

Code Mobility Meets Self-organisation: a Higher-order Calculus of Computational Fields ^{*}

Ferruccio Damiani¹, Mirko Viroli², Danilo Pianini², and Jacob Beal³

¹ University of Torino, Italy

`ferruccio.damiani@unito.it`

² University of Bologna, Italy

`{mirko.viroli,danilo.pianini}@unibo.it`

³ Raytheon BBN Technologies, USA

`jakebeal@bbn.com`

Abstract. Self-organisation mechanisms, in which simple local interactions result in robust collective behaviors, are a useful approach to managing the coordination of large-scale adaptive systems. Emerging pervasive application scenarios, however, pose an openness challenge for this approach, as they often require flexible and dynamic deployment of new code to the pertinent devices in the network, and safe and predictable integration of that new code into the existing system of distributed self-organisation mechanisms. We approach this problem of combining self-organisation and code mobility by extending “computational field calculus”, a universal calculus for specification of self-organising systems, with a semantics for distributed first-class functions. Practically, this allows self-organisation code to be naturally handled like any other data, e.g., dynamically constructed, compared, spread across devices, and executed in safely encapsulated distributed scopes. Programmers may thus be provided with the novel first-class abstraction of a “distributed function field”, a dynamically evolving map from a network of devices to a set of executing distributed processes.

1 Introduction

In many different ways, our environment is becoming ever more saturated with computing devices. Programming and managing such complex distributed systems is a difficult challenge and the subject of much ongoing investigation in contexts such as cyber-physical systems, pervasive computing, robotic systems, and large-scale wireless sensor

^{*} This work has been partially supported by HyVar (www.hyvar-project.eu, this project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 644298 - Damiani), by EU FP7 project SAPERE (www.sapere-project.eu, under contract No 256873 - Viroli), by ICT COST Action IC1402 ARVI (www.cost-arvi.eu - Damiani), by ICT COST Action IC1201 BETTY (www.behavioural-types.eu - Damiani), by the Italian PRIN 2010/2011 project CINA (sysma.imtlucca.it/cina - Damiani & Viroli), by Ateneo/CSP project SALT (salt.di.unito.it - Damiani), 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.

networks. A common theme in these investigations is *aggregate programming*, which aims to take advantage of the fact that the goal of many such systems are best described in terms of the aggregate operations and behaviours, e.g., “distribute the new version of the application to all subscribers”, or “gather profile information from everybody in the festival area”, or “switch on safety lights on fast and safe paths towards the emergency exit”. Aggregate programming languages provide mechanisms for building systems in terms of such aggregate-level operations and behaviours, and a global-to-local mapping that translates such specifications into an implementation in terms of the actions and interactions of individual devices. In this mapping, self-organisation techniques provide an effective source of building blocks for making such systems robust to device faults, network topology changes, and other contingencies. A wide range of such aggregate programming approaches have been proposed [3]: most of them share the same core idea of viewing the aggregate in terms of dynamically evolving *fields*, where a field is a function that maps each device in some domain to a computational value. Fields then become first-class elements of computation, used for tasks such as modelling input from sensors, output to actuators, program state, and the (evolving) results of computation.

Many emerging pervasive application scenarios, however, pose a challenge to these approaches due to their openness. In these scenarios, there is need to flexibly and dynamically deploy new or revised code to pertinent devices in the network, to adaptively shift which devices are running such code, and to safely and predictably integrate it into the existing system of distributed processes. Prior aggregate programming approaches, however, have either assumed that no such dynamic changes of code exist (e.g., [2, 21]), or else provide no safety guarantees ensuring that dynamically composed code will execute as designed (e.g., [15, 22]). Accordingly, our goal in this paper is develop a foundational model that supports both code mobility and the predictable composition of self-organisation mechanisms. Moreover, we aim to support this combination such that these same self-organisation mechanisms can also be applied to manage and direct the deployment of mobile code.

To address the problem in a general and tractable way, we start from the *field calculus* [21], a recently developed minimal and universal [5] computational model that provides a formal mathematical grounding for the many languages for aggregate programming. In field calculus, all values are fields, so a natural approach to code mobility is to support fields of first-class functions, just as with first-class functions in most modern programming languages and in common software design patterns such as MapReduce [10]. By this mechanism, functions (and hence, code) can be dynamically consumed as input, passed around by device-to-device communication, and operated upon just like any other type of program value. Formally, expressions of the field calculus are enriched with function names, anonymous functions, and application of function-valued expressions to arguments, and the operational semantics properly accommodates them with the same core field calculus mechanisms of neighbourhood filtering and alignment [21]. This produces a unified model supporting both code mobility and self-organisation, greatly improving over the independent and generally incompatible mechanisms which have typically been employed in previous aggregate programming approaches. Programmers are thus provided with a new first-class abstraction of

a “distributed function field”: a dynamically evolving map from the network to a set of executing distributed processes.

Section 2 introduces the concepts of higher-order field calculus; Section 3 formalises their semantics; Section 4 illustrates the approach with an example; and Section 5 concludes with a discussion of related and future work.

2 Fields and First-Class Functions

The defining property of fields is that they allow us to see computation from two different viewpoints. On the one hand, by the standard “local” viewpoint, computation is seen as occurring in a single device, and it hence manipulates data values (e.g., numbers) and communicates such data values with other devices to enable coordination. On the other hand, by the “aggregate” (or “global”) viewpoint [21], computation is seen as occurring on the overall network of interconnected devices: the data abstraction manipulated is hence a whole distributed *field*, a dynamically evolving data structure having extent over a subset of the network. This latter viewpoint is very useful when reasoning about aggregates of devices, and will be used throughout this document. Put more precisely, a field value ϕ may be viewed as a function $\phi : D \rightarrow \mathcal{L}$ that maps each device δ in the domain D to an associated data value ℓ in range \mathcal{L} . Field computations then take fields as input (e.g., from sensors) and produce new fields as outputs, whose values may change over time (e.g., as inputs change or the computation progresses). For example, the input of a computation might be a field of temperatures, as perceived by sensors at each device in the network, and its output might be a Boolean field that maps to `true` where temperature is greater than 25°C, and to `false` elsewhere.

Field Calculus The *field calculus* [21] is a tiny functional calculus capturing the essential elements of field computations, much as λ -calculus [7] captures the essence of functional computation and FJ [12] the essence of object-oriented programming. The primitive expressions of field calculus are data values denoted ℓ (Boolean, numbers, and pairs), representing constant fields holding the value ℓ everywhere, and variables x , which are either function parameters or state variables (see the `rep` construct below). These are composed into programs using a Lisp-like syntax with five constructs:

(1) *Built-in function call* (`o e1 ... en`): A built-in operator `o` is a means to uniformly model a variety of “point-wise” operations, i.e. involving neither state nor communication. Examples include simple mathematical functions (e.g., addition, comparison, sine) and context-dependent operators whose result depends on the environment (e.g., the 0-ary operator `uid` returns the unique numerical identifier δ of the device, and the 0-ary `nbr-range` operator yields a field where each device maps to a subfield mapping its neighbours to estimates of their current distance from the device). The expression (`o e1 ... en`) thus produces a field mapping each device identifier δ to the result of applying `o` to the values at δ of its $n \geq 0$ arguments e_1, \dots, e_n .

(2) *Function call* (`f e1 ... en`): Abstraction and recursion are supported by function definition: functions are declared as (`def f(x1 ... xn) e`) (where elements x_i are formal parameters and `e` is the body), and expressions of the form (`f e1 ... en`) are the way of calling function `f` passing n arguments.

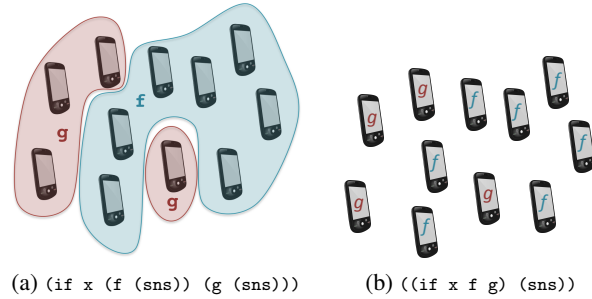


Fig. 1: Field calculus functions are evaluated over a domain of devices. E.g., in (a) the `if` operation partitions the network into two subdomains, evaluating `f` where field `x` is true and `g` where it is false (both applied to the output of sensor `sns`). With first-class functions, however, domains must be constructed dynamically based on the identity of the functions stored in the field, as in (b), which implements an equivalent computation.

(3) *Time evolution* (`rep x e0 e`): The “repeat” construct supports dynamically evolving fields, assuming that each device computes its program repeatedly in asynchronous rounds. It initialises state variable `x` to the result of initialisation expression `e0` (a value or a variable), then updates it at each step by computing `e` against the prior value of `x`. For instance, `(rep x 0 (+ x 1))` is the (evolving) field counting in each device how many rounds that device has computed.

(4) *Neighbourhood field construction* (`nbr e`): Device-to-device interaction is encapsulated in `nbr`, which returns a field ϕ mapping each neighbouring device to its most recent available value of `e` (i.e., the information available if devices broadcast the value of `e` to their neighbours upon computing it). Such “neighbouring” fields can then be manipulated and summarised with built-in operators, e.g., `(min-hood (nbr e))` outputs a field mapping each device to the minimum value of `e` amongst its neighbours.

(5) *Domain restriction* (`if e0 e1 e2`): Branching is implemented by this construct, which computes `e1` in the restricted domain where `e0` is true, and `e2` in the restricted domain where `e0` is false.

Any field calculus computation may thus be viewed as a function f taking zero or more input fields and returning one output field, i.e., having the signature $f : (D \rightarrow \mathcal{L})^k \rightarrow (D \rightarrow \mathcal{L})$. Figure 1a illustrates this concept, showing an example with complementary domains on which two functions are evaluated. This aggregate-level model of computation over fields can then be “compiled” into an equivalent system of local operations and message passing actually implementing the field calculus program on a distributed system [21].

Higher-order Field Calculus The *higher-order field calculus (HFC)* is an extension of the field calculus with embedded first-class functions, with the primary goal of allowing it to handle functions just like any other value, so that code can be dynamically injected, moved, and executed in network (sub)domains. If functions are “first class” in the language, then: (i) functions can take functions as arguments and return a function as result (higher-order functions); (ii) functions can be created “on the fly” (anonymous

$e ::= x \mid v \mid (e \bar{e}) \mid (\text{rep } x \ w \ e) \mid (\text{nbr } e) \mid (\text{if } e \ e \ e)$	expression
$v ::= \ell \mid \phi$	value
$\ell ::= b \mid n \mid \langle \ell, \ell \rangle \mid o \mid f \mid (\text{fun } (\bar{x}) \ e)$	local value
$w ::= x \mid \ell$	variable or local value
$F ::= (\text{def } f(\bar{x}) \ e)$	user-defined function
$P ::= \bar{F} \ e$	program

Fig. 2: Syntax of HFC (differences from field calculus are highlighted in grey).

functions); *iii*) functions can be moved between devices (via the `nbr` construct); and *iv*) the function one executes can change over time (via `rep` construct).

The syntax of the calculus is reported in Fig. 2. Values in the calculus include fields ϕ , which are produced at run-time and may not occur in source programs; also, local values may be smoothly extended by adding other ground values (e.g., characters) and structured values (e.g., lists). Borrowing syntax from [12], the overbar notation denotes metavariables over sequences and the empty sequence is denoted by \bullet . E.g., for expressions, we let \bar{e} range over sequences of expressions, written e_1, e_2, \dots, e_n ($n \geq 0$). The differences from the field calculus are as follows: function application expressions ($e \bar{e}$) can take an arbitrary expression e instead of just an operator o or a user-defined function name f ; anonymous functions can be defined (by syntax $(\text{fun } (\bar{x}) \ e)$); and built-in operators, user-defined function names, and anonymous functions are values. This implies that the range of a field can be a function as well. To apply the functions mapped to by such a field, we have to be able to transform the field back into a single aggregate-level function. Figure 1b illustrates this issue, with a simple example of a function call expression applied to a function-valued field with two different values.

How can we evaluate a function call with such a heterogeneous field of functions? It would seem excessive to run a separate copy of function f for every device that has f as its value in the field. At the opposite extreme, running f over the whole domain is problematic for implementation, because it would require devices that may not have a copy of f to help in evaluating f . Instead, we will take a more elegant approach, in which making a function call acts as a branch, with each function in the range applied only on the subspace of devices that hold that function. Formally, this may be expressed as transforming a function-valued field ϕ into a function f_ϕ that is defined as:

$$f_\phi(\psi_1, \psi_2, \dots) = \bigcup_{f \in \phi(D)} f(\psi_1|_{\phi^{-1}(f)}, \psi_2|_{\phi^{-1}(f)}, \dots) \quad (1)$$

where ψ_i are the input fields, $\phi(D)$ is set of all functions held as data values by some devices in the domain D of ϕ , and $\psi_i|_{\phi^{-1}(f)}$ is the restriction of ψ_i to the subspace of only those devices that ϕ maps to function f . In fact, when the field of functions is constant, this reduces to be precisely equivalent to a standard function call. This means that we can view ordinary evaluation of function f as equivalent to creating a function-valued field with a constant value f , then making a function call applying that field to its argument fields. This elegant transformation is the key insight of this paper, enabling first-class functions to be implemented with a minimal change to the existing semantics while also ensuring compatibility with the prior semantics as well, thus also inheriting its previously established desirable properties.

3 The Higher-order Field Calculus: Dynamic and Static Semantics

Dynamic Semantics (Big-Step Operational Semantics) As for the field calculus [21], devices undergo computation in rounds. In each round, a device sleeps for some time, wakes up, gathers information about messages received from neighbours while sleeping, performs an evaluation of the program, and finally emits a message to all neighbours with information about the outcome of computation before going back to sleep. The scheduling of such rounds across the network is fair and non-synchronous. This section presents a formal semantics of device computation, which is aimed to represent a specification for any HFC-like programming language implementation.

The syntax of the HFC calculus has been introduced in Section 2 (Fig. 2). In the following, we let meta-variable δ range over the denumerable set \mathbf{D} of *device identifiers* (which are numbers). To simplify the notation, we shall assume a fixed program P . We say that “device δ fires”, to mean that the main expression of P is evaluated on δ .

We model device computation by a big-step operational semantics where the result of evaluation is a *value-tree* θ , which is an ordered tree of values, tracking the result of any evaluated subexpression. Intuitively, the evaluation of an expression at a given time in a device δ is performed against the recently-received value-trees of neighbours, namely, its outcome depends on those value-trees. The result is a new value-tree that is conversely made available to δ ’s neighbours (through a broadcast) for their firing; this includes δ itself, so as to support a form of state across computation rounds (note that any implementation might massively compress the value-tree, storing only enough information for expressions to be aligned). A *value-tree environment* Θ is a map from device identifiers to value-trees, collecting the outcome of the last evaluation on the neighbours. This is written $\bar{\delta} \mapsto \bar{\theta}$ as short for $\delta_1 \mapsto \theta_1, \dots, \delta_n \mapsto \theta_n$.

The syntax of field values, value-trees and value-tree environments is given in Fig. 3 (top). Figure 3 (middle) defines: the auxiliary functions ρ and π for extracting the root value and a subtree of a value-tree, respectively (further explanations about function π will be given later); the extension of functions ρ and π to value-tree environments; and the auxiliary functions *args* and *body* for extracting the formal parameters and the body of a (user-defined or anonymous) function, respectively. The computation that takes place on a single device is formalised by the big-step operational semantics rules given in Fig. 3 (bottom). The derived judgements are of the form $\delta; \Theta \vdash e \Downarrow \theta$, to be read “expression e evaluates to value-tree θ on device δ with respect to the value-tree environment Θ ”, where: (i) δ is the identifier of the current device; (ii) Θ is the field of the value-trees produced by the most recent evaluation of (an expression corresponding to) e on δ ’s neighbours; (iii) e is a run-time expression (i.e., an expression that may contain field values); (iv) the value-tree θ represents the values computed for all the expressions encountered during the evaluation of e —in particular $\rho(\theta)$ is the resulting value of expression e . The first firing of a device δ after activation or reset is performed with respect to the empty tree environment, while any other firing must consider the outcome of the most recent firing of δ (i.e., whenever Θ is not empty, it includes the value of the most recent evaluation of e on δ)—this is needed to support the stateful semantics of the `rep` construct.

The operational semantics rules are based on rather standard rules for functional languages, extended so as to be able to evaluate a subexpression e' of e with respect to

Field values, value-trees, and value-tree environments:	
$\phi ::= \bar{\delta} \mapsto \bar{\ell}$	field value
$\theta ::= v(\bar{\theta})$	value-tree
$\Theta ::= \bar{\delta} \mapsto \bar{\theta}$	value-tree environment
Auxiliary functions:	
$\rho(v(\bar{\theta})) = v$	
$\pi_i(v(\theta_1, \dots, \theta_n)) = \theta_i$ if $1 \leq i \leq n$	$\pi^{\ell, n}(v(\theta_1, \dots, \theta_{n+2})) = \theta_{n+2}$ if $\rho(\theta_{n+1}) = \ell$
$\pi_i(\theta) = \bullet$ otherwise	$\pi^{\ell, n}(\theta) = \bullet$ otherwise
For $aux \in \rho, \pi_i, \pi^{\ell, n}$: $\begin{cases} aux(\delta \mapsto \theta) = \delta \mapsto aux(\theta) & \text{if } aux(\theta) \neq \bullet \\ aux(\delta \mapsto \theta) = \bullet & \text{if } aux(\theta) = \bullet \\ aux(\Theta, \Theta') = aux(\Theta), aux(\Theta') \end{cases}$	
$args(f) = \bar{x}$ if $(\text{def } f(\bar{x}) e)$	$body(f) = e$ if $(\text{def } f(\bar{x}) e)$
$args((\text{fun } (\bar{x}) e)) = \bar{x}$	$body((\text{fun } (\bar{x}) e)) = e$
Rules for expression evaluation:	
$\frac{[\text{E-LOC}]}{\delta; \Theta \vdash \ell \Downarrow \ell()}$	$\frac{[\text{E-FLD}]}{\delta; \Theta \vdash \phi \Downarrow \phi'()} \quad \begin{array}{l} \phi' = \phi \upharpoonright_{\text{dom}(\Theta) \cup \{\delta\}} \\ \delta; \Theta \vdash e \Downarrow \theta \end{array}$
$\frac{[\text{E-B-APP}]}{\delta; \pi_1(\Theta) \vdash e_1 \Downarrow \theta_1 \quad \dots \quad \delta; \pi_n(\Theta) \vdash e_n \Downarrow \theta_n \quad v = \varepsilon_{\delta; \Theta}^{\circ}(\rho(\theta_1), \dots, \rho(\theta_n))}{\delta; \pi_{n+1}(\Theta) \vdash e_{n+1} \Downarrow \theta_{n+1} \quad \rho(\theta_{n+1}) = o}$	
$\frac{[\text{E-D-APP}]}{\delta; \pi_1(\Theta) \vdash e_1 \Downarrow \theta_1 \quad \dots \quad \delta; \pi_n(\Theta) \vdash e_n \Downarrow \theta_n \quad \rho(\theta_{n+1}) = \ell \quad args(\ell) = x_1, \dots, x_n \quad body(\ell) = e}{\delta; \pi^{\ell, n}(\Theta) \vdash e[x_1 := \rho(\theta_1) \quad \dots \quad x_n := \rho(\theta_n)] \Downarrow \theta_{n+2} \quad v = \rho(\theta_{n+2})}$	
$\frac{[\text{E-REP}]}{\delta; \Theta \vdash e_{n+1}(e_1, \dots, e_n) \Downarrow v(\theta_1, \dots, \theta_{n+2})}{\delta; \Theta \vdash e_{n+1} \Downarrow \theta_{n+1} \quad \rho(\theta_{n+1}) = \ell \quad \delta; \pi_1(\Theta) \vdash e[x := \ell_0] \Downarrow \theta_1 \quad \ell_1 = \rho(\theta_1) \quad \ell_0 = \begin{cases} \rho(\Theta(\delta)) & \text{if } \Theta \neq \emptyset \\ \ell & \text{otherwise} \end{cases}}$	
$\frac{[\text{E-NBR}]}{\delta; \Theta \vdash (\text{nbr } e) \Downarrow \phi(\theta_1)}{\delta; \Theta \vdash (\text{rep } x \ell e) \Downarrow \ell_1(\theta_1) \quad \Theta_1 = \pi_1(\Theta) \quad \delta; \Theta_1 \vdash e \Downarrow \theta_1 \quad \phi = \rho(\Theta_1)[\delta \mapsto \rho(\theta_1)]}$	
$\frac{[\text{E-THEN}]}{\delta; \Theta \vdash (\text{if } e e' e'') \Downarrow \ell(\theta_1, \theta_2)}{\delta; \pi_1(\Theta) \vdash e \Downarrow \theta_1 \quad \rho(\theta_1) = \text{true} \quad \delta; \pi^{\text{true}, 0} \Theta \vdash e' \Downarrow \theta_2 \quad \ell = \rho(\theta_2)}$	
$\frac{[\text{E-ELSE}]}{\delta; \Theta \vdash (\text{if } e e' e'') \Downarrow \ell(\theta_1, \theta_2)}{\delta; \pi_1(\Theta) \vdash e \Downarrow \theta_1 \quad \rho(\theta_1) = \text{false} \quad \delta; \pi^{\text{false}, 0} \Theta \vdash e'' \Downarrow \theta_2 \quad \ell = \rho(\theta_2)}$	

Fig. 3: Big-step operational semantics for expression evaluation

the value-tree environment Θ' obtained from Θ by extracting the corresponding subtree (when present) in the value-trees in the range of Θ . This process, called *alignment*, is modelled by the auxiliary function π , defined in Fig. 3 (middle). The function π has two different behaviours (specified by its subscript or superscript): $\pi_i(\theta)$ extracts the i -th subtree of θ , if it is present; and $\pi^{\ell, n}(\theta)$ extracts the $(n+2)$ -th subtree of θ , if it is present and the root of the $(n+1)$ -th subtree of θ is equal to the local value ℓ .

Rules [E-LOC] and [E-FLD] model the evaluation of expressions that are either a local value or a field value, respectively. For instance, evaluating the expression 1 produces (by rule [E-LOC]) the value-tree 1(), while evaluating the expression + produces the value-tree +(). Note that, in order to ensure that domain restriction is obeyed (cf.

Section 2), rule [E-FLD] restricts the domain of the value field ϕ to the domain of Θ augmented by δ .

Rule [E-B-APP] models the application of built-in functions. It is used to evaluate expressions of the form $(e_{n+1} e_1 \cdots e_n)$ such that the evaluation of e_{n+1} produces a value-tree θ_{n+1} whose root $\rho(\theta_{n+1})$ is a built-in function o . It produces the value-tree $v(\theta_1, \dots, \theta_n, \theta_{n+1})$, where $\theta_1, \dots, \theta_n$ are the value-trees produced by the evaluation of the actual parameters e_1, \dots, e_n ($n \geq 0$) and v is the value returned by the function. Rule [E-B-APP] exploits the special auxiliary function ε , whose actual definition is abstracted away. This is such that $\varepsilon_{\delta; \Theta}^o(\bar{v})$ computes the result of applying built-in function o to values \bar{v} in the current environment of the device δ . In particular, we assume that the built-in 0-ary function `uid` gets evaluated to the current device identifier (i.e., $\varepsilon_{\delta; \Theta}^{\text{uid}}() = \delta$), and that mathematical operators have their standard meaning, which is independent from δ and Θ (e.g., $\varepsilon_{\delta; \Theta}^+(1, 2) = 3$). The ε function also encapsulates measurement variables such as `nbr-range` and interactions with the external world via sensors and actuators. In order to ensure that domain restriction is obeyed, for each built-in function o we assume that: $\varepsilon_{\delta; \Theta}^o(v_1, \dots, v_n)$ is defined only if all the field values in v_1, \dots, v_n have domain $\mathbf{dom}(\Theta) \cup \{\delta\}$; and if $\varepsilon_{\delta; \Theta}^o(v_1, \dots, v_n)$ returns a field value ϕ , then $\mathbf{dom}(\phi) = \mathbf{dom}(\Theta) \cup \{\delta\}$. For instance, evaluating the expression $(+ 1 2)$ produces the value-tree $3(1(), 2(), +())$. The value of the whole expression, 3, has been computed by using rule [E-B-APP] to evaluate the application of the sum operator $+$ (the root of the third subtree of the value-tree) to the values 1 (the root of the first subtree of the value-tree) and 2 (the root of the second subtree of the value-tree). In the following, for sake of readability, we sometimes write the value v as short for the value-tree $v()$. Following this convention, the value-tree $3(1(), 2(), +())$ is shortened to $3(1, 2, +)$.

Rule [E-D-APP] models the application of user-defined or anonymous functions, i.e., it is used to evaluate expressions of the form $(e_{n+1} e_1 \cdots e_n)$ such that the evaluation of e_{n+1} produces a value-tree θ_{n+1} whose root $\ell = \rho(\theta_{n+1})$ is a user-defined function name or an anonymous function. It is similar to rule [E-B-APP], however it produces a value-tree which has one more subtree, θ_{n+2} , which is produced by evaluating the body of the function ℓ with respect to the value-tree environment $\pi^{\ell, n}(\Theta)$ containing only the value-trees associated to the evaluation of the body of the same function ℓ .

To illustrate rule [E-REP] (`rep` construct), as well as computational rounds, we consider program $(\text{rep } x \ 0 \ (+ \ x \ 1))$ (cf. Section 2). The first firing of a device δ after activation or reset is performed against the empty tree environment. Therefore, according to rule [E-REP], to evaluate $(\text{rep } x \ 0 \ (+ \ x \ 1))$ means to evaluate the subexpression $(+ \ 0 \ 1)$, obtained from $(+ \ x \ 1)$ by replacing x with 0. This produces the value-tree $\theta_1 = 1(1(0, 1, +))$, where root 1 is the overall result as usual, while its sub-tree is the result of evaluating the third argument. Any subsequent firing of the device δ is performed with respect to a tree environment Θ that associates to δ the outcome of the most recent firing of δ . Therefore, evaluating $(\text{rep } x \ 0 \ (+ \ x \ 1))$ at the second firing means to evaluate the subexpression $(+ \ 1 \ 1)$, obtained from $(+ \ x \ 1)$ by replacing x with 1, which is the root of θ_1 . Hence the results of computation are 1, 2, 3, and so on.

Value-trees also support modelling information exchange through the `nbr` construct, as of rule [E-NBR]. Consider the program $e' = (\text{min-hood}(\text{nbr}(\text{sns-num})))$, where the 1-ary built-in function `min-hood` returns the lower limit of values in the

range of its field argument, and the 0-ary built-in function `sns-num` returns the numeric value measured by a sensor. Suppose that the program runs on a network of three fully connected devices δ_A , δ_B , and δ_C where `sns-num` returns 1 on δ_A , 2 on δ_B , and 3 on δ_C . Considering an initial empty tree-environment \emptyset on all devices, we have the following: the evaluation of `(sns-num)` on δ_A yields $1(\text{sns-num})$ (by rules [E-LOC] and [E-B-APP], since $\varepsilon_{\delta_A; \emptyset}^{\text{sns-num}}() = 1$); the evaluation of `(nbr (sns-num))` on δ_A yields $(\delta_A \mapsto 1)(1(\text{sns-num}))$ (by rule [E-NBR]); and the evaluation of e' on δ_A yields

$$\theta_A = 1((\delta_A \mapsto 1)(1(\text{sns-num})), \text{min-hood})$$

(by rule [E-B-APP], since $\varepsilon_{\delta_A; \emptyset}^{\text{min-hood}}((\delta_A \mapsto 1)) = 1$). Therefore, after its first firing, device δ_A produces the value-tree θ_A . Similarly, after their first firing, devices δ_B and δ_C produce the value-trees

$$\begin{aligned} \theta_B &= 2((\delta_B \mapsto 2)(2(\text{sns-num})), \text{min-hood}) \\ \theta_C &= 3((\delta_C \mapsto 3)(3(\text{sns-num})), \text{min-hood}) \end{aligned}$$

respectively. Suppose that device δ_B is the first device that fires a second time. Then the evaluation of e' on δ_B is now performed with respect to the value tree environment $\Theta_B = (\delta_A \mapsto \theta_A, \delta_B \mapsto \theta_B, \delta_C \mapsto \theta_C)$ and the evaluation of its subexpressions `(nbr (sns-num))` and `(sns-num)` is performed, respectively, with respect to the following value-tree environments obtained from Θ_B by alignment:

$$\begin{aligned} \Theta'_B &= \pi_1(\Theta_B) = (\delta_A \mapsto (\delta_A \mapsto 1)(1(\text{sns-num})), \delta_B \mapsto \dots, \delta_C \mapsto \dots) \\ \Theta''_B &= \pi_1(\Theta'_B) = (\delta_A \mapsto 1(\text{sns-num}), \delta_B \mapsto 2(\text{sns-num}), \delta_C \mapsto 3(\text{sns-num})) \end{aligned}$$

We have that $\varepsilon_{\delta_B; \Theta''_B}^{\text{sns-num}}() = 2$; the evaluation of `(nbr (sns-num))` on δ_B with respect to Θ'_B yields $\phi(2(\text{sns-num}))$ where $\phi = (\delta_A \mapsto 1, \delta_B \mapsto 2, \delta_C \mapsto 3)$; and $\varepsilon_{\delta_B; \Theta_B}^{\text{min-hood}}(\phi) = 1$. Therefore the evaluation of e' on δ_B produces the value-tree $1(\phi(2(\text{sns-num})), \text{min-hood})$. Namely, the computation at device δ_B after the first round yields 1, which is the minimum of `sns-num` across neighbours—and similarly for δ_A and δ_C .

We now present an example illustrating first-class functions. Consider the program `((pick-hood (nbr (sns-fun))))`, where the 1-ary built-in function `pick-hood` returns at random a value in the range of its field argument, and the 0-ary built-in function `sns-fun` returns a 0-ary function returning a value of type `num`. Suppose that the program runs again on a network of three fully connected devices δ_A , δ_B , and δ_C where `sns-fun` returns $\ell_0 = (\text{fun } () 0)$ on δ_A and δ_B , and returns $\ell_1 = (\text{fun } () e')$ on δ_C , where $e' = (\text{min-hood (nbr (sns-num))})$ is the program illustrated in the previous example. Assume that `sns-num` returns 1 on δ_A , 2 on δ_B , and 3 on δ_C . Then after its first firing, device δ_A produces the value-tree

$$\theta'_A = 0(\ell_0((\delta_A \mapsto \ell_0)(\ell_0(\text{sns-fun})), \text{pick-hood}), 0)$$

where the root of the first subtree of θ'_A is the anonymous function ℓ_0 (defined above), and the second subtree of θ'_A , 0, has been produced by the evaluation of the body 0 of ℓ_0 . After their first firing, devices δ_B and δ_C produce the value-trees

$$\begin{aligned} \theta'_B &= 0(\ell_0((\delta_B \mapsto \ell_0)(\ell_0(\text{sns-fun})), \text{pick-hood}), 0) \\ \theta'_C &= 3(\ell_1((\delta_C \mapsto \ell_1)(\ell_1(\text{sns-fun})), \text{pick-hood}), \theta_C) \end{aligned}$$

respectively, where θ_C is the value-tree for e given in the previous example.

Suppose that device δ_A is the first device that fires a second time. The computation is performed with respect to the value tree environment $\Theta'_A = (\delta_A \mapsto \theta'_A, \delta_B \mapsto \theta'_B, \delta_C \mapsto \theta'_C)$ and produces the value-tree $1(\ell_1(\phi'(\ell_1(\text{sns-fun})), \text{pick-hood}), \theta'_A)$, where

$$\phi' = (\delta_A \mapsto \ell_1, \delta_C \mapsto \ell_1) \text{ and } \theta'_A = 1((\delta_A \mapsto 1, \delta_C \mapsto 3)(1(\text{sns-num})), \text{min-hood}),$$

since, according to rule [E-D-APP], the evaluation of the body e' of ℓ_1 (which produces the value-tree θ''_A) is performed with respect to the value-tree environment $\pi^{\ell_1, 0}(\Theta'_A) = (\delta_C \mapsto \theta_C)$. Namely, device δ_A executed the anonymous function ℓ_1 received from δ_C , and this was able to correctly align with execution of ℓ_1 at δ_C , gathering values perceived by `sns-num` of 1 at δ_A and 3 at δ_C .

Static Semantics (Type-Inference System) We have developed a variant of the Hindley-Milner type system [9] for the HFC calculus. This type system has two kinds of types, *local types* (the types for local values) and *field types* (the types for field values), and is aimed to guarantee the following two properties:

Type Preservation If a well-typed expression e has type T and e evaluates to a value tree θ , then $\rho(\theta)$ also has type T .

Domain Alignment The domain of every field value arising during the evaluation of a well-typed expression on a device δ consists of δ and of the aligned neighbours.

Alignment is key to guarantee that the semantics correctly relates the behaviour of `if`, `nbr`, `rep` and function application—namely, two fields with different domain are never allowed to be combined. Besides performing standard checks (i.e., in a function application expression $(e_{n+1} e_1 \dots e_n)$ the arguments e_1, \dots, e_n have the expected type; in an `if`-expression $(\text{if } e_0 e_1 e_2)$ the condition e_0 has type `bool` and the branches e_1 and e_2 have the same type; etc.) the type system perform additional checks in order to ensure domain alignment. In particular, the type rules check that:

- In an anonymous function $(\text{fun } (\bar{x}) e)$ the free variables \bar{y} of e that are not in \bar{x} have local type. This prevents a device δ from creating a closure $e' = (\text{fun } (\bar{x}) e)[\bar{y} := \bar{\phi}]$ containing field values $\bar{\phi}$ (whose domain is by construction equal to the subset of the aligned neighbours of δ). The closure e' may lead to a domain alignment error since it may be shifted (via the `nbr` construct) to another device δ' that may use it (i.e., apply e' to some arguments); and the evaluation of the body of e' may involve use of a field value ϕ in $\bar{\phi}$ such that the set of aligned neighbours of δ' is different from the domain of ϕ .
- In a `rep`-expression $(\text{rep } x w e)$ it holds that x , w and e have (the same) local type. This prevents a device δ from storing in x a field value ϕ that may be reused in the next computation round of δ , when the set of the set of aligned neighbours may be different from the domain of ϕ .
- In a `nbr`-expression $(\text{nbr } e)$ the expression e has local type. This prevents the attempt to create a “field of fields” (i.e., a field that maps device identifiers to field values)—which is pragmatically often overly costly to maintain and communicate.
- In an `if`-expression $(\text{if } e_0 e_1 e_2)$ the branches e_1 and e_2 have (the same) local type. This prevents the `if`-expression from evaluating to a field value whose domain is different from the subset of the aligned neighbours of δ .

4 A Pervasive Computing Example

We now illustrate the application of first-class functions using a pervasive computing example. In this scenario, people wandering a large environment (like an outdoor festival, an airport, or a museum) each carry a personal device with short-range point-to-point ad-hoc capabilities (e.g. a smartphone sending messages to others nearby via Bluetooth or Wi-Fi). All devices run a minimal “virtual machine” that allows runtime injection of new programs: any device can initiate a new distributed process (in the form of a 0-ary anonymous function), which the virtual machine spreads to all other devices within a specified range (e.g., 30 meters). For example, a person might inject a process that estimates crowd density by counting the number of nearby devices or a process that helps people to rendezvous with their friends, with such processes likely implemented via various self-organisation mechanisms. The virtual machine then executes these using the first-class function semantics above, providing predictable deployment and execution of an open class of runtime-determined processes.

Virtual Machine Implementation The complete code for our example is listed in Figure 4, with syntax coloring to increase readability: grey for comments, red for field calculus keywords, blue for user-defined functions, and green for built-in operators. In this code, we use the following naming conventions for built-ins: functions `sns-*` embed sensors that return a value perceived from the environment (e.g., `sns-injection-point` returns a Boolean indicating whether a device’s user wants to inject a function); functions `*-hood` yield a local value ℓ obtained by aggregating over the field value ϕ in input (e.g., `sum-hood` sums all values in each neighbourhood); functions `*-hood+` behave the same but exclude the value associated with the current device; and built-in functions `pair`, `fst`, and `snd` respectively create a pair of locals and access a pair’s first and second component. Additionally, given a built-in `o` that takes $n \geq 1$ locals and returns a local, the built-ins `o[*,...,*]` are variants of `o` where one or more inputs are fields (as indicated in the bracket, `l` for local or `f` for field), and the return value is a field, obtained by applying operator `o` in a point-wise manner. For instance, `as =` compares two locals returning a Boolean, `= [f, f]` is the operator taking two field inputs and returns a Boolean field where each element is the comparison of the corresponding elements in the inputs, and similarly `= [f, l]` takes a field and a local and returns a Boolean field where each element is the comparison of the corresponding element of the field in input with the local.

The first two functions in Figure 4 implement frequently used self-organisation mechanisms. Function `distance-to`, also known as *gradient* [8, 14], computes a field of minimal distances from each device to the nearest “source” device (those mapping to `true` in the Boolean input field). This is computed by repeated application of the triangle inequality (via `rep`): at every round, source devices take distance zero, while all others update their distance estimates `d` to the minimum distance estimate through their neighbours (`min-hood+` of each neighbour’s distance estimate (`nbr d`) plus the distance to that neighbour `nbr-range`); source and non-source are discriminated by `mux`, a built-in “multiplexer” that operates like an `if` but differently from it always evaluates both branches on every device. Repeated application of this update procedure self-stabilises

```

;; Computes a field of minimum distance from 'source' devices
(def distance-to (source) ;; has type: (bool) → num
  (rep d infinity (mux source 0 (min-hood+ (+[f,f] (nbr d) (nbr-range))))))

;; Computes a field of pairs of distance to nearest 'source' device, and the most recent value of 'v' there
(def gradcast (source v) ;; has type: ∀ β. (bool, β) → β
  (snd ((fun (x)
    (rep t x (mux source (pair 0 v)
      (min-hood+
        (pair[f,f] (+[f,f] (nbr-range) (nbr (fst t)))
          (nbr (snd t)))))))
    (pair infinity v)))

;; Evaluate a function field, running 'f' from 'source' within 'range' meters, and 'no-op' elsewhere
(def deploy (range source g no-op) ;; has type: ∀ β. (num, bool, () → β, () → β) → β
  ((if (< (distance-to source) range) (gradcast source g) no-op)))

;; The entry-point function executed to run the virtual machine on each device
(def virtual-machine () ;; has type: () → num
  (deploy (sns-range) (sns-injection-point) (sns-injected-fun) (fun () 0)))

;; Sums values of 'summand' into a minimum of 'potential', by descent
(def converge-sum (potential summand) ;; has type: (num, num) → num
  (rep v summand (+ summand
    (sum-hood+ (mux[f,f,1] (= [f,1] (nbr (parent potential)) (uid))
      (nbr v) 0))))))

;; Maps each device to the uid of the neighbour with minimum value of 'potential'
(def parent (potential) ;; has type: (num) → num
  (snd (min-hood (pair[1,f] potential
    (mux[f,f,1] (< [f,1] (nbr potential) potential)
      (nbr (uid)) NaN))))))

;; Simple low-pass filter for smoothing noisy signal 'value' with rate constant 'alpha'
(def low-pass (alpha value) ;; has type: (num, num) → num
  (rep filtered value (+ (* value alpha) (* filtered (- 1 alpha))))))

```

Fig. 4: Virtual machine code (top) and application-specific code (bottom).

into the desired field of distances, regardless of any transient perturbations or faults [13]. The second self-organisation mechanism, `gradcast`, is a directed broadcast, achieved by a computation identical to that of `distance-to`, except that the values are pairs (note that `pair[f,f]` produces a field of pairs, not a pair of fields), with the second element set to the value of `v` at the source: `min-hood` operates on pairs by applying lexicographic ordering, so the second value of the pair is automatically carried along shortest paths from the source. The result is a field of pairs of distance and most recent value of `v` at the nearest source, of which only the value is returned.

The latter two functions in Figure 4 use these self-organisation methods to implement our simple virtual machine. Code mobility is implemented by function `deploy`, which spreads a 0-ary function `g` via `gradcast`, keeping it bounded within distance `range` from sources, and holding 0-ary function `no-op` elsewhere. The corresponding field of functions is then executed (note the double parenthesis). The `virtual-machine` then simply calls `deploy`, linking its arguments to sensors configuring deployment range and detecting who wants to inject which functions (and using `(fun () 0)` as no-op function).

In essence, this virtual machine implements a code-injection model much like those used in a number of other pervasive computing approaches (e.g., [15, 11, 6])—though of course it has much more limited features, since it is only an illustrative example. With these previous approaches, however, code shares lexical scope and cannot have its network domain externally controlled. Thus, injected code may spread through the network unpredictably and may interact unpredictably with other injected code that it encounters. The extended field calculus semantics that we have presented, however, ensures that injected code moves only within the range specified to the virtual machine and remains lexically isolated from different injected code, so that no variable can be unexpectedly affected by interactions with neighbours.

Simulated Example Application We further illustrate the application of first-class functions with an example in a simulated scenario. Consider a museum, whose docents monitor their efficacy in part by tracking the number of patrons nearby while they are working. To monitor the number of nearby patrons, each docent’s device injects the following anonymous function (of type: $() \rightarrow \text{num}$):

```
(fun () (low-pass 0.5 (converge-sum (distance-to (sns-injection-point))
                                   (sns-patron))))
```

This counts patrons using the function `converge-sum` defined in Figure 4(bottom), a simple version of another standard self-organisation mechanism [4] which operates like an inverse broadcast, summing the values sensed by `sns-patron` (1 for a patron, 0 for a docent) down the distance gradient back to its source—in this case the docent at the injection point. In particular, each device’s local value is summed with those identifying it as their parent (their closest neighbour to the source, breaking ties with device unique identifiers from built-in function `uid`), resulting in a relatively balanced spanning tree of summations with the source at its root. This very simple version of summation is somewhat noisy on a moving network of devices, so its output is passed through a simple low-pass filter, the function `low-pass`, also defined in Figure 4(bottom), in order to smooth its output and improve the quality of estimate.

Figure 5a shows a simulation of a docent and 250 patrons in a large 100x30 meter museum gallery. Of the patrons, 100 are a large group of school-children moving together past the stationary docent from one side of the gallery to the other, while the rest are wandering randomly. In this simulation, people move at an average 1 m/s, the docent and all patrons carry personal devices running the virtual machine, executing asynchronously at 10Hz, and communicating via low-power Bluetooth to a range of 10 meters. The simulation was implemented using the ALCHEMIST [18] simulation framework and the Protelis [17] incarnation of field calculus, updated to the extended version of the calculus presented in this paper.

In this simulation, at time 10 seconds, the docent injects the patron-counting function with a range of 25 meters, and at time 70 seconds removes it. Figure 5a shows two snapshots of the simulation, at times 11 (top) and 35 (bottom) seconds, while Figure 5b compares the estimated value returned by the injected process with the true value. Note that upon injection, the process rapidly disseminates and begins producing good estimates of the number of nearby patrons, then cleanly terminates upon removal.

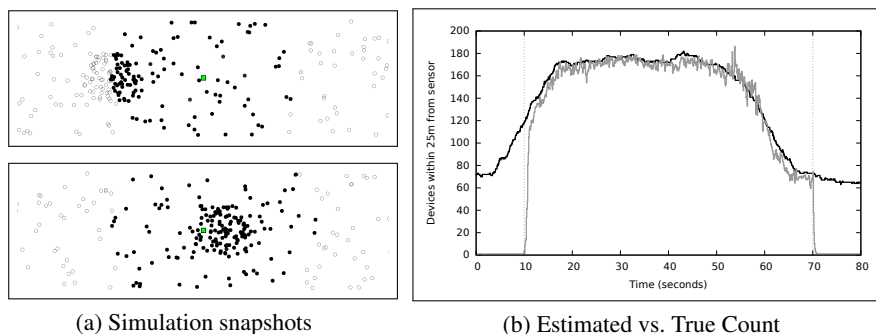


Fig. 5: (a) Two snapshots of museum simulation: patrons (grey) are counted (black) within 25 meters of the docent (green). (b) Estimated number of nearby patrons (grey) vs. actual number (black) in the simulation.

5 Conclusion, Related and Future Work

Conceiving emerging distributed systems in terms of computations involving aggregates of devices, and hence adopting higher-level abstractions for system development, is a thread that has recently received a good deal of attention. A wide range of aggregate programming approaches have been proposed, including Proto [2], TOTA [15], the (bio)chemical tuple-space model [19], Regiment [16], the $\sigma\tau$ -Linda model [22], Paintable Computing [6], and many others included in the extensive survey of aggregate programming languages given in [3]. Those that best support self-organisation approaches to robust and environment-independent computations have generally lacked well-engineered mechanisms to support openness and code mobility (injection, update, etc.). Our contribution has been to develop a core calculus, building on the work presented in [21], that smoothly combines for the first time self-organisation and code mobility, by means of the abstraction of “distributed function field”. This combination of first-class functions with the domain-restriction mechanisms of field calculus allows the predictable and safe composition of distributed self-organisation mechanisms at runtime, thereby enabling robust operation of open pervasive systems. Furthermore, the simplicity of the calculus enables it to easily serve as both an analytical framework and a programming framework, and we have already incorporated this into Protelis [17], thereby allowing these mechanisms to be deployed both in simulation and in actual distributed systems.

Future plans include consolidation of this work, by extending the calculus and its conceptual framework, to support an analytical methodology and a practical toolchain for system development, as outlined in [4]. First, we aim to apply our approach to support various application needs for dynamic management of distributed processes [1], which may also impact the methods of alignment for anonymous functions. Second, we plan to isolate fragments of the calculus that satisfy behavioural properties such as self-stabilisation, quasi-stabilisation to a dynamically evolving field, or density independence, following the approach of [20]. Finally, these foundations can be applied in developing APIs enabling the simple construction of complex distributed applications, building on the work in [4] to define a layered library of self-organisation patterns, and applying these APIs to support a wide range of practical distributed applications.

References

1. J. Beal. Dynamically defined processes for spatial computers. In *Spatial Computing Workshop*, pages 206–211, New York, September 2009. IEEE.
2. J. Beal and J. Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, 21:10–19, March/April 2006.
3. J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. In M. Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter 16, pages 436–501. IGI Global, 2013. A longer version available at: <http://arxiv.org/abs/1202.5509>.
4. J. Beal and M. Viroli. Building blocks for aggregate programming of self-organising applications. In *2nd FoCAS Workshop on Fundamentals of Collective Systems*, pages 1–6. IEEE CS, to appear, 2014.
5. J. Beal, M. Viroli, and F. Damiani. Towards a unified model of spatial computing. In *7th Spatial Computing Workshop (SCW 2014)*, AAMAS 2014, Paris, France, May 2014.
6. W. Butera. *Programming a Paintable Computer*. PhD thesis, MIT, Cambridge, USA, 2002.
7. A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.
8. L. Clement and R. Nagpal. Self-assembly and self-repairing topologies. In *Workshop on Adaptability in Multi-Agent Systems, RoboCup Australian Open*, 2003.
9. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages*, POPL '82, pages 207–212. ACM, 1982.
10. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
11. D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.
12. 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*, 23(3), 2001.
13. S. Kutten and B. Patt-Shamir. Time-adaptive self stabilization. In *Proceedings of ACM symposium on Principles of distributed computing*, pages 149–158. ACM, 1997.
14. F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Trans. Softw. Eng.*, 13(1):32–38, 1987.
15. M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. on Software Engineering Methodologies*, 18(4):1–56, 2009.
16. R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *Workshop on Data Management for Sensor Networks*, pages 78–87, Aug. 2004.
17. D. Pianini, J. Beal, and M. Viroli. Practical aggregate programming with PROTELIS. In *ACM Symposium on Applied Computing (SAC 2015)*, 2015. To appear.
18. D. Pianini, S. Montagna, and M. Viroli. Chemical-oriented simulation of computational systems with Alchemist. *Journal of Simulation*, 7:202–215, 2013.
19. M. Viroli, M. Casadei, S. Montagna, and F. Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Transactions on Autonomous and Adaptive Systems*, 6(2):14:1 – 14:24, June 2011.
20. M. Viroli and F. Damiani. A calculus of self-stabilising computational fields. In *Coordination Languages and Models*, volume 8459 of LNCS, pages 163–178. Springer-Verlag, June 2014.
21. M. Viroli, F. Damiani, and J. Beal. A calculus of computational fields. In *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Science*, pages 114–128. Springer Berlin Heidelberg, 2013.
22. M. Viroli, D. Pianini, and J. Beal. Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In *Proceedings of Coordination 2012*, volume 7274 of *Lecture Notes in Computer Science*, pages 212–229. Springer, 2012.