier for each device, numbering 0 to $n-1$. The “small perturbation” simulation has 100 devices and the source device performs a 2D random walk with steps drawn uniformly from range $[-0.1, 0.1]$ in each dimension. In this case, multi-path improves performance significantly. The “large perturbation” simulation, on the other hand, shows how multi-path can actually degrade performance when there is too much volatility: here 200 devices wander between random waypoints at 0.01 meters/second, which produces enough volatility to sometimes disrupt multi-path summation worse than C.

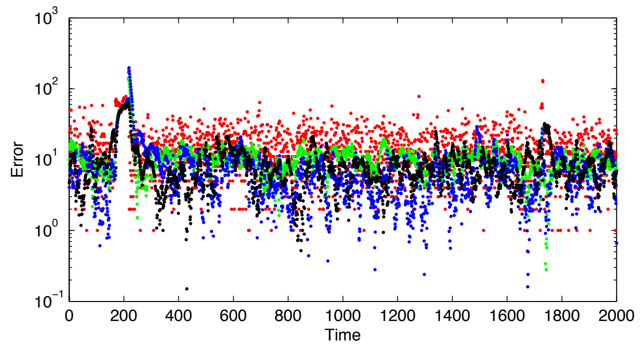
C. T vs. Exponential Filter

Finally, let us consider the common task of low-pass filtering a signal in order to smooth out noise. A common simple method for this is an exponential filter, which can be implemented:

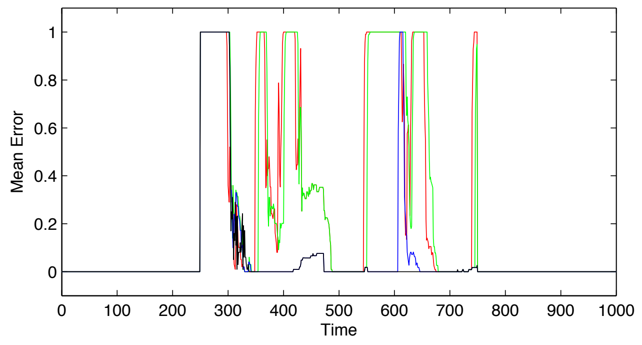
```
(def exponential-filter (x alpha)
  (rep v x (+ (* x alpha) (* v (- 1 alpha)))))
```

This is substitutable with T used to track a value:

```
(def T-filter (signal) (T signal signal 0))
```



(a) Crowd Opinion Feedback



(b) Evacuation Alert

Fig. 8. Example crowd opinion feedback (a) and evacuation alert (b) applications are incrementally improved from their baseline performance (red) by first replacing T with an exponential filter (green), then C with multi-path summation or logical "or" (blue), and finally G with Flex-Gradient (black).

Where this configuration of T passes all noise, however, an exponential filter trades off smoothness with response rate. Figure 7 illustrates this by comparing the response of T with that of an exponential filters with two different values of `alpha`. In particular, we simulate the response for unit square and sine wave of period 20 seconds with the addition of sparse noise $U[-1,1]^{11}$ sampled every 0.1 seconds. T passes all noise; $\alpha = 0.1$ tracks with a small delay but passes some noise, while $\alpha = 0.02$ smooths away virtually all noise at the cost of a significant delay.

VII. Application Examples

We now illustrate how substitutable functions can be applied to rapidly engineer and optimise resilient distributed applications, using two examples, live feedback on opinions from a crowd and evacuation alerts. The building blocks, plus field calculus' construct `if` and built-in operators, are sufficient to create both applications, hiding lower-level constructs `rep` and `nbr` "under the hood" as advocated in [5].

In the first example, the collective opinions of crowd members at a large festival are estimated by tracking how many are indicating (e.g., via an app on a smart device) a positive opinion of the act they are currently closest to. This application is first implemented using 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)))
```

Figure 8(a) 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. In particular, we show results from simulating all four variants in parallel on a network of 400 devices with 20-meter unit disc communication, in a 200x40 meter space, with stationary acts at $[-80, 0]$ and $[80, 0]$, and all other devices wandering between random waypoints at 0.3 meters/second. Devices have a fixed opinion of each act, 30% positive for the left act and 60% positive for the right act, and report their opinion of the act they are currently closest to. Error is computed as the sum of the absolute differences between true and estimated opinion count. In this case, substituting T produces a large improvement, C a somewhat smaller improvement, and G mostly a minor incremental improvement.

The second example is dissemination of an evacuation alert from a controlled zone, plus computation of recommended paths for evacuation, coordinated via a designated device. This is first implemented using T to track whether any device in the zone is currently alerted (using G to create a potential field to the coordinator and C to perform a logical "or"), then using G to broadcast that value from the coordinator throughout the zone and again to compute paths to the non-alerted areas outside of the zone:

```
(def evacuation-alert (zone coordinator alert)
  (G (not
      (if zone
        (G coordinator
          (T-filter
           (C (G coordinator 0 nbr-range add-range)
              or alert false)
          nbr-range identity))
        false))
   0 nbr-range add-range))
```

Figure 8(b) shows how this application's performance can then be incrementally improved by first replacing T with an exponential filter, then C with multi-path logical "or," and finally both distance-measure Gs (but not the broadcast G) with Flex-Gradient. In particular, we show results from simulating all four variants in parallel on a network of 200 devices with 15-meter unit disc communication, in a 200x20 meter space, with a stationary coordinator at $[-50, 0]$ and all other devices wandering between random waypoints at 0.1 meters/second. The "zone" is all devices in the left half of the space, and the emergency alert is perceived by any device within 3 meters of $[-25, 5]$ starting at 250 seconds and lasting until 750 seconds. Error is computed as the fraction of devices that are provided with evacuation routes that will not make rapid progress out of the zone, computed as those where the dot product of the ideal direction and the direction to the neighbor with lowest potential is less than 0.5. In this case, each substitution makes a significant improvement in the quality of the evacuation information provided.

VIII. Contributions

We have identified a large class of self-stabilising distributed algorithms, including a set of general "building block" operators that conceptually simplify the specification of programs within this class. Further, we have introduced a notion of substitutable algorithms that allow dynamical performance to be optimised by replacing building block operators with other coordination mechanisms that converge to the same values, illustrating this process via simulation and application examples.

An important future direction for improvement is to obtain a more systematic characterisation for the dynamic trade-space, in order to enable a more systematic approach to optimisation via mechanism substitution. In addition to making human engineering easier, this may also enable automated substitution optimisation, both during the engineering process and dynamically at run-time. Other important directions for improvement are expansion of the library of building blocks (including to non-spatial systems), identification of more substitution relationships between building blocks and high-performance resilient coordination mechanisms, and development and deployment of applications based on this approach.

Acknowledgments

This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 644298 HyVar (Damiani), EU FP7 project SAPERE under contract No 256873 (Viroli), ICT COST Actions IC1402 ARVI and IC1201 BETTY (Damiani), Italian PRIN 2010/2011 project CINA (Damiani & Viroli), Ateneo/CSP project SALT (Damiani), and 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] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll, "Organizing the aggregate: Languages for spatial computing," in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, M. Mernik, Ed. IGI Global, 2013, ch. 16, pp. 436–501.

[2] M. Viroli, F. Damiani, and J. Beal, "A calculus of computational fields," in *Advances in Service-Oriented and Cloud Computing*, ser. Communications in Computer and Info. Sci., C. Canal and M. Villari, Eds. Springer, 2013, vol. 393, pp. 114–128.

[3] F. Damiani, M. Viroli, D. Pianini, and J. Beal, "Code mobility meets self-organisation: A higher-order calculus of computational fields," in *Formal Techniques for Distributed Objects, Components, and Systems*, ser. LNCS, S. Graf and M. Viswanathan, Eds. Springer International, 2015, vol. 9039, pp. 113–128.

[4] M. Viroli and F. Damiani, "A calculus of self-stabilising computational fields," in *Coordination Languages and Models*, ser. LNCS, eva Kühn and R. Pugliese, Eds. Springer-Verlag, Jun. 2014, vol. 8459, pp. 163–178.

[5] J. Beal and M. Viroli, "Building blocks for aggregate programming of self-organising applications," in *Workshop on Fundamentals of Collective Adaptive Systems*, 2014, pp. 8–13.

[6] E. Sklar, "Netlogo, a multi-agent simulation environment," *Artificial life*, vol. 13, no. 3, pp. 303–311, 2007.

[7] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler, "Hood: a neighborhood abstraction for sensor networks," in *Int'l Conf. on Mobile Systems, Applications, and Services*. ACM Press, 2004.

[8] M. Mamei and F. Zambonelli, "Programming pervasive and mobile computing applications: The TOTA approach," *ACM Trans. Software Eng. Methodologies*, vol. 18, no. 4, pp. 1–56, 2009.

[9] The Klavins Lab, *Gro: The cell programming language*, University of Washington, 2012.

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

[11] F. Zambonelli *et al.*, "Developing pervasive multi-agent systems with nature-inspired coordination," *Pervasive and Mobile Computing*, vol. 17, Part B, pp. 236–252, 2015.

[12] D. Coore, "Botanical computing: A developmental approach to generating interconnect topologies on an amorphous computer," Ph.D. dissertation, MIT, 1999.

[13] R. Nagpal, "Programmable self-assembly: Constructing global shape using biologically-inspired local interactions and origami mathematics," Ph.D. dissertation, MIT, 2001.

[14] S. R. Madden, R. Szewczyk, M. J. Franklin, and D. Culler, "Supporting aggregate queries over ad-hoc wireless sensor networks," in *Workshop on Mobile Comp. and Sys. Apps.*, 2002.

[15] R. Newton and M. Welsh, "Region streams: Functional macroprogramming for sensor networks," in *1st Int'l Workshop on Data Management for Sensor Networks*, Aug. 2004, pp. 78–87.

[16] J. Beal and J. Bachrach, "Infrastructure for engineered emergence in sensor/actuator networks," *IEEE Intelligent Systems*, vol. 21, pp. 10–19, March/April 2006.

[17] D. Pianini, J. Beal, and M. Viroli, "Practical aggregate programming with PROTELIS," in *ACM Symposium on Applied Computing (SAC 2015)*, 2015.

[18] J.-L. Giavitto, O. Michel, J. Cohen, and A. Spicher, "Computation in space and space in computation," Univerite d'Evry, LaMI, Tech. Rep. 103-2004, 2004.

[19] J. L. Fernandez-Marquez, G. D. M. Serugendo, S. Montagna, M. Viroli, and J. L. Arcos, "Description and composition of bio-inspired design patterns: a complete overview," *Natural Computing*, vol. 12, no. 1, pp. 43–67, 2013.

[20] A. Church, "A set of postulates for the foundation of logic," *Annals of Mathematics*, vol. 33, no. 2, pp. 346–366, 1932.

[21] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: A minimal core calculus for Java and GJ," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 3, 2001.

[22] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Symposium on Principles of Programming Languages*, ser. POPL '82. ACM, 1982, pp. 207–212.

[23] M. Schneider, "Self-stabilization," *ACM Computing Surveys*, vol. 25, pp. 45–67, 1993.

[24] S. Dolev, *Self-Stabilization*. MIT Press, 2000.

[25] R. E. Mirollo and S. H. Strogatz, "Synchronization of pulse-coupled biological oscillators," *SIAM Journal on Applied Mathematics*, vol. 50, no. 6, pp. 1645–1662, 1990.

[26] J. Beal, J. Bachrach, D. Vickery, and M. Tobenkin, "Fast self-healing gradients," in *ACM Symposium on Applied Computing*. ACM, 2008, pp. 1969–1975.

[27] J. Beal, "Flexible self-healing gradients," in *ACM Symposium on Applied Computing*. ACM, March 2009, pp. 1197–1201.