A Calculus of Computational Fields

Mirko Viroli[1], Ferruccio Damiani[2], and Jacob Beal[3]

[1] University of Bologna, Italy
mirko.viroli@unibo.it
[2] University of Torino, Italy
ferruccio.damiani@unito.it
[3] Raytheon BBN Technologies, USA
jakebeal@bbn.com

Abstract. A number of recent works have investigated the notion of "computational fields" as a means of coordinating systems in distributed, dense and mobile environments such as pervasive computing, sensor networks, and robot swarms. We introduce a minimal core calculus meant to capture the key ingredients of languages that make use of computational fields: functional composition of fields, functions over fields, evolution of fields over time, construction of fields of values from neighbours, and restriction of a field computation to a sub-region of the network. This calculus can act as a core for actual implementation of coordination languages and models, as well as pave the way towards formal analysis of properties concerning expressiveness, self-stabilisation, topology independence, and relationships with the continuous space-time semantics of spatial computations.

1 Introduction

In a world ever more densely saturated with computing devices, it is increasingly important to have effective tools for developing coordination strategies that can govern collections of these devices. The goals of such systems are typically best expressed in terms of operations and behaviours over aggregates of devices, e.g., "send a tornado warning to all phones in the forecast area,", or "activate all displays in the route towards the nearest group of friends of mine." Effective models and programming languages are needed to allow the construction of distributed systems at the natural level of aggregates of devices, contrasting with the classical individual-device view that often obfuscates the system design.

Recently, approaches based on models of computation over continuous space and time have been introduced, which promise to deliver aggregate programming capabilities for the broad class of *spatial computers*: networks of devices embedded in space, such that the difficulty of moving information between devices is strongly correlated with the physical distance between devices. Examples of spatial computers include sensor networks, robot swarms, mobile ad-hoc networks, reconfigurable computing, emerging pervasive computing scenarios, and colonies of engineered biological cells.

A large number of formal models, programming languages, and infrastructures have been created with the aim of supporting computation over space-time, surveyed in [5]. Several of these are directly related to the field of coordination models and languages,

C. Canal and M. Villari (Eds.): ESOCC 2013, CCIS 393, pp. 114–128, 2013.

such as the pioneer model of TOTA [11], the (bio)chemical tuple-space model [17], the $\sigma\tau$-Linda model [19], and the pervasive ecosystems model in [13]. Their recurrent core idea is that through a process of diffusion, recombination, and composition, information injected in one device (or a few devices) can produce global, dynamically evolving *computational fields*—functions mapping each device to a structured value. Such fields are aggregate-level distributed data structures which, due to the ongoing feedback loops that produce and maintain them, are generally robust to changes in the underlying topology (e.g., due to faults, mobility, or openness) and to unexpected interactions with the external environment. They are thus useful for implementing and composing self-organising coordination patterns to adaptively regulate the behaviour of complex distributed systems [11,17,18].

A sound engineering methodology for space-time coordination systems will require more than just specification, but the ability to predict to a good extent the behaviour of computational fields from the underlying local interaction rules—a problem currently solved only for particular cases [4]. This paper contributes to that goal by introducing a core calculus meant to precisely capture a set of key ingredients of programming languages supporting the creation of computational fields: composition of fields, functions over fields, evolution of fields over time, construction of fields of values from neighbours, and restriction of a field computation to a sub-region of the network.

The proposed calculus is largely inspired by Proto [3,12], the archetypal spatial computing language (and is in fact a fragment of it). As with Proto, it is based on the idea of expressing aggregate system behaviour by a functional composition of operators that manipulate (evolve, combine, restrict) continuous fields. Critically, these specifications can be also interpreted as local rules on individual devices, which are iteratively executed in asynchronous "computation rounds", comprising reception of messages from neighbours, computing the local value of fields, and spreading messages to neighbours. The operational semantics of the proposed calculus precisely models single device computation, which is ultimately responsible for the whole network execution. The distinguished interaction model of this approach, which is first formalised into a calculus in this paper, is based on representing state and message content in an unified way as an annotated evaluation tree. Field construction, propagation, and restriction are then supported by local evaluation "against" the evaluation trees received from neighbours.

The calculus thus developed formalises key constructs of existing coordination languages or models targeting spatial computing. As such, we believe it paves the way towards formal analysis of key properties applicable to various coordination systems, concerning soundness, expressiveness, self-stabilisation, topology independence, and relationships with the continuous space-time semantics of spatial computations.

The remainder of the paper is organized as follows. Section 2 describes the proposed linguistic constructs and their application to system coordination. Section 3 illustrates how single devices interpret the proposed constructs locally. Section 4 presents the formal calculus. Section 5 discusses the soundness property of the calculus. Section 6 concludes by discussing related works and outlining possible directions for future works.

`e ::= x` $\mid$ `l` $\mid$ `(o` $\overline{\texttt{e}}$`)` $\mid$ `(f` $\overline{\texttt{e}}$`)` $\mid$ `(rep x w e)` $\mid$ `(nbr e)` $\mid$ `(if e e e)`	expression
`w ::= x` $\mid$ `l`	variable or value
`F ::= (def f(`$\overline{\texttt{x}}$`) e)`	function
`P ::=` $\overline{\texttt{F}}$ `e`	program

Fig. 1. Surface syntax

2 Computational Fields

Generalising the common notion of scalar and vector field in physics, a computational field is a map from every computational device in a space to an arbitrary computational object. Examples of fields used in distributed situated systems include temperature in a building as perceived by a sensor network (a scalar field), the best routes to get to a location (a vector field), the area near an object of interest (a boolean indicator field), or the people allowed access to computational resources in particular areas (a set-valued field). With careful choice of operators for manipulating fields, the aggregate and local views of a program can be kept coherent and each element of the aggregate-level program can be implemented by simple, automatically generated local interaction rules [2]. Following this idea, in this section we present a core language to express such operators. This language is identified based on the strengths and commonalities across many different approaches to spatial computing reviewed in [5] (though we do not rule out the possibility that others may be identified), and drawing on the Proto [3,12] implementations of these mechanisms.

We describe the selected mechanisms directly showing the syntax of the proposed calculus, reported in Figure 1. We take the global, aggregate-level viewpoint, considering the main syntactic element `e` as being a field expression, or simply a field. As a standard syntactic notation in calculi for object-oriented and functional languages [10], we use the overbar notation to denote metavariables over lists, e.g., we let $\overline{\texttt{e}}$ range over lists of expressions, written $\texttt{e}_1\ \texttt{e}_2\ \ldots\ \texttt{e}_n$.

A basic expression can be a literal value `l` (also called local value), such as a floating point number, a boolean, or a tuple—note most of the ideas of computational fields are agnostic to the structure of such values. According to the global viewpoint, a literal field expression `l` actually represents the constant function mapping `l` to all nodes. A basic expression can also be a variable `x`, which can be the formal parameter of a function or a store of information to support stateful computations (see `rep` construct below).

Such basic expressions (values and variables) can be composed by the following 5 constructs. The first one is *functional composition*, a natural means of manipulating fields as they are functions themselves: $(\texttt{o}\ \texttt{e}_1\ \texttt{e}_2\ \ldots\ \texttt{e}_n)$ is the field obtained by composing together all the fields $\texttt{e}_1, \texttt{e}_2, \ldots, \texttt{e}_n$ by an operator `o`. Operators are built-in, and include standard mathematical ones (e.g. addition, sine): they are applied in a pointwise manner to all devices. For instance, if $\texttt{e}_t$ is a field of Fahrenheit temperatures, then the corresponding field of Celsius temperatures is naturally written `(* (/ 5 9) (- `$\texttt{e}_t$` 32))`. Execution of built-in operators is context-dependent, i.e., it can be affected by the current state of the external world. So, 0-ary operator `self` gives a field that maps each device to its identifier, `dt` maps each device to the time

elapsed since its previous computation round, and finally `nbr-range` maps each device to a table associating estimated distances to each neighbour (such a table being a field itself).

The second construct is *function (definition and) call*, which we use as abstraction tool and to support recursion: (`f` e_1 e_2 … e_n) is the field obtained as result of applying user-defined function `f` to the fields e_1, e_2, … e_n. Such functions are declared with syntax (`def f(`$\overline{x}$`) e`). For instance, after definition `(def convert (x) (* (/ 5 9) (- x 32)))`, expression `(convert` e_t`)` denotes the same field of Celsius temperatures as above. Note that function definitions, along with the top-level expression, form a program `P`.

The third construct is *time evolution*, used to keep track of a changing state over time: (`rep x w e`) is initially the field `w` (a local value or a variable) that is stored in the new variable `x`, and at each step in time is updated to a new field as computed by `e`, based on 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. Similarly, `(rep x 0 (+ x (dt)))` is the field of time passing.

The fourth construct is *neighbourhood field construction*, the mechanism by which information moves between devices: (`nbr e`) maps each device to the field of its neighbours' local value of field `e`; hence, it is a field of neighbourhood fields like the output of `nbr-range` above. As an example, let `min-hood` be the operator taking a neighbourhood field and returning its minimum value, then `(min-hood (nbr` e_t`))` is the field mapping each device to the minimum temperature perceived in its neighbourhood.

The last construct is *domain restriction*, a sort of distributed branch: (`if` e_0 e_1 e_2) is the field obtained by superimposing field e_1 computed everywhere e_0 is true and e_2 everywhere e_0 is false. As an example `(if` e_{fah} e_t `(convert` e_t`))` is the field of temperatures provided in Fahrenheit (resp. Celsius) where the field e_{fah} is true (resp. false). Restriction is the most subtle of the five mechanisms, because it has the effect of preventing the unexpected spreading of computation to devices outside of the required domain, even within arbitrarily nested function calls, as will be clarified in the following.

We now present some examples to illustrate how these five key mechanisms can be combined to implement useful spatial patterns.

```
(def gossip-min (source) (rep d source (min-hood (nbr d))))

(def distance-to (source)
  (rep d infinity (mux source 0 (min-hood (+ (nbr d) (nbr-range))))))

(def distance-obs-to (source obstacle)
      (if (not obstacle) (distance-to source) infinity))
```

We first exemplify how constructs `rep` and `nbr` can be nested to create a long-distance computation, to achieve network-wide propagation processes. Function `gossip-min` takes a `source` field and produces a new field mapping each device to the minimum value that `source` initially takes. The `rep` construct initially sets the output variable `d` at `source`, and it iteratively updates the value at each device with the minimum

one available in d's neighbours. Hence, `gossip-min` describes a process of gossiping values until the minimum one converges throughout the network.

Similarly, function `distance-to` takes as its input a `source` field holding boolean values, and returns a new scalar field that maps each device to the estimated distance to the nearest device where `source` is true. This works by first setting `d` to infinity, then updating it as follows: sources are of course at distance 0, while all other devices use the triangle inequality, finding the minimum sum of a neighbour's estimated distance `d` and the distance to that neighbour. Operator `mux`, used to combine the two, is a purely functional multiplexer, which uses the first input to choose whether to return the second or third. The field returned by `distance-to` is often also referred to as a *gradient* [11,4,17], and is a key building block for many computations in mobile ad-hoc networks, such as finding routes to points of interest. There are many similar variants with different purposes, most of which automatically repair themselves when either the sources or network structure change.

The last definition exemplifies the use of construct `if`. It creates two different spatial domains: one where the obstacle is present (field `obstacle` holds positive boolean value) and one where is not. In the former an infinity constant field is computed; in the latter we spread the `distance-to` field. As a result, distance estimation as provided by `distance-to` automatically takes into account the need of circumventing obstacle areas, since information does not cross the two domains due to the semantics of `nbr` as explained in next section.

A number of coordination mechanisms can be constructed on the basis of these examples, like the gradient-based patterns discussed in [17,13,19], which find applications in many areas, including crowd steering in pervasive computing.

3 From Global to Local

The description of field constructs so far has focused on what we can call the *global viewpoint*, in which the computation is considered as occurring on the overall computational fields distributed in the network. For the calculus to be actually executed, however, each device has to perform a specific set of actions at particular times, including interaction with neighbours and local computations. The result of these local actions then produces the overall evolution of computational fields. We call this description of the language in term of individual devices the *local viewpoint*, and it is this view that we shall use for the operational semantics. Let us now begin with an informal presentation of the peculiar aspects of that operational semantics, to aid in understanding the full formalisation presented in Section 4.

Following the approach considered in Proto [12] and many other distributed programming languages, devices undergo computation in rounds. In each round, a device sleeps for some limited time, wakes up, gathers information about messages received while sleeping, performs its actual field evaluation, and finally emits a message to all neighbours with information about the outcome of computation, before going back to sleep.

Taking the local viewpoint, we may model a field computation by modeling the evaluation of a single device at a single round, assuming the scheduling of such rounds

across the network be fair and non-synchronous—either fully asynchronous or partially synchronous, meaning that devices cannot execute infinitely quickly. Assuming that the main bottleneck in the system is communication rather than computation (which is frequently the case in wireless communication networks), this model can be readily achieved by any collection of devices with internal clocks that schedule execution of rounds at regular intervals. So long as the relative drift between clocks is not extreme, execution on such a system will be fair and partially synchronous.

To support the combination of field constructs, we design our operational semantics as follows. First, our functional style of composition, definition and calls, fits well with a small-step evaluation semantics, in which we start from the initial expression to evaluate and reduce it to a normal form representing the outcome of computation, including the local value of the resulting field and the information to be spread to neighbours. In order to keep track of the state of variables introduced by `rep` constructs, and values at `nbr` constructs to be exchanged with neighbours, we take our computational state to be the dynamically produced evaluation tree. During a round of computation, such a tree is incrementally decorated with partial results expressed as *annotations* of the form "$\cdot v$" or *superscripts* "^{s}". These decorations track the local outcome of evaluation and determine which subexpression will be next evaluated.

To illustrate our management of evaluation order and computational rounds, as well as the `rep` construct, let us begin by considering expression `(rep x 0 (+ x 1))` (cf. Section 2). As this tree is evaluated according to the operational semantics, it goes through a sequence of four transitions. We show these informally by in each step underlining the next portion of the tree to be rewritten, by coloring the changes introduced by each rewrite red (they will appear grey in a non-color print of the paper), and by labelling the transitions with the (nested) rules of the operational semantics causing the transition. The rules may be ignored for now, and be considered later to understand the formal calculus in Section 4. The first computation round goes as follows:

$$\texttt{(rep x 0 (+ } \underline{\texttt{x}} \texttt{ 1))} \xrightarrow{\text{[REP,CONG,VAR]}} \texttt{(rep x 0 (+ x}\cdot\texttt{0 } \underline{\texttt{1}}\texttt{))} \xrightarrow{\text{[REP,CONG,VAL]}}$$
$$\texttt{(rep x 0 } \underline{\texttt{(+ x}\cdot\texttt{0 1}\cdot\texttt{1))}} \xrightarrow{\text{[REP,CONG,OP]}} \underline{\texttt{(rep x 0 (+ x}\cdot\texttt{0 1}\cdot\texttt{1)}\cdot\texttt{1)}}$$
$$\xrightarrow{\text{[REP]}} \texttt{(rep}^{1}\texttt{ x 0 (+ x}\cdot\texttt{0 1}\cdot\texttt{1)}\cdot\texttt{1)}\cdot\texttt{1}$$

Annotations are computed depth-first in the expression tree until eventually reaching the outer expression: we first annotate variable `x` with its current (initial) value 0, then simply identically annotate value 1, then perform built-in operation + causing annotation of its sub-tree with 1, and finally execute the `rep` construct, which records the result value as a superscript to `rep` and as an annotation of the whole expression.

Once the evaluation is complete, with the result value in the outer-most annotation, the whole evaluation tree will be shipped as a message to neighbours, in order to align `nbr` statements and share values between neighbours, as described later. Pragmatically, of course, any implementation might massively compress the tree, sending only enough information for `nbr` statements to be aligned.

The subsequent round begins after an initialisation that erases all non-superscript decorations. This second round leads to evaluation tree $\texttt{(rep}^{2}\texttt{ x 0 (+ x}\cdot\texttt{1 1}\cdot\texttt{1)}\cdot\texttt{2)}\cdot\texttt{2}$, third one to $\texttt{(rep}^{3}\texttt{ x 0 (+ x}\cdot\texttt{2 1}\cdot\texttt{1)}\cdot\texttt{3)}\cdot\texttt{3}$, and so on.

The main purpose of managing evaluation trees in this way is to support information exchange through the `nbr` construct. Consider the expression `(min-hood (nbr (t)))` (cf. Section 2), where `t` is a 0-ary built-in operator that returns the temperature perceived in each device. If a device σ perceives a temperature of 7 degrees Celsius, and executes its first computation round before its neighbours, then the result of computation should clearly be 7. This is implemented by the following sequence of transitions:

$$\texttt{(min-hood (nbr (t)))} \xrightarrow{\text{[CONG,CONG,OP]}} \texttt{(min-hood (nbr (t)}\cdot 7\texttt{))} \xrightarrow{\text{[CONG,NBR]}}$$
$$\texttt{(min-hood (nbr (t)}\cdot 7\texttt{)}\cdot(\sigma \mapsto 7)) \xrightarrow{\text{[OP]}} \texttt{(min-hood (nbr (t)}\cdot 7\texttt{)}\cdot(\sigma \mapsto 7))\cdot 7$$

We first enter the subexpression with the 0-ary operator `t` which yelds 7. We then evaluate `nbr` to the field of neighbour values, associating only σ to 7, written $(\sigma \mapsto 7)$. Finally, we evaluate unary operator `min-hood`, which extracts the smallest element of the input field, which in this case is 7.

Construct `nbr` retrieves values from neighbours using the *tree environment* of the device σ, which models its store of recent messages received from neighbours. The tree environment is a mapping $\Theta = (\sigma_1 \mapsto e_1, \ldots, \sigma_n \mapsto e_n)$ created at each round, from neighbours (σ_i) to their last-received evaluation tree (e_i), which we call the *neighbour tree* of σ_i . The evaluation of (`nbr` e), where e is evaluated to local value `l`, takes values from the tree environment to produce a field $(\sigma \mapsto \texttt{l}, \sigma_1 \mapsto \texttt{l}_1, \ldots, \sigma_n \mapsto \texttt{l}_n)$, mapping σ to `l` and each neighbour σ_i to the corresponding local value $\texttt{l}_i$ from σ_i.

In the example above we assumed that none of the neighbours of σ had already completed a round of computation, and that therefore Θ was empty and accordingly `(nbr (t))` gave simply $(\sigma \mapsto 7)$. If we instead assume that the first round of computation on the device σ takes place when the neighbours σ_1 and σ_2 have completed exactly one round of computation, perceiving temperatures of 4 and 9 degrees respectively, then the tree environment of σ would be $(\sigma_1 \mapsto e_1, \sigma_2 \mapsto e_2)$, where $e_1 =$ `(min-hood (nbr (t)`$\cdot 4)\cdot(\sigma \mapsto 4))\cdot 4$ and $e_2 =$ `(min-hood (nbr (t)`$\cdot 9)\cdot(\sigma \mapsto 9))\cdot 9$. The computation goes similarly, the only difference is that the evaluation of `(nbr (t)`$\cdot 7$) now produces the field $\phi = (\sigma \mapsto 7, \sigma_1 \mapsto 4, \sigma_2 \mapsto 9)$ and the final outcome of the computation round on σ is the tree `(min-hood (nbr (t)`$\cdot 7)\cdot\phi)\cdot 4$.

More specifically, the extraction of values from neighbours is achieved by computing the local evaluation tree "against" the set of its neighbour trees: when evaluation enters a subtree, in the tree environment Θ we correspondingly enter the corresponding subtree on all of its neighbour trees, which are structurally compatible by construction since each node executes the same program. This process on neighbour trees is called *alignment*. So, in the example above, sub-tree `(nbr (t)`$\cdot 7$) is recursively evaluated against the neighbour sub-trees $(\sigma_1 \mapsto$ `(nbr (t)`$\cdot 4) \cdot (\sigma_1 \mapsto 4), \sigma_2 \mapsto$ `(nbr (t)`$\cdot 9) \cdot (\sigma_2 \mapsto 9))$, in which the neighbour values are immediately available as the outermost annotation of the argument of `nbr`.

One reason for using this structural alignment mechanism is to seamlessly handle the cases where `nbr` subtrees could be nested at a deep level of the evaluation tree because of (possibly recursive) function calls. Assume definition `(def f (x) (min-hood (nbr x)))`, and the main expression `(f (t))` whose

expected behaviour is then equivalent to our prior example `(min-hood (nbr (t)))`. This expression would be handled by the following sequence of transitions:

$$\texttt{(f (t))} \xrightarrow{\text{[CONG,OP]}} \texttt{(f (t)}\cdot 7\texttt{)} \xrightarrow{\text{[FUN,CONG,CONG,VAR]}} \texttt{(f}^{\texttt{(min-hood (nbr x}\cdot 7\texttt{))}}\ \texttt{(t)}\cdot 7\texttt{)} \xrightarrow{\text{[FUN,CONG,NBR]}}$$
$$\texttt{(f}^{\texttt{(min-hood (nbr x}\cdot 7\texttt{)}\cdot\phi\texttt{)}}\ \texttt{(t)}\cdot 7\texttt{)} \xrightarrow{\text{[FUN,OP]}} \texttt{(f}^{\texttt{(min-hood (nbr x}\cdot 7\texttt{)}\cdot\phi\texttt{)}}\ \texttt{(t)}\cdot 7\texttt{)}\cdot 4$$

After the function arguments are all evaluated, the second transition creates a superscript to function `f`, holding the evaluation tree corresponding to its body. This gets evaluated as usual, and its resulting annotation 4 is transferred to become the annotation of the function call. So, note that the evaluation tree is a dynamically expanding data structure because of such function superscripts being generated and navigated at each call, with alignment automatically handling `nbr` construct, even for arbitrary recursive call structures. Note that this mechanism also prevents terminating recursive calls from implying infinite evaluation trees, since only those calls that are actually made are annotated.

This management of memory trees also easily accommodates the semantics of restriction. An `if` sub-expression is evaluated by first evaluating its condition, then evaluating the selected branch, and finally erasing all decorations on the non-taken branch, including superscripts. In this way, neighbour trees corresponding to devices that took a different branch will be automatically discarded at alignment time, since entering the same subexpression is impossible because of a bad match. For example, consider expression `(if (b) (f (t)) 0)`, where operator `b` returns a boolean field that is true at σ and σ_2, and false at σ_1. Assuming again that first round of σ happens after first round of σ_1 and σ_2, we have:

$$\texttt{(if (b) (f (t)) 0)} \xrightarrow{\text{[CONG,OP]}} \texttt{(if (b)}\cdot\texttt{true (f (t)) 0)} \rightarrow^*$$
$$\texttt{(if (b)}\cdot\texttt{true (f}^{\texttt{(min-hood (nbr x}\cdot 7\texttt{)}\cdot(\sigma\mapsto 7,\sigma_2\mapsto 9)\texttt{)}}\ \texttt{(t)}\cdot 7\texttt{)}\cdot 7\ \texttt{0)} \xrightarrow{\text{[THEN]}}$$
$$\texttt{(if (b)}\cdot\texttt{true (f}^{\texttt{(min-hood (nbr x}\cdot 7\texttt{)}\cdot(\sigma\mapsto 7,\sigma_2\mapsto 9)\texttt{)}}\ \texttt{(t)}\cdot 7\texttt{)}\cdot 7\ \texttt{|0|)}\cdot 7$$

The reason why the `rep` sub-expression now yields field $(\sigma \mapsto 7, \sigma_2 \mapsto 9)$ is that the neighbour tree of σ_1 cannot be aligned, for it has `(b)` annotated with `false`, which does not match. Hence, `nbr` will retrieve values only from the *aligned nodes* that followed the same branch, avoiding interference from nodes residing in different regions of the partition made by restriction. The erasure of the non-taken branch by operator $|.|$ (0 trivially erases to 0 in this case) is used to completely reinitialise computation there, since the node no longer belongs to the domain in which the non-taken branch should be evaluated.

4 The Computational Field Calculus

The computational field calculus formalisation is set forth in Figure 2 and described here in turn after a few preliminaries. We let σ range over device unique identifiers and ϕ over field values (mapping set of devices to local values). Given any meta-variable y we let $\mathring{y}$ range over an element y or the *null decoration* (which in the calculus is $\circ$ when it has to be expressed, and blank otherwise). The calculus is agnostic to the syntax of local values: we only assume they include at least device identifiers and value 0. We let metavariables f and t range over boolean-interpreted values, orderly 0 and any other value.

Runtime Expression Syntax:

$e ::= a\cdot\mathring{v}$ — runtime expression (rte)

$a ::= \mathtt{x} \mid v \mid (\mathtt{nbr}\ e) \mid (\mathtt{if}\ e\ e\ e) \mid (\mathtt{rep}^{\mathtt{s}}\ \mathtt{x}\ \mathtt{w}\ e) \mid (\mathtt{f}^{\mathtt{s}}\ \overline{e}) \mid (\mathtt{o}\ \overline{e})$ — auxiliary rte

$v ::= \mathtt{l} \mid \phi$ — runtime value

$\mathtt{s} ::= \mathring{a}$ — superscript

$\mathtt{w} ::= \mathtt{x} \mid \mathtt{l}$ — variable or local value

$\phi ::= \overline{\sigma} \mapsto \overline{\mathtt{l}}$ — field value

$\Theta ::= \overline{\sigma} \mapsto \overline{e}$ — tree environment

$\Gamma ::= \overline{\mathtt{x}} := \overline{v}$ — variable environment

Congruence Contexts:

$\mathbb{C} ::= (\mathtt{nbr}\ [\,]) \mid (\mathtt{f}^{\mathtt{s}}\ \overline{e}\ [\,]\ \overline{e}) \mid (\mathtt{o}\ \overline{e}\ [\,]\ \overline{e}) \mid (\mathtt{if}\ [\,]\ e\ e) \mid (\mathtt{if}\ a\cdot t\ [\,]\ e) \mid (\mathtt{if}\ a\cdot f\ e\ [\,])$

Alignment contexts:

$\mathbb{A} ::= \mathbb{C} \mid (\mathtt{rep}^{\mathtt{s}}\ \mathtt{x}\ \mathtt{w}\ [\,]) \mid (\mathtt{f}^{[\,]}\ \overline{a\cdot v})$

Auxiliary functions:

$$\pi_{\mathbb{A}}(\Theta,\Theta') = \pi_{\mathbb{A}}(\Theta), \pi_{\mathbb{A}}(\Theta')$$
$$\pi_{\mathbb{A}}(\sigma \mapsto (\mathbb{A}'[e])\cdot v) = \sigma \mapsto e \quad \text{if } \mathbb{A}' :: \mathbb{A}$$
$$\pi_{\mathbb{A}}(\sigma \mapsto e) = \bullet \quad \text{otherwise}$$

$$\mathtt{s} \rhd a = a$$
$$\mathtt{s} \rhd \circ = \mathtt{s}$$

$$(\mathtt{nbr}\ [\,]) :: (\mathtt{nbr}\ [\,])$$
$$(\mathtt{f}^{\mathtt{s}'}\ e'_1 \ldots e'_{i-1}\ [\,]\ e'_{i+1} \ldots e'_n) :: (\mathtt{f}^{\mathtt{s}}\ e_1 \ldots e_{i-1}\ [\,]\ e_{i+1} \ldots e_n)$$
$$(\mathtt{o}\ e'_1 \ldots e'_{i-1}\ [\,]\ e'_{i+1} \ldots e'_n) :: (\mathtt{o}\ e_1 \ldots e_{i-1}\ [\,]\ e_{i+1} \ldots e_n)$$
$$(\mathtt{if}\ [\,]\ e'_1\ e'_2) :: (\mathtt{if}\ [\,]\ e_1\ e_2))$$
$$(\mathtt{if}\ a'\cdot t\ [\,]\ e') :: (\mathtt{if}\ a\cdot t\ [\,]\ e)$$
$$(\mathtt{if}\ a'\cdot f\ e'\ [\,]) :: (\mathtt{if}\ a\cdot f\ e\ [\,])$$
$$(\mathtt{rep}^{\mathtt{s}'}\ \mathtt{x}\ \mathtt{w}\ [\,]) :: (\mathtt{rep}^{\mathtt{s}}\ \mathtt{x}\ \mathtt{w}\ [\,])$$
$$(\mathtt{f}^{[\,]}\ e'_1 \ldots e'_n) :: (\mathtt{f}^{[\,]}\ e_1 \ldots e_n)$$

Reduction Rules:

$$\text{[VAL]}\ \frac{}{\Theta;\Gamma \vdash v \to v\cdot v} \qquad \text{[VAR]}\ \frac{}{\Theta;\Gamma \vdash \mathtt{x} \to \mathtt{x}\cdot\Gamma(\mathtt{x})|_{dom(\Theta),\mathcal{E}(\mathtt{self})}}$$

$$\text{[NBR]}\ \frac{\pi_{(\mathtt{nbr}\ [\,])}(\Theta) = \overline{\sigma} \mapsto \overline{a}\cdot\overline{\mathtt{l}} \qquad \phi = (\overline{\sigma} \mapsto \overline{\mathtt{l}}, \mathcal{E}(\mathtt{self}) \mapsto \mathtt{l})}{\Theta;\Gamma \vdash (\mathtt{nbr}\ a\cdot\mathtt{l}) \to (\mathtt{nbr}\ a\cdot\mathtt{l})\cdot\phi}$$

$$\text{[OP]}\ \frac{}{\Theta;\Gamma \vdash (\mathtt{o}\ \overline{a}\cdot\overline{v}) \to (\mathtt{o}\ \overline{a}\cdot\overline{v})\cdot\mathcal{E}(\mathtt{o},\overline{v})}$$

$$\text{[THEN]}\ \frac{}{\Theta;\Gamma \vdash (\mathtt{if}\ a\cdot t\ a'\cdot\mathtt{l}\ e) \to (\mathtt{if}\ a\cdot t\ a'\cdot\mathtt{l}\ |e|)\cdot\mathtt{l}}$$

$$\text{[ELSE]}\ \frac{}{\Theta;\Gamma \vdash (\mathtt{if}\ a\cdot f\ e\ a'\cdot\mathtt{l}) \to (\mathtt{if}\ a\cdot f\ |e|\ a'\cdot\mathtt{l})\cdot\mathtt{l}}$$

$$\text{[CONG]}\ \frac{\pi_{\mathbb{C}}(\Theta);\Gamma \vdash a \to e}{\Theta;\Gamma \vdash \mathbb{C}[a] \to \mathbb{C}[e]}$$

$$\text{[REP]}\ \frac{\pi_{(\mathtt{rep}^{\mathring{\mathtt{l}}}\ \mathtt{x}\ \mathtt{w}\ [\,])}(\Theta);\Gamma,(\mathtt{x} := (\Gamma(\mathtt{w}) \rhd \mathring{\mathtt{l}})) \vdash a \to a'\cdot\mathring{v}}{\Theta;\Gamma \vdash (\mathtt{rep}^{\mathring{\mathtt{l}}}\ \mathtt{x}\ \mathtt{w}\ a) \to (\mathtt{rep}^{\mathring{\mathtt{l}}\rhd\mathring{v}}\ \mathtt{x}\ \mathtt{w}\ a'\cdot\mathring{v})\cdot\mathring{v}}$$

$$\text{[FUN]}\ \frac{\pi_{(\mathtt{f}^{[\,]}\ \overline{a}\cdot\overline{v})}(\Theta);(args(\mathtt{f}) := \overline{v}) \vdash (body(\mathtt{f}) \rhd \mathtt{s}) \to a\cdot\mathring{v}}{\Theta;\Gamma \vdash (\mathtt{f}^{\mathtt{s}}\ \overline{a}\cdot\overline{v}) \to (\mathtt{f}^{a}\ \overline{a}\cdot\overline{v})\cdot\mathring{v}}$$

Fig. 2. Device Semantics

Runtime Expression Syntax. A runtime expression is the evaluation tree created out of a surface expression. It is similar to expressions in the surface syntax (cf. Figure 1) with the following differences (see Figure 2): *(i)* a (runtime) value v is either a local value $\mathtt{l}$ or a field value ϕ; *(ii)* a run-time expression e can be coupled (at any level of depth) with optional *annotation* $\mathring{v}$ representing the transient side-effect of a computation; *(iii)* constructs `rep` and function calls can have a *superscript* ($\mathtt{s}$) representing the durable side-effect of a computation. Note that, syntactically, surface syntax expressions can (and will) be used to denote runtime expressions with null decorations in all annotations and superscripts.

The erasure operator $|\cdot|$ turns a runtime expression e (or an auxiliary rte a) to the surface expression $|e|$ (resp. $|a|$) obtained by dropping all annotations and superscripts. The erasure of an expression e (or a) is defined if and only if for every auxiliary rte a' occurring in e (resp. a): (i) $a' = (\texttt{nbr}\, a''\cdot v)$ implies that the runtime value v is a local value, and (ii) $a' = (\texttt{f}^{a''}\, \overline{e})$ implies that $|a''|$ is the body of the the function `f`.

Note that fields are actually mappings, for which we introduce some syntactic conventions and operators. A field value ϕ can either be written as $\sigma_1 \mapsto \texttt{l}_1, \ldots, \sigma_n \mapsto \texttt{l}_n$ or be shortened by notation $\overline{\sigma} \mapsto \overline{\texttt{l}}$. The domain of ϕ, which is the set $\{\sigma_1, ..., \sigma_n\}$, is denoted by $\mathbf{dom}(\phi)$. The value $\texttt{l}_i$ associated to a given device σ_i by field ϕ is retrieved by notation $\phi(\sigma_i)$. Since a field can be seen as a list, we use the notation $\bullet$ for the empty field, and comma as list concatenation operator: e.g. ϕ, ϕ' is the field having both the mappings of ϕ and ϕ'. We shall sometime restrict the domain of a field ϕ to a given set of devices $\overline{\sigma}$, which we denote as $\phi|_{\overline{\sigma}}$. When restriction is applied to local values it works as the identity function. A tree environment, Θ, maps devices to runtime expressions (namely, it keeps neighbour trees), and a variable environment, Γ, maps variables to runtime values. Since tree environments and variable environments are also mappings, all the above conventions and operators will be used for them as well.

To take into account special constants, mathematical operations, usual abstract data types operations, and context-dependent operators, we introduce a special function ε. This is such that $\varepsilon(\texttt{o}, \overline{v})$ computes the result of applying built-in operator `o` to values $\overline{v}$. In particular, we assume constant `self` gets evaluated to the current device identifier. In order not to escape the domain restricted by operator `if`, as discussed in Sections 2 and 3, for each primitive operator `o` we assume that: (i) $\varepsilon(\texttt{o}, v_1, \cdots, v_n)$ is defined (i.e., its evaluation does not get stuck) only if all the field values in $v_1, \ldots, v_n$ have the same domain; and (ii) if $\varepsilon(\texttt{o}, v_1, \cdots, v_n)$ returns a field value ϕ and there is at least one field value v_i in $v_1, \ldots, v_n$, then $\mathbf{dom}(\phi) = \mathbf{dom}(v_i)$.

Congruence Contexts and Alignment Contexts. The operational semantics uses *congruence contexts*, ranged over by $\mathbb{C}$, to impose an order of evaluation of subexpressions in an orthogonal way with respect to the actual semantic rules; and it uses *alignment contexts*, ranged over by $\mathbb{A}$, to properly navigate into evaluation trees. In particular, note that $\mathbb{C}$ is a subcase of $\mathbb{A}$ (see Figure 2).

A context $\mathbb{A}$ is an auxiliary runtime expression with a *hole* $[]$. As usual, we write $\mathbb{A}[e]$ to denote runtime expression obtained by filling the hole of $\mathbb{A}$ with the runtime expression e. If a given runtime expression e matches $\mathbb{C}[e']$, then e' is the next subexpression of e where evaluation will occur, positioned in e as described by the position of $[]$ in $\mathbb{C}$. The way the syntax of congruence contexts $\mathbb{C}$ is structured constraints the operational semantics to evaluate the first argument of `if` and then, depending on its outcome, the second or third, and to non-deterministically evaluate arguments in function and operation calls. For instance, the runtime expression `(* 1·1 (+ 2·2 3))` matches $\mathbb{C}'[e']$ only by $\mathbb{C}' =$ `(* 1·1 [])` and $e' =$ `(+ 2·2 3)`: this means that e' contains the next subexpression to evaluate. The expression e', in turn, matches $\mathbb{C}''[e'']$ only by $\mathbb{C}'' =$ `(+ 2·2 [])` and $e'' =$ `3`. Therefore 3 is the next subexpression to evaluate (becoming $3 \cdot 3$).

Auxiliary Functions. The projection operator π implements the mechanism for synchronising navigation of an evaluation tree with those of neighbour trees. Namely, $\pi_{\mathbb{A}}(\Theta)$ takes a tree environment Θ and extracts a new tree environment obtained by discarding the trees that do not match the alignment context $\mathbb{A}$ (according to the *alignment context matching relation* "::") and extracting the corresponding subtree matching the hole in the remaining ones. As an example, given $\Theta_0 = (\ \sigma_1 \mapsto (\texttt{if}\ a\, t\ e_1\ e_2)\cdot v_1,\ \sigma_2 \mapsto (\texttt{if}\ a'\, f\ e_3\ e_4)\cdot v_2\ )$ and $\mathbb{A} = (\texttt{if}\ a'\, t\ [\,]\ e_2')$, we have $\pi_{\mathbb{A}}(\Theta_0) = (\sigma_1 \mapsto e_1)$. In fact, the evaluation tree for σ_2 is discarded since it does not match $\mathbb{A}$ due to the label of first argument being f, while the evaluation tree for σ_1 matches and extracts e_1.

The replacement operator $\triangleright$ is introduced that retains the right-hand side if this is not empty, otherwise it takes the left-hand side. It is useful to handily update null decorations.

Reduction Rules. Following [10], we formulate the reduction relation by means of reduction rules (which may be applied at any point in an expression) and congruence rules (which express the fact that if $e \to e'$ then $(\texttt{o}\ e_1 \ldots e_{i-1}\ e\ e_{i+1} \ldots e_n) \to (\texttt{o}\ e_1 \ldots e_{i-1}\ e'\ e_{i+1} \ldots e_n)$, and so on). The reduction relation is of the form $\boxed{\Theta; \Gamma \vdash e \to e'}$, to be read "expression e reduces to expression e' in one step", where Θ is the current tree environment and Γ is the current store of variables (which is built incrementally in each reduction step by the congruence rules [REP] and [FUN] when evaluation enters the third argument of a `rep`-expressions or the body of a function, respectively).

The reduction relation models the execution of a single computation round, computed as $\Theta; \bullet \vdash a \to^* a' \cdot v$ where: Θ is the set of evaluation trees produced by neighbours at their prior computation round; the variable environment is empty (the main expression must not contain free variables); and a is the runtime expression resulting from the computation of previous round with all the annotations (not superscripts) erased—at very first round a is simply the top-level surface expression. During computation steps the run-time expression will be decorated with annotations, until one appears at top level in the final runtime expression $a' \cdot v$, where v represents the local value of the computational field currently computed. Also some superscripts will be present at the end of the round, for they represent the side-effect of computation on the evaluation tree that should be transferred to next round. In particular, as already mentioned: *(i)* the final runtime expression $a' \cdot v$ will be shipped to neighbours replacing there the one previously sent (and being dropped only when the current device exists the neighbourhood); *(ii)* the runtime expression obtained from $a' \cdot v$ by dropping all annotations (not superscripts), denoted by $init(a' \cdot v)$, will be used as starting point for the next round computation.

We now describe each reduction rule in turn. Computation rules have a common pattern: they compute a result value v, which appears as top-level annotation—in the following we shall say that v is the "local result". Rule [VAL] simply identically annotates a value. Rule [VAR] looks at the value $\Gamma(\texttt{x})$ associated to `x` by the variable environment, and (in case it is a field) restricts it to the set of currently aligned neighbours $\overline{\sigma}$ (plus the local device $\varepsilon(\texttt{self})$). Rule [NBR] is the one actually exploiting Θ: let `l` be the value locally computed, we extract the corresponding values $\overline{\texttt{l}}$ from aligned neighbours $\overline{\sigma}$, and use as local result the corresponding field $\overline{\sigma} \mapsto \overline{\texttt{l}}$ (adding the local slot $\texttt{self} \mapsto \texttt{l}$). Rule [OP] computes the result of applying operator `o` to values $\overline{v}$ (done by function ε,

which gives semantics to operators), to be used as local result. Rules [THEN] and [ELSE] handle condition branching: rule [THEN] (resp. [ELSE]) uses the label of second (resp. third) argument as local result in case of positive (resp. negative) condition, and erases the other branch (which may contain superscripts generated in the previous round).

Rule [CONG] can be understood as a compact representation for six different congruence rules, corresponding to the 6 cases for the context $\mathbb{C}$. While navigating the evaluation tree inside context $\mathbb{C}$ to identify the next evaluation site a (which should be non-annotated), this rule contemporarily enters the same context into all slots of the tree environment Θ, guaranteeing that the expression to evaluate is kept synchronised with the corresponding trees in Θ. Note that rule [CONG] does not describe the congruence rules for `rep`-expressions and function applications. In fact, the metavariable $\mathbb{C}$ does not range over contexts of the form $(\texttt{rep}^{\mathring{\texttt{l}}}\ \texttt{x}\ \texttt{w}\ [])$ and $(\texttt{f}^{[]}\ \overline{a} \cdot \overline{v})$. The rational for this choice is that the corresponding rules, [REP] and [FUN], need to update the variable environment Γ by adding to Γ the `rep`-bound variable `x` or by completely replacing Γ with the environment for the function formal parameters $args(\texttt{f})$, respectively. Moreover, [REP] and [FUN] are not pure congruence rules: each of them encodes a congruence rule possibly followed by a computation rule. Note that this encoding exploits the notation $\mathring{y}$ and the auxiliary function $\triangleright$ defined above.

Rule [REP] handles evolution of a field. When the superscript $\mathring{\texttt{l}}$ is null, the evaluation of the body of `rep`-expression is carried on in an environment that assigns to the `rep`-bound variable `x` the value of the variable or local value `w`—with abuse of notation we indicate it as $\Gamma(\texttt{w})$: when `w` is a local value `l` we assume $\Gamma(\texttt{l}) = \texttt{l}$. When the superscript $\mathring{\texttt{l}}$ is a local value `l`, the evaluation of the body of `rep`-expression is carried on in an environment that assigns to the `rep`-bound variable `x` the value `l`. If the reduction step performed (in the premise of the rule) on the body of the `rep`-expression produces an evaluated runtime expression (i.e., if the annotation $\mathring{v}$ is not null), then the local result is propagated to the `rep`-expression (which becomes evaluated).

When the actual parameters of a function call are evaluated, rule [FUN] performs a reduction step on the function body in an environment consisting of the proper association of formal parameters $args(\texttt{f})$ to values $\overline{v}$: the (possibly null) resulting annotation $\mathring{v}$ is transferred as local result. If the superscript `s` is null, replacement operator $\triangleright$ guarantees the function body is used instead.

5 Properties

A key property to pave the way towards advanced forms of behavioural analysis is the following soundness. We say that the operational semantics of the field calculus is *sound* to mean that the execution of a *well-formed* surface program satisfies the following two properties:

P1. The reduction does not get stuck.
P2. The domain of every field value arising during the reduction consists of the identifiers of the aligned neighbours and of the identifier of the `self` device.

While the former follows from the standard type soundness argument, the latter is needed to guarantee a proper handling of restriction. Of course, it is key to find a

definition of well-formedness for expressions that filters out those expressions which would eventually lead to either P1 or P2 failing to hold, without restring the expressive power of the language.

Let us illustrate how well-formedness should work with some counter-examples, all connected to the novel issues of field values rather than just the more typical elements shared with many other calculi. Any program containing a non-well-formed function or expression is non-well-formed. An example of a non-well-formed function is `(def wrong-distance-to (x) (distance-to (nbr x)))`, using the function `distance-to` defined in Section 2. In this example, the field value ϕ, which is produced by `(nbr x)` and passed into `distance-to`, conflicts with its use as the first input to `mux`, which requires a local value for ε. Rule [OP] thus cannot be applied, and the evaluation cannot be completed.

Another example is the function `(def wrong-f-two (x) (min-hood (min-hood (nbr (nbr x)))))`, which tries to find the minimum value of `x` within two hops. This fails to evaluate because Rule [NBR] requires its input to be a local value, and thus cannot be applied to the outer `nbr`. This prevents the need to communicate a field value whose size scales linearly with the number of neighbours, which might be extremely burdensome. A well-formed alternative that produces the same computational result as `wrong-f-two` is intended to is `(def right-f-two (x) (min-hood (nbr (min-hood (nbr x)))))`. This takes advantage of the commutative property of minimisation to break the minimisation into two stages, thus avoiding the communication explosion of the not well-formed formulation.

A final example is the function `(def wrong-nbr-if (x y z) (min-hood (- (if (sense 1) (nbr x) (nbr y)) (nbr z)))`. This will fail to evaluate on Rules [THEN] and [ELSE], since they require local values for the test and returned values. This prevents conflicts between field domains, as in this case, where the field produced by `(nbr z)` would contain all neighbours, while the field produced by the `if` expression would contain only a subset, leaving the fields mismatched in domain at the subtraction. A correct alternative is `(def right-nbr-if (x y z) (min-hood (- (nbr (if (sense 1) x y)) (nbr z)))`, which conducts the test locally, ensuring that the domains of the two fields match.

We argue that these sorts of well-formedness problems are detectable as type errors through static analysis, without having to evaluate the program in a full context. We are currently working at a formalisation of the notion of well-formed surface program by means of a simple type system designed to support the formal statement for properties P1 and P2.

6 Conclusion, Related and Future Work

A number of works present notions of computational fields; a thorough review may be found in [5]. Regarding the most similar: the Hood sensor network abstraction [20] and Butera's "paintable computing" hardware model [7] implement computational fields using only the local view, and thus do not ensure well-formed domains. The $\sigma\tau$-Linda model [19] proposes an extension of Linda with few constructs for spreading tuples to form fields, and adopting a notion of computation rounds very similar to the

one we formalised. More generally, while all of the key ingredients for programming computational fields are supported in a number of different languages (see [5]), at present only Proto supports all five that we found critical to include in the calculus.

A number of other formal calculi have also been developed for parallel computations in structured environments, like 3π-calculus [8], Ambient calculus [9], and P-systems [14]: they all describe parallel computation over variously abstracted notions of space; differently from our calculus they do not focus on raising the level of abstraction beyond local interaction rules and up to aggregate-level descriptions.

A core operational semantics for discrete execution of Proto programs was developed in [15]. Although closely related to the present one, it was a preliminary attempt extremely limited in the types of computations it could represent, since it did not tackle the fully general problem of combining restriction, evolution, and recursive function calls (i.e. dynamically expanding evaluation trees), which we have addressed through the idea of aligning annotated evaluation trees. Based on [15], in [16] a full formalisation of discrete Proto was provided. This resulted in a rather large semantics aimed at a faithful representation of every construct in Proto and of their execution by the platform—e.g., including an intricate technique for optimising message size. The resulting model is then too complicated to readily use in proving language properties. In contrast, the operational semantics of the calculus presented in this paper is general enough to cover all of Proto and many other spatial languages [5], and is compact enough to be suitable as a basis for tackling interesting properties.

In particular, we believe that equipping the calculus with a sound static type system can bootstrap investigations on other important properties. For example, the work in [6] develops a precise model of spatial computing covering the same key mechanisms considered in this paper, but for fields over *continuous space-time* rather than discrete device executions. In future works, we mean to prove that there is a broad class of cases where our model converges to the continuous one in the limit, as the density of devices increases and the length of time steps decreases. This would allow characterisation of those programs that have a predictable conformation to the aggregate-level behavior independently on the topology (density) and on the timing of devices. Another interesting thread concerns finding a characterisation of expressiveness of spatial computing languages [1], with clear implications in the design of new mechanisms.

This calculus should thus serve as an important step toward identifying an engineering methodology for developing spatial computing and coordination systems able to make use of complex yet predictably well-behaved self-organising mechanisms, both in today's and in emergent distributed computing scenarios.

Acknowledgements. We thank the anonymous FOCLASA referees for comments and suggestions for improving the presentation.

References

1. Beal, J.: A basis set of operators for space-time computations. In: Spatial Computing Workshop (2010), `http://www.spatial-computing.org/scw10/`
2. Beal, J.: Engineered self-organization approaches to adaptive design. In: Roy, R., Shehab, E., Hockley, C., Khan, S. (eds.) 1st International Conference on Through-life Engineering Services, pp. 35–42. Cranfield University Press (November 2012)

3. Beal, J., Bachrach, J.: Infrastructure for engineered emergence in sensor/actuator networks. IEEE Intelligent Systems 21, 10–19 (2006)
4. Beal, J., Bachrach, J., Vickery, D., Tobenkin, M.: Fast self-healing gradients. In: Proceedings of ACM SAC 2008, pp. 1969–1975. ACM (2008)
5. Beal, J., Dulman, S., Usbeck, K., Viroli, M., Correll, N.: Organizing the aggregate: Languages for spatial computing. In: Mernik, M. (ed.) Formal and Practical Aspects of Domain-Specific Languages: Recent Developments, ch. 16, pp. 436–501. IGI Global (2013), A longer version available at: `http://arxiv.org/abs/1202.5509`
6. Beal, J., Usbeck, K., Benyo, B.: On the evaluation of space-time functions. The Computer Journal (2012), Online first, available through doi:10.1093/comjnl/bxs099
7. Butera, W.: Programming a Paintable Computer. PhD thesis, MIT, Cambridge, MA, USA (2002)
8. Cardelli, L., Gardner, P.: Processes in space. In: Ferreira, F., Löwe, B., Mayordomo, E., Mendes Gomes, L. (eds.) CiE 2010. LNCS, vol. 6158, pp. 78–87. Springer, Heidelberg (2010)
9. Cardelli, L., Gordon, A.D.: Mobile ambients. Theoretical Computer Science 240(1), 177–213 (2000)
10. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A minimal core calculus for Java and GJ. ACM Transactions on Programming Languages and Systems 23(3) (2001)
11. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications: The tota approach. ACM Trans. on Software Engineering Methodologies 18(4), 1–56 (2009)
12. MIT Proto, `http://proto.bbn.com` (retrieved January 1, 2012)
13. Montagna, S., Viroli, M., Fernandez-Marquez, J.L., Di Marzo Serugendo, G., Zambonelli, F.: Injecting self-organisation into pervasive service ecosystems. Mobile Networks and Applications 18(3), 398–412 (2013)
14. Paun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000)
15. Viroli, M., Beal, J., Casadei, M.: Core operational semantics of Proto. In: Proceedings of ACM SAC 2011, pp. 1325–1332. ACM (March 2011)
16. Viroli, M., Beal, J., Usbeck, K.: Operational semantics of proto. Science of Computer Programming 78(6), 633–656 (2013)
17. Viroli, M., Casadei, M., Montagna, S., Zambonelli, F.: Spatial coordination of pervasive services through chemical-inspired tuple spaces. ACM Transactions on Autonomous and Adaptive Systems 14, 14:1–14:24 (2011)
18. Viroli, M., Casadei, M., Omicini, A.: A framework for modelling and implementing self-organising coordination. In: Proceedings of ACM SAC 2009, vol. III, pp. 1353–1360, March 8-12. ACM (2009)
19. Viroli, M., Pianini, D., Beal, J.: Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In: Sirjani, M. (ed.) COORDINATION 2012. LNCS, vol. 7274, pp. 212–229. Springer, Heidelberg (2012)
20. Whitehouse, K., Sharp, C., Brewer, E., Culler, D.: Hood: a neighborhood abstraction for sensor networks. In: Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services. ACM Press (2004)