

Core Operational Semantics of Proto

Mirko Viroli
Alma Mater Studiorum –
Università di Bologna
Via Venezia 52, 47521
Cesena (FC), Italy
mirko.viroli@unibo.it

Jacob Beal
BBN Technologies
10 Moulton Street, Cambridge,
MA 02138, USA
jakebeal@bbn.com

Matteo Casadei
Alma Mater Studiorum –
Università di Bologna
Via Venezia 52, 47521
Cesena (FC), Italy
m.casadei@unibo.it

ABSTRACT

The Proto spatial computing language [6] simplifies the creation of scalable, robust, distributed programs by abstracting a network of locally communicating devices as a continuous geometric manifold. However, Proto’s successful application in a number of domains is becoming a challenge to its coherence across different platforms and distributions. We thus present an operational semantics for a core subset of the Proto language. This semantics covers all the key operations of the three space-time operator families unique to Proto—restriction, feedback, and neighborhood—as well as a few of the pointwise operations that it shares with most other languages. Because Proto programs are distributed, we also present an operational semantics for their asynchronous execution across a network. This formalization will provide a reference to aid implementers in preserving language coherence across platforms, domains, and distributions.

1. INTRODUCTION

As the scale and variety of deployed and emerging distributed systems continues to increase, the challenges of creating scalable, robust aggregate behavior are increasing. One large class of these systems are *spatial computers*—potentially large collections of devices distributed to fill some space, such that the difficulty of moving information between devices is strongly dependent on the distance separating them. Example of spatial computers include sensor networks, peer-to-peer wireless, colonies of engineered bacteria, robotic swarms, and pervasive computing systems. Aggregate-level programming and control of such systems is an important and unresolved challenge. Many different approaches have been proposed, including distributed logic programming [18, 1], viral tuple-passing [13], abstract graph algorithms [11], spatial data streaming [16], and topological surgery [10], none of which has yet proven successful enough to be adopted into widespread use.

One promising approach to the programming and control of such systems is to view the network of devices as a discrete approximation of the space through which they

are distributed. The Proto spatial computing language [6] embraces this approach: a program is specified in terms of geometric computations and information flow on a continuous manifold. These aggregate-level programs are automatically transformed into a set of local interactions between discrete devices, which approximate the desired global behavior. This approach has advantages for scalability, since more devices are simply a better approximation of the continuous model, for portability, since changing to a different platform just means changing how the continuous model is approximated, and for robustness, since small changes are just changes in approximation quality and large changes appear as changes in manifold structure that geometric computations inherently adapt to. Moreover, Proto has already been successfully applied to problems in such diverse areas as sensor networks [2], swarm and modular robotics [4], and synthetic biology [7, 9].

The breadth, however, is becoming a challenge to the coherence of Proto, as different platforms have very different demands and execution characteristics, and the number of platforms to which Proto is being applied is steadily increasing. It would thus be easy for different versions of Proto to start diverging in implementation and effective semantics, particularly since the Proto tool-chain is distributed as free and open source software.

We thus believe it is imperative to establish a formal semantics for Proto, and in this paper we present an operational semantics for a core subset of the Proto language. By isolating and stabilizing the key concepts of the language, this formalization is the first step toward a reference for implementers, helping to preserve the coherence of the language across the increasing number of platforms, application domains, and distributions.

2. THE PROTO LANGUAGE

The Proto language approaches the challenges of spatial computing by wholeheartedly embracing continuous space and time. The network of devices to be programmed is viewed as a discrete approximation of the continuous space that they occupy, using the *amorphous medium* abstraction [5]. An amorphous medium [6] is a manifold with a computational device at every point, where every device knows the recent past state of all other devices in its neighborhood (Figure 1). While an amorphous medium cannot, of course, be constructed, it can be approximated on the discrete network of a spatial computer.

Proto [6], uses the amorphous medium abstraction to factor programming a spatial computer into three loosely coupled sub-problems: global descriptions of programs,

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

SAC’11 March 21-25, 2011, TaiChung, Taiwan.

Copyright 2010 ACM 978-1-4503-0113-8/11/03 ...\$10.00.

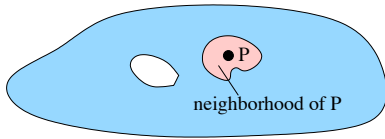


Figure 1: An amorphous medium is a manifold where every point is a device that knows its neighbors’ recent past state.

compilation from global to local execution on an amorphous medium, and discrete approximation of an amorphous medium by a real network.

Proto is a functional language that is interpreted to produce a dataflow graph of operations on fields, where some of those operations may be functions implemented with their own dataflow graph on fields. This program is then evaluated against a manifold to produce a field with values that evolve over time. Proto uses four families of operations: point-wise operations like `+` that involve neither space nor time, restriction operations like `if` that limit execution to a subspace, feedback operations like `letfed` that establish state and evolve it in continuous time, and neighborhood operations that compute over neighbor state (gathered with `nbr`) and space-time measures like `nbr-range`, then summarize the computed values in the neighborhood with a set operation like `integral` (`int-hood`) or `minimum` (`min-hood`).

With Proto’s carefully chosen set of operators, compilation and discrete approximation are straightforward, because each primitive is chosen to have a continuous-space specification that can be coherently approximated with local device actions, and their composition rules preserve this property (except in extreme circumstances). Thus, Proto makes it easy for a programmer to carry out complicated spatial computations using simple geometric programs that are robust to changes in the network and that self-scale to networks with different shape, diameter, density of devices, and execution and communication properties [3]. We do not attempt to present the full details or motivation of Proto in this paper: its documentation and free software implementation are available online, with the primary distribution site currently at <http://proto.bbn.com> [15]

2.1 Example and Core Semantics Subset

In this paper, we formalize only a minimal core set of Proto semantics. A full formalization would be extremely lengthy, with most of that length devoted to the routine portions of the language (pointwise operations identical to the basic math, logic, and type operations of most language) and to minor variants of space-time operations. Instead, we have identified a minimal core of the language such that, once operational semantics are identified for this minimal core of Proto, it will be possible to generate the rest of the semantics can be generated by using these operations as templates for other closely related operations. For example, we will formalize the neighborhood operator `nbr-range`; the operators `nbr-lag`, `nbr-vec`, and `nbr-angle` are all identical except instead of collecting the ranges to neighbors, these operators collect other space/time displacement metrics.

For this core, we thus need to select representative operations from the four families of space-time operations. There are many possible reasonable choices of such representative

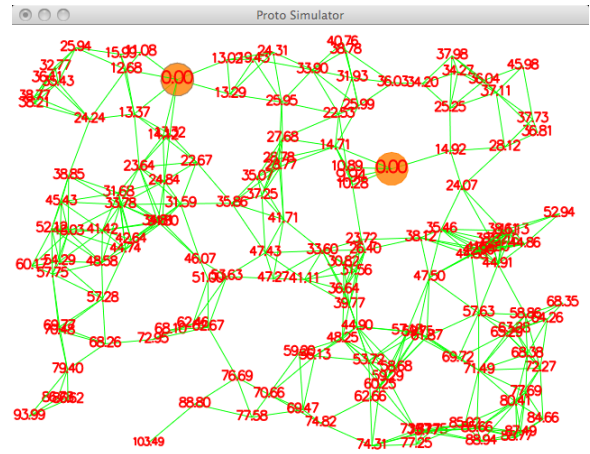


Figure 2: Example of (`distance-to` (`sense 1`)) executing on a network of 150 simulated devices. Each device is displaying its estimated distance to the nearest source device (orange circles).

operations. We are thus grounding our choice in the minimal operations necessary to implement one of the most common and important computations performed in Proto: computing the shortest distance from every devices to a set of source devices. Proto code for a simple version of `distance-to` is:

```
(def distance-to (source)
  (rep d
    inf
    (mux source
      0
      (fold-hood* min
        inf
        (+ (nbr d) (nbr-range))))))
```

where the subfunction `min` is defined as

```
(def min (a b) (if (< a b) a b))
```

This code uses a nice sampler of operations to compute shortest-path distance to devices in the `source` through a triangle-inequality relaxation. Every device maintains a feedback state variable `d` for its current distance estimate. This starts at infinity (`inf`) and then is relaxed each round by the computation beginning with `mux`. The `mux` is a point-wise “multiplexing” branch: each device tests whether it is one of the `source` devices. If it is, its distance to the nearest `source` devices is obviously zero. Otherwise, it minimizes over estimates of distance to the source through its neighbors, summing the `(nbr d)` distance estimates of neighbors with the `(nbr-range)` distance to each neighbor.

The code listed here is a simpler version of the usual `distance-to` algorithm [8]: the more complex algorithm offers fast self-stabilization in response to changes in the network or source set. A typical execution, (`distance-to` (`sense 1`)), is shown in Figure 2. In this case, the source is the set of devices where a boolean-valued sensor has been triggered (the `sense` operator selects from a set of user-actuated testing sensors).

Grounding our semantics work in this simple version of a foundational computation has the dual benefits of providing an intuitive check on our work and immediate application

in assuring correct semantics of a key Proto computation. Of the four families, only restriction is missing, so we use an `if` in the definition of `min` where a `mux` would be equally appropriate. We thus select the following representative core of Proto operators for formalization:

- Pointwise: numeric constants, numeric comparison, four-function arithmetic, function definition, pointwise branch (`mux`), generic sensor (`sense`)
- Restriction: `if`
- Feedback: `rep`
- Neighborhood: `nbr`, `nbr-range`, `fold-hood*`, arithmetic on neighborhoods

A few notes are worth explaining regarding our choices. We have not included any purely syntactic operations like `let` or macros, since these are typical of many languages. Likewise, we have not included time-metrics like `dt` and `nbr-lag`, despite their importance, as their operational semantics will be nearly identical to either sensors or `nbr-range`. We have also chosen the discrete neighborhood summary operator `fold-hood*` rather than a continuous one like `min-hood` because the *executed* version of every continuous neighborhood summary operator can be implemented with `fold-hood*` (even if the theoretical abstraction version cannot) and the reverse is not true.

2.2 Functional vs. Operational Semantics

Proto’s functional nature allows for an elegant specification. On each device, each function application returns either a value or a field, which is a mapping from a neighborhood (i.e., certain nearby devices) to values. For instance, `nbr` takes an expression and yields a field mapping each neighbor to the result of computing the expression there.

On the other hand, as far as guiding the implementation of coherent platforms for supporting Proto, it is key to provide a formalization shedding light on what step-by-step behaviour occurs in each device, and what messages are actually exchanged in order to support the management of those fields. For instance, the underlying behaviour of `nbr` in a device a is such that: (i) the expression is evaluated in a and its results are broadcast in the neighborhood, and (ii) the last version of the messages received at a for that specific `nbr` instruction is considered, a field structure is created, and is accordingly returned as output of `nbr`.

The goal of the operational semantics presented here is to provide a solid ground to describe how all the functions and operations work together to create those spatial structures out of local computations and exchange of messages. We here point out the assumptions about connection between our formalization and the space-time abstraction of Proto.

Concerning space, we model the network of devices as a graph with unidirectional links representing the ability of devices to communicate. Each link also carries an estimation of the distance between the two devices, used by `nbr-range` to construct a field of distances to neighbors. The position of devices can even change over time, which we model in terms of changes to such graph.

Concerning time, we assume asynchronous execution of devices: we simply assume that in each device, after a computation round is over, the next one at some point starts. This highly flexible model is the one specified for Proto,

though as a practical matter, most Proto platforms assign to each device an execution frequency f (typically but not necessarily equal in all devices), such that each $1/f$ time units the device executes one “computation round,” executing its program and sending the necessary messages.

3. LANGUAGE SEMANTICS

We now provide the operational semantics of our chosen core fragment of the Proto language/platform. Our specification is divided in two parts, described in this section and in the next one. The first part concerns internal device behaviour, namely, it gives a semantics to the language issues of Proto, describing how the execution of Proto programs locally affects a single device. The second part concerns platform issues of Proto, namely, how messages are exchanged between devices and how topology aspects are dealt with.

Our formalization necessarily combines techniques typical of functional languages (since Proto is fundamentally a LISP dialect) and process algebras (since it is used to program message exchanges in a distributed system, like in π -calculus [14])—see e.g. [17] for other usages of such a combined approach).

3.1 Proto Language Syntax

We adopt some syntactic notation and terminology of standard and widely accepted frameworks like FJ [12]. We refer to *metavariables* as variables used in the formalization, as schema of formulas or terms—to distinguish between variables belonging to the Proto language. For the Proto surface syntax, we let metavariable n range over numbers (integer and real), x over variables, and F over newly defined function names. For syntax of internals, we let a range over device identifiers, and i over indexes (natural numbers) also called *slots*. Given any metavariable x , metavariable \bar{x} is used to range over sequences of elements of kind x , and we let x_1, \dots, x_n be its elements. An empty sequence is denoted “•”. The syntax of the Proto fragment we consider in this paper is as follows:

$e ::= v \mid x \mid f \mid (f \bar{e})$	Expressions
$v ::= n \mid \phi$	Values (num. or fields)
$f ::=$	Functions
o	Math operators
$sense$	Sensing
rep	Stateful iteration
$if^{\bar{x};\bar{x}'} \mid mux$	Selection
$nbr^i \mid nbr-range^i$	Field creation
$fold-hood*$	Neighbourhood folding
F	Defined function
$o ::= + \mid - \mid * \mid / \mid > \mid > \mid = \mid ! =$	Math operators
$d ::= (def F (\bar{x}) e)$	Function definition

The essence of a Proto program is a functional expression e , which could be a value v , a variable x (e.g., the argument of a function for which this expression is the body), a function f , or the application $(f \bar{e})$ of function f to the possibly empty sequence of expressions \bar{e} . A run-time value is either a number n , or a field ϕ (namely, a mapping from neighbour devices to numbers). Note that a field cannot be expressed directly by the programmer as part of the Proto syntax: however, it is added here for it can be produced and run-time by the operational semantics, namely, by the `nbr` construct.

A function can be: an arithmetic operator \circ ; the sensing function **sense** by which a device can inspect a binary environmental signal; the stateful iteration function **rep** by which a variable can be created, initialized and later modified; the selection functions **if** and **mux**; the **nbr** function to create a field with values computed from any neighbours; the **nbr-range** function creating a field of distances from neighbours; the **fold-hood*** function computing a number out of a field; and finally any function F we give semantics by a definition d (specifying arguments \bar{x} and body e). A Proto program is hence an expression e_p using a set \mathcal{D} of definitions, which in the following we both assume as given and fixed.

In the syntax above, slot i of a **nbr** or **nbr-range** function is considered as annotations automatically generated and added by the Proto compiler to uniquely identify the occurrence of that construct in the program: such generated slots are positive natural numbers, while slot 0 is internally used to store proximity information, as will be shown below. Similarly, annotation $\bar{x}; \bar{x}'$ in **if** construct is created by the compiler: it gathers the variables created inside the two branches of the **if**— \bar{x} and \bar{x}' in each branch, respectively.

3.2 Configuration Syntax

To complete the description of syntactical aspects of our model, we now take into account the overall state of computation inside a device, including aspects like the assignment of variables, the store of messages to/from neighbours, and the situation of signals to be sensed. The actual (internal) syntax we use is as follows.

s	::=	$\langle \Sigma \rangle e$	Device state
μ	::=	$i \mapsto n$	Message content
Σ	::=		Store
		0	Empty store
		$(\Sigma \mid \Sigma)$	Store composition
		$sns(n)$	Activated sensor
		$x := n$	Variable
		$a : \mu$	Incoming message
		μ	Outgoing message
ϕ	::=	$0 \mid a \mapsto n \mid (\phi \mid \phi)$	Field

The state of a device is a couple $\langle \Sigma \rangle e$, where e is the expression we are evaluating and Σ is a memory for side-effects, called a *store*. The content of a message μ is a numeric value n associated to a slot. A store is basically a (possibly empty—i.e. 0) composition, by operator “|”, of four kinds of terms: $sns(n)$ means sensor n is active, $x := n$ means variable x has been created and assigned to value n , $a : \mu$ is the last message received from a for a given slot, and finally $\mu = i \mapsto n$ is an outgoing message prepared after the evaluation of **nbr** ^{i} gave value n , or after **nbr-range** ^{i} has been evaluated (in which case n is an unused field). As in process algebras, such a composition is a multiset, namely operator “|” is always assumed to be associative, commutative, and absorbing 0. We also define a field as a composition of terms $a \mapsto n$, mapping device a to value n .

Some further ad-hoc notation is added to simplify the management of the above syntactic structures. Given a structure k , we write $\{k\}$ for a composition of elements of kind k by operator “|”: additionally, if in $\{k\}$ some metavariables \bar{x} is used where an x is expected, this means that each x_j is to be used in the corresponding k_j ($j \geq 0$), orderly. For example, let \bar{a} be the sequence $a_0 a_1 a_2$, and \bar{n} the sequence

1 2 3; then $\{\bar{a} \mapsto \bar{n}\}$ is the field $a_0 \mapsto 1 \mid a_1 \mapsto 2 \mid a_2 \mapsto 3$.

We also add an auxiliary composition operator denoted “ \otimes ”: this acts as a variant of operator “|” guaranteeing that the multisets on left and right side have no elements in common—“ \otimes ” is also given lower priority than “|”. Namely, “ \otimes ” is a partial binary function yielding no result if the two multisets given as arguments have non empty intersection, and yielding their composition by “|” otherwise. Used in conjunction with notation $\{k\}$ above, this will be used to extract all elements of a multiset matching a given pattern—which will be helpful e.g. to update fields created by **nbr** when new messages arrive. For instance, match

$$\{a_0 : \bar{i} \mapsto \bar{n}\} \otimes \Sigma \equiv (a_0 : i_0 \mapsto 0 \mid a_0 : i_1 \mapsto 1 \mid a_1 : i_0 \mapsto 5)$$

is used to extract, from the store of three elements on right, all those matching $\{a_0 : _ \mapsto _ \}$, leaving all other elements in Σ . The above match provides precisely one result, namely:

$$\bar{i} \equiv i_0 i_1 \quad \bar{n} \equiv 0 1 \quad \Sigma \equiv a_1 : i_0 \mapsto 5$$

On the other hand, match

$$\{a_2 : \bar{i} \mapsto \bar{n}\} \otimes \Sigma \equiv (a_0 : i_1 \mapsto 1 \mid a_1 : i_0 \mapsto 5)$$

provides result

$$\bar{i} \equiv \bullet \quad \bar{n} \equiv \bullet \quad \Sigma \equiv (a_0 : i_1 \mapsto 1 \mid a_1 : i_0 \mapsto 5)$$

for no matching elements have been found.

3.3 Context

Before defining operational semantics, we orthogonally state the order of evaluation of expressions, which we specify by means of a so-called evaluation context [19]. A context E is an expression with one hole \square in it, representing the next place where a sub-expression needs to be evaluated. We let notation $E[e]$ mean context E after substituting \square with expression e . Contexts are syntactically defined as follows.

E	::=	$\square \mid (\circ E e) \mid (\circ v E)$
		$(\text{sense } E) \mid (\text{rep } x E e) \mid (\text{if } \bar{x}; \bar{x}' E e e)$
		$(\text{mux } e E e) \mid (\text{mux } e n E) \mid (\text{mux } E n n)$
		$(\text{fold-hood* } f E e) \mid (\text{fold-hood* } f n E)$
		$(\text{nbr}^i E) \mid (F \bar{n} E \bar{e})$

As an example of usage, consider the two productions of E to $(\circ E e)$ or $(\circ v E)$: this means that the first argument of \circ should be evaluated first, only when it reached a value v then second argument can be evaluated. In fact, the only match of expression $(+ 5 (- 3 2))$ with pattern $E[e]$ is the one unifying E with $(+ 5 \square)$ and e with $(- 3 2)$: this will be used to mean that the next evaluation step is to compute sub-expression $(- 3 2)$.

Accordingly, the syntactic structure of E above prescribes that: the first argument to a function \circ is first evaluated, and when it comes to a value then the second is evaluated; the only argument of **sense** is to be evaluated (similarly for **nbr**); in **rep** only the second argument is evaluated (evaluation of third argument will be executed conditionally by proper rules of the operational semantics); in **if** only the condition is evaluated (again, the two branches will be evaluated conditionally by proper rules of the operational semantics); in **mux** we first evaluate branches and then the condition; the second argument of **fold-hood*** (initial expression) is evaluated before third (accumulation function);

and finally in a defined function F arguments are evaluated from left to right.

3.4 Operational Semantics

Operational semantics is given by a transition system of the kind (S, \rightarrow) , where S is the set of states s of the device of interest, and $\rightarrow \subseteq S \times S$ is the transition relation. As usual, we write $s \rightarrow s'$ as a shorthand for $(s, s') \in \rightarrow$, meaning that the device of interest moves from state s to state s' by an internal computation step. In the model we present here we consider a single execution round for a device, namely:

- As initial state we consider $\langle \Sigma \rangle \mathbf{e}_p$, where Σ is formed by: (i) the set of activated sensors $\{sns(\bar{n})\}$, (ii) the set of incoming messages $\{\bar{a} : \bar{i} \mapsto \bar{n}\}$ arrived so far, and (iii) the state of variables $\{\bar{x} := \bar{n}\}$ already initialized by a **rep** construct.
- The computation then starts on a step-by-step basis, by evaluating the program expression \mathbf{e}_p and possibly affecting/reading the store Σ , until the expression is completely evaluated to a number—the result of the computation round on that device.
- At that point, Σ will also include a set of outgoing messages $\{\bar{i} \mapsto \bar{n}\}$.

We assume that it is the role of the platform to properly manage incoming/outgoing messages, as well as restoring the initial expression at each computation round—this will be formalized in next section.

Rules of the operational semantics are provided in Figure 3. Rule [CTX] handles evaluation contexts, allowing other rules to abstract from evaluation order. It states that in an expression matching $\mathbf{E}[\mathbf{e}]$ we can first let \mathbf{e} evaluate to \mathbf{e}' (in one step), that is, we select the subexpression e to be evaluated and get rid of the external (possibly complex) context \mathbf{E} in which it resides. The overall expression we obtain is $\mathbf{E}[\mathbf{e}']$, where we replaced \mathbf{e}' in the proper context. Of course, this computation can also affect the store Σ .

Rule [SNS] handles the **sense** construct. We first get all active sensors \bar{n} : if they include the searched \mathbf{n} we yield 1, otherwise we yield 0. This construct is used in Proto to intercept and accordingly manage stimuli coming from the external environment over a specific device.

Rule [VAR] simply makes any variable evaluate to the value it is currently assigned to, as read in the store. Rules [REP-*] handle the **rep** construct: e.g. it makes expression (**rep** x 0 (+ x 1)) evaluating to 0 at the first time, and from then to 1,2, and so on. Rule [REP-I] is active if we do not have an assignment for variable x in the store yet: if this is the case we add to the store (and return as output) the assignment to the initial value \mathbf{n} . Rule [REP-E] is active if an assignment exists, in which case we simply let the third argument (the expression updating the variable) proceed one step. Finally, rule [REP-F] executes when the third argument is a value, assigning to the variable and returning.

Rule [IF] handles the **if** construct and is active if the second argument (condition) is already evaluated: if it is 1 we return the second argument, otherwise we return the third one. Additionally, all the state variables in the branch that is not evaluated are re-initialized—namely, removed from the store. This is because of the space-time semantics being implemented by the **if** and **rep** constructs: an **if** restricts

the domain of each of its branches, while a **rep** is a continuous time evolution of state from an initial value. Thus, when a **rep** is contained in the branch not taken at a device, its prior state becomes undefined and it must begin evolution again from a (possibly different) initial value.

Rule [MUX] behaves similarly to [IF], but it executes only after the two branches have been evaluated, and provides for no reinitialization.

Rule [NBR] handles the **nbr** construct. Let i be the slot for the field we want to create. We first access from the store any pertinent $a : i \mapsto \mathbf{n}$ term, and accordingly create a field structure that is returned as output. Additionally, an output message $i \mapsto \mathbf{n}$ is created containing the result of local evaluation of the argument. Construct **nbr-range** is similarly handled by rule [NBR-RNG]. The two differences are: we here prepare an output message with dummy content 0, and the field is created using as domain only those devices $a \in \bar{a}$ for which a message $a : i \mapsto 0$ exists in the store. As distance \mathbf{n}_{dist} of a from current node we use the one extracted from the slot-0 message $a : 0 \mapsto \mathbf{n}_{dist}$ —such a message being automatically put there by the infrastructure.

Rules [FHOOD-*] handle **fold-hood*** construct: e.g. the expression (**fold-hood*** \min \inf ϕ) returning the minimum value in the field ϕ —function **min** computes the minimum of two values, as defined in previous section. The first rule deals with the case that the field (third argument) is not empty: we drop an item $a \mapsto \mathbf{n}$ from it, and substitute \mathbf{n}_0 with the result of applying folding function \mathbf{f} to \mathbf{n}_0 and \mathbf{n} . When the field is empty, the second rule simply returns the second argument \mathbf{n}_0 .

Mathematical operators \mathbf{o} are handled by rule [MATH] which basically propagates the use of arithmetic operator \mathbf{o} to the two arguments: we assume that arguments can be fields as well (the operation is executed pointwise on the value of each field). Finally, rule [DEF] searches for the proper definition of F , and returns the retrieved body after substituting formal parameters \bar{x} with actual ones \bar{v} .

3.5 Example

We now show a brief example based on the (**distance-to** (**sense** 1)) program considered in Figure 2. As initial store for a device with identifier a_2 we consider

$$\mathbf{d} := \mathbf{inf} \mid a_1 : 0 \mapsto 10 \mid a_3 : 0 \mapsto 10 \mid a_1 : 1 \mapsto 10 \mid a_3 : 1 \mapsto \mathbf{inf}$$

meaning that current device has variable \mathbf{d} (current estimation distance of a_2 from the source) set to infinity, there are two neighbors a_1 and a_3 both at distance 10 from a_2 (slot 0), and the last messages received from a_1 and a_3 due to the only **nbr** construct (slot 1) holds value 10 and infinity (current estimation distance of a_1 and a_3 from the source, respectively). The sequence of transitions from initial program (**distance-to** (**sense** 1)) to final result 20 are shown in Figure 5. In particular we note that the transition due to rule [NBR] also causes a side effect on the store: term $1 : \mathbf{inf}$ is added, meaning that the current estimation distance from the source is infinity, which will be broadcast to neighbors. Also, last transition will update the value of variable \mathbf{d} to 20. Hence, the resulting behaviour of a computation round is to (i) prepare an outgoing message with current value of \mathbf{d} , (ii) update \mathbf{d} with the new estimation based on last messages from neighbors, and (iii) give the new value of \mathbf{d} as output.

$\frac{\langle \Sigma \rangle e \rightarrow \langle \Sigma' \rangle e'}{\langle \Sigma \rangle E[e] \rightarrow \langle \Sigma' \rangle E[e']}$	[CTX]
$\frac{(\mathbf{n} \in \bar{\mathbf{n}} \Rightarrow \mathbf{n}_o = 1) \quad \text{or} \quad (\mathbf{n} \notin \bar{\mathbf{n}} \Rightarrow \mathbf{n}_o = 0)}{\langle \{sns(\bar{\mathbf{n}})\} \otimes \Sigma \rangle (\text{sense } \mathbf{n}) \rightarrow \langle \{sns(\bar{\mathbf{n}})\} \otimes \Sigma \rangle \mathbf{n}_o}$	[SNS]
$\frac{-}{\langle \mathbf{x} := \mathbf{n} \mid \Sigma \rangle \mathbf{x} \rightarrow \langle \mathbf{x} := \mathbf{n} \mid \Sigma \rangle \mathbf{n}}$	[VAR]
$\frac{-}{\langle \{\mathbf{x} := \bullet\} \otimes \Sigma \rangle (\text{rep } \mathbf{x} \ \mathbf{n} \ \mathbf{e}) \rightarrow \langle \mathbf{x} := \mathbf{n} \mid \Sigma \rangle \mathbf{n}}$	[REP-I]
$\frac{\langle \{\mathbf{x} := \mathbf{n}\} \otimes \Sigma \rangle \mathbf{e} \rightarrow \langle \{\mathbf{x} := \mathbf{n}\} \otimes \Sigma' \rangle \mathbf{e}'}{\langle \{\mathbf{x} := \mathbf{n}\} \otimes \Sigma \rangle (\text{rep } \mathbf{x} \ \mathbf{n}' \ \mathbf{e}) \rightarrow \langle \{\mathbf{x} := \mathbf{n}\} \otimes \Sigma' \rangle (\text{rep } \mathbf{x} \ \mathbf{n}' \ \mathbf{e}')}$	[REP-E]
$\frac{-}{\langle \{\mathbf{x} := \mathbf{n}\} \otimes \Sigma \rangle (\text{rep } \mathbf{x} \ \mathbf{n}' \ \mathbf{n}'') \rightarrow \langle \mathbf{x} := \mathbf{n}'' \mid \Sigma \rangle \mathbf{n}''}$	[REP-F]
$\frac{(\mathbf{n} = 1 \Rightarrow \mathbf{e}_o = \mathbf{e}, \bar{\mathbf{x}}_o = \bar{\mathbf{x}}') \quad \text{or} \quad (\mathbf{n} \neq 1 \Rightarrow \mathbf{e}_o = \mathbf{e}', \bar{\mathbf{x}}_o = \bar{\mathbf{x}})}{\langle \{\bar{\mathbf{x}}_o := \bar{\mathbf{n}}\} \otimes \Sigma \rangle (\text{if } \bar{\mathbf{x}}_o : \bar{\mathbf{x}} \ \mathbf{n} \ \mathbf{e} \ \mathbf{e}') \rightarrow \langle \Sigma \rangle \mathbf{e}_o}$	[IF]
$\frac{(\mathbf{n} = 1 \Rightarrow \mathbf{n}_o = \mathbf{n}') \quad \text{or} \quad (\mathbf{n} \neq 1 \Rightarrow \mathbf{n}_o = \mathbf{n}'')}{\langle \Sigma \rangle (\text{mux } \mathbf{n} \ \mathbf{n}' \ \mathbf{n}'') \rightarrow \langle \Sigma \rangle \mathbf{n}_o}$	[MUX]
$\frac{-}{\langle \{\bar{a} : i \mapsto \bar{\mathbf{n}}\} \otimes \Sigma \rangle (\text{nbr}^i \ \mathbf{n}) \rightarrow \langle \{\bar{a} : i \mapsto \bar{\mathbf{n}}\} \otimes \Sigma \mid i \mapsto \mathbf{n} \rangle \{\bar{a} \mapsto \bar{\mathbf{n}}\}}$	[NBR]
$\frac{-}{\langle \{\bar{a} : 0 \mapsto \bar{\mathbf{n}}\} \otimes \{\bar{a} : i \mapsto 0\} \otimes \Sigma \rangle (\text{nbr-range}^i) \rightarrow \langle \{\bar{a} : 0 \mapsto \bar{\mathbf{n}}\} \otimes \{\bar{a} : i \mapsto 0\} \otimes \Sigma \mid i \mapsto 0 \rangle \{\bar{a} \mapsto \bar{\mathbf{n}}\}}$	[NBR-RNG]
$\frac{-}{\langle \Sigma \rangle (\text{fold-hood* } \mathbf{f} \ \mathbf{n}_0 \ \{a \mapsto \mathbf{n}\} \otimes \phi) \rightarrow \langle \Sigma \rangle (\text{fold-hood* } \mathbf{f} \ (\mathbf{f} \ \mathbf{n}_0 \ \mathbf{n}) \ \phi)}$	[FHOOD-R]
$\frac{-}{\langle \Sigma \rangle (\text{fold-hood* } \mathbf{f} \ \mathbf{n}_0 \ 0) \rightarrow \langle \Sigma \rangle \mathbf{n}_0}$	[FHOOD-F]
$\frac{\mathbf{v}_1 \circ \mathbf{v}_2 = \mathbf{v}_0}{\langle \Sigma \rangle (\text{o } \mathbf{v}_1 \ \mathbf{v}_2) \rightarrow \langle \Sigma \rangle \mathbf{v}_0}$	[MATH]
$\frac{(\text{def } \mathbf{F} \ (\bar{\mathbf{x}}) \ \mathbf{e}) \in \mathcal{D}}{\langle \Sigma \rangle (\mathbf{F} \ \bar{\mathbf{v}}) \rightarrow \langle \Sigma \rangle \mathbf{e}[\bar{\mathbf{v}}/\bar{\mathbf{x}}]}$	[DEF]

Figure 3: Operational Semantics of Proto language

$\frac{\langle \Sigma \rangle e \rightarrow \langle \Sigma' \rangle e'}{N \mid a :: \langle \Sigma \rangle e \mapsto N \mid a :: \langle \Sigma' \rangle e'}$	[DEVICE]
$\frac{-}{\{a \rightarrow [\bar{\mu}'] \bar{a}'\} \otimes \{a \xrightarrow{\bar{\mathbf{n}}_r} \bar{a}\} \otimes N \mid a :: \langle \bar{\mu} \otimes \Sigma \rangle \mathbf{n} \mapsto \{a \xrightarrow{\bar{\mathbf{n}}_r} \bar{a}\} \otimes N \mid a :: \langle \Sigma \rangle \mathbf{e}_p \mid \{a \rightarrow [\bar{\mu}] \bar{a}\}}$	[RELOAD]
$\frac{-}{N \mid (a \rightarrow [\bar{\mu}] \bar{a}') \mid a' :: \langle \{a : \bar{\mu}'\} \otimes \Sigma \rangle \mathbf{e} \mapsto N \mid a' :: \langle \{a : \bar{\mu}\} \otimes \Sigma \rangle \mathbf{e}}$	[RECEIVE]
$\frac{-}{\{a \xrightarrow{\bar{\mathbf{n}}_r} \bar{a}\} \otimes N \mid a :: \langle \{\bar{a} : 0 \mapsto \bar{\mathbf{n}}\} \otimes \Sigma \rangle \mathbf{e} \mapsto \{a \xrightarrow{\bar{\mathbf{n}}_r'} \bar{a}'\} \otimes N \mid a :: \langle \{\bar{a}' : 0 \mapsto \bar{\mathbf{n}}_r'\} \otimes \Sigma \rangle \mathbf{e}}$	[MOVE]
$\frac{-}{\{\bullet \xrightarrow{\bar{\mathbf{n}}_r'} a\} \otimes \{a \xrightarrow{\bar{\mathbf{n}}_r} \bullet\} \otimes N \mid a :: \langle \Sigma \rangle \mathbf{e} \mapsto N}$	[DROP]
$\frac{-}{N \mid a :: \langle \{sns(\bar{\mathbf{n}})\} \otimes \Sigma \rangle \mathbf{e} \mapsto N \mid a :: \langle \{sns(\bar{\mathbf{n}}')\} \otimes \Sigma \rangle \mathbf{e}}$	[SIGNAL]

Figure 4: Operational Semantics of Proto platform

(distance-to (sense 1))	[DEF]
(rep d inf (mux (sense 1) 0 (fold-hood* min inf (+ (nbr d) (nbr-range))))))	[SNS]
(rep d inf (mux 0 0 (fold-hood* min inf (+ (nbr d) (nbr-range))))))	[NBR]
(rep d inf (mux 0 0 (fold-hood* min inf (+ (a ₁ ↦ 10 a ₃ ↦ inf) (nbr-range))))))	[NBR-RNG]
(rep d inf (mux 0 0 (fold-hood* min inf (+ (a ₁ ↦ 10 a ₃ ↦ inf) (a ₁ ↦ 10 a ₃ ↦ 10))))))	[MATH]
(rep d inf (mux 0 0 (fold-hood* min inf (a ₁ ↦ 20 a ₃ ↦ inf))))	[F-HOOD-R]
(rep d inf (mux 0 0 (fold-hood* min (min inf 20) (a ₃ ↦ inf))))	[DEF,...]
...	
(rep d inf (mux 0 0 (fold-hood* min 20 (a ₃ ↦ inf))))	[F-HOOD-R]
(rep d inf (mux 0 0 (fold-hood* min (min 20 inf) (0))))	[DEF,...]
...	
(rep d inf (mux 0 0 (fold-hood* min 20 (0))))	[F-HOOD-F]
(rep d inf (mux 0 0 20))	[MUX]
(rep d inf 20)	[REP-F]
20	

Figure 5: Example sequences of transitions, reporting on right the firing rule

4. PLATFORM SEMANTICS

After describing the inner details of computations inside devices, we now turn to platform details, detailing the overall system behaviour: how messages are exchanged, how devices get initialized each time, and how the network of devices can change its structure over time.

4.1 Syntax

The syntactical aspects of a network configuration are described as follows.

$N ::=$	Network
0	Empty network
$(N \mid N)$	Composition
$a :: s$	Device
$a \xrightarrow{n} b$	Link
$a \rightarrow [\bar{\mu}]b$	Pending message from a to b

Again, a network N is a multiset of three kinds of element: (i) $a :: s$ means the network includes device a in state s ; (ii) $a \xrightarrow{n} b$ means device a is connected with device b and their distance is n ; and (iii) $a \rightarrow [\bar{\mu}]b$ means a message from a has been produced to reach b with content $\bar{\mu}$ —namely, a mapping from slots to values.

4.2 Operational semantics

Operational semantics is given by a transition system of the kind $(\mathcal{N}, \rightarrow)$, where \mathcal{N} is the set of states N of the network, and $\rightarrow \subseteq \mathcal{N} \times \mathcal{N}$ is the transition relation. This transition system models all the possible events that can occur in the network and that the platform has somehow to manage, in order to keep the network in a consistent state.

The main assumptions of our model are as follows:

- Message exchange is asynchronous, with each device sending sometime between computation rounds. Order is preserved for each sender, but messages may be lost.
- Execution is asynchronous, parallel, and atomic. As soon as a device computation round is over, the execution of a new round is enabled to start.

Rules of the operational semantics are provided in Figure 4. Rule [DEVICE] is a standard model of asynchronous parallel computation in a device: at any time, any device

may perform a single round of computation, according to the language semantics in the previous section.

Rule [RELOAD] can fire when a device a completes a computation round (program evaluated to result \mathbf{n}). In this case, the Proto platform extracts all outgoing messages $\bar{\mu}$ from a , and retrieves information about neighbours \bar{a} ; accordingly, the device program is restored to original state \mathbf{e}_p , and one message containing $\bar{\mu}$ is sent to any neighbour in \bar{a} . Such messages overwrite any other message previously sent by a and not yet received by its recipient. Note that at this point, the device could start another computation round, though most current Proto implementations wait a predefined ΔT .

With rule [RECEIVE], device a' consumes a message sent from a with content $\bar{\mu}$: the effect is that the pending message is consumed and replaces the set of message-slots corresponding to a currently existing in the store of a' .

Rule [MOVE] handles topological changes (e.g. a device moves, or dies). This is modeled as a change in the outgoing connections of a device a , from devices \bar{a} with distances \bar{n}_r to devices \bar{a}' with distances \bar{n}'_r (changes involving more devices are modelled as a sequence of [MOVE] transitions). As this change occurs, the platform changes the store of a , replacing the record of distances to neighbors (message-slot index 0).

Similarly, rule [DROP] handles devices leaving the network (e.g. due to a failure), removing its representation $a :: s$. Note that this rule is enabled only if there no links to/from this node—some previous transitions [MOVE] must execute to clean the neighbourhood first.

Finally, rule [SIGNAL] changes the values sensed by device a , replacing all $sns(\mathbf{n})$ terms in the store of a .

4.3 Example

Going back to the `distance-to` example, we consider a simple (sub)network with 4 devices a_0, \dots, a_3 , where each a_i is linked to just a_{i-1} and a_{i+1} , distance between any pair of neighbors is 10, and sensor 1 is active for a_0 only. Initial network is hence formed by a term $a_i :: \langle \Sigma_i \rangle \mathbf{e}_p$ for each device (Σ_i initially reifies distance to neighbors as slot 0), and by links of the kind $a_i \xrightarrow{10} a_{i+1}$.

As this system bootstraps, devices start computing by rule [DEVICE]. The first computation round for any device simply sets variable \mathbf{d} to `inf` due to rule [REP-I]. When this computation is over, rule [RELOAD] fires which re-

trieves (and send) no output message—since `nbr` is not executed yet. The current network state can be summarised as `inf, inf, inf, inf`—output values in a_0, \dots, a_3 .

The second computation round for a_0 sets its estimated distance (both in output and in variable `d`) to 0, since sensing is activated there. The second computation round in any other device has no effect, for estimated distance remains set to `inf`. Everywhere, as the second computation round is over, rule [RELOAD] fires which sends a message to neighbors with content `inf` (the initial value of `d`). The current network state is now 0, `inf, inf, inf`.

The third computation round of a_0 causes a message “ $a_0 \mapsto 0 : 0$ ” to be sent to a_1 . When rule [RECEIVE] fires for a_1 this message is received and inserted in the store of a_1 . At the next computation round of a_1 , its estimation distance moves from `inf` to 10, which will eventually get to a_2 —the current network state is now 0, 10, `inf, inf`. The next computation round for a_2 is the one described in Section 3.5.

The whole process of computations and exchange of messages will eventually make the system reach the final state 0, 10, 20, 30: at that point outputs of computations do not change in spite of messages continuously flooding—note that such a final state is reached independently of the order of computation rounds in each device.

5. CONCLUSIONS AND DISCUSSION

In this paper, we have presented a formal operational semantics of Proto. Although the operational semantics presented covers only a representative core subset of the language, all of the major concepts driving the design of Proto are covered and the extension to the entirety of the language should be relatively routine and follow much the same pattern as the semantics presented herein.

We believe this formalization paves the way to a number of future works, all concerned with improving design, analysis and implementation of spatial computing applications. First, the presented operational semantics aims at a reference for implementers of Proto, which will be key to guaranteeing coherence across existing and emerging implementations for diverse execution platforms. Second, as Proto is being used to study self-organization patterns (e.g. [8]), the operational semantics can aid analysis by acting as an executable specification or as a basis for stochastic model-checking or reachability analysis. Finally, this semantics will be an aid to clarifying discussion of Proto itself and its continued future development: for example, propagation of state may be speeded by changing the interpretation of `nbr` expressions inside state updates.

Acknowledgments

Partial support for this work was provided by the EU FP7 project “SAPERRE - Self-aware Pervasive Service Ecosystems” under contract No. 256873, and by the DARPA META project “ARRoW.”

6. REFERENCES

- [1] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai. Meld: A declarative approach to programming ensembles. In *IEEE IROS '07*, 2007.
- [2] J. Bachrach and J. Beal. Programming a sensor network as an amorphous medium. In *Distributed Computing in Sensor Systems (DCOSS) 2006 Poster*, June 2006.
- [3] J. Bachrach, J. Beal, and T. Fujiwara. Continuous space-time semantics allow adaptive program execution. In *IEEE SASO 2007*, July 2007.
- [4] J. Bachrach, J. Beal, and J. McLurkin. Composable continuous space programs for robotic swarms. *Neural Computing and Applications*, 19(6):825–847, 2010.
- [5] J. Beal. Programming an amorphous computational medium. In *Unconventional Programming Paradigms International Workshop*, September 2004.
- [6] J. Beal and J. Bachrach. Infrastructure for engineered emergence in sensor/actuator networks. *IEEE Intelligent Systems*, pages 10–19, March/April 2006.
- [7] J. Beal and J. Bachrach. Cells are plausible targets for high-level spatial languages. In *Spatial Computing Workshop*, 2008.
- [8] J. Beal, J. Bachrach, D. Vickery, and M. Tobenkin. Fast self-healing gradients. In *ACM Symposium on Applied Computing*, March 2008.
- [9] J. Beal, T. Lu, and R. Weiss. Automatic compilation from high-level languages to genetic regulatory networks. In *2nd International Workshop on Bio-Design Automation*, 2010.
- [10] J.-L. Giavitto, C. Godin, O. Michel, and P. Prusinkiewicz. Computational models for integrative and developmental biology. Technical Report 72-2002, Univerite d’Evry, LaMI, 2002.
- [11] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairo. In *DCOSS*, pages 126–140, 2005.
- [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*, 23:396–450, 2001.
- [13] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: the TOTA approach. *ACM Transactions on Software Engineering and Methodology*, 2008.
- [14] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I. *Information and Computation*, 100(1):1–40, September 1992.
- [15] MIT Proto. software available at <http://proto.bbn.com/>, Retrieved Nov. 1st, 2010.
- [16] R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. In *First International Workshop on Data Management for Sensor Networks (DMSN)*, Aug. 2004.
- [17] A. Ricci, M. Viroli, and G. Piancastelli. simpA: An agent-oriented approach for programming concurrent applications on top of Java. *Science of Computer Programming*, 76(1):37–62, January 2011.
- [18] M. D. Rosa, S. C. Goldstein, P. Lee, J. D. Campbell, and P. Pillai. Programming modular robots with locally distributed predicates. In *ICRA '08*, 2008.
- [19] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov. 1994.