Electronic Appendix to: Engineering Resilient Collective Adaptive Systems by Self-Stabilisation

MIRKO VIROLI, Università di Bologna, Italy
GIORGIO AUDRITO, Università di Torino, Italy
JACOB BEAL, Raytheon BBN Technologies, USA
FERRUCCIO DAMIANI, Università di Torino, Italy
DANILO PIANINI, Università di Bologna, Italy

A TYPES FOR BUILT-IN FUNCTIONS USED IN THE EXAMPLES

Figure 1 presents the collection of built-in functions and operators used in this paper (a small subset of possible built-in functions covered by this calculus). A few notes regarding these functions:

- Recall that each built-in function with local arguments is overloaded to work on fields on a pointwise basis.
- The multiplex operator `mux` selects between its second and third arguments based on the value of the first one. This is similar to the `if` keyword but not equivalent: `mux` evaluates both of these arguments everywhere, whereas `if` only evaluates each on the subspace with the matching Boolean value.
- A special role is played by the second-order operator `foldHood` and its specialisations for different aggregation functions (`minHood`, `maxHood` and so on) that collapse a field value into a local value (reminiscent of "reduce" functions common in parallel programming frameworks like MPI). The versions of these operators ending in `+` also aggregate the value corresponding to the current device (which is otherwise ignored), while the versions ending in `Loc` also aggregate a given local value in place of the value corresponding to the current device.

B A MINIMAL CONVENIENT EXTENSION: FUNCTIONAL PARAMETRISATION

As pointed out in Section 3.1 (just before Example 3.1), the pragmatic convenience of the calculus defined so far can be improved to express general-purpose *building blocks*, which are parametric algorithms designed to be applied to a broad class of problems, and necessarily make use of functional parameters to tune their behaviour.

To this end, we extend the syntax of user-defined functions to admit *functional parameters*, ranged over by $\mathtt{z}$. Such *extended functions* can be defined as $\mathtt{def\ d}(\overline{\mathtt{x}})(\overline{\mathtt{z}})\{\mathtt{e}\}$ and called by $\mathtt{d}(\overline{\mathtt{e}})(\overline{\mathtt{f}})$

Authors' addresses: MIRKO VIROLI, Università di Bologna, Cesena, Italy, mirko.viroli@unibo.it; GIORGIO AUDRITO, Università di Torino, Torino, Italy, giorgio.audrito@di.unito.it; JACOB BEAL, Raytheon BBN Technologies, Cambridge (MA), USA, jakebeal@ieee.org; FERRUCCIO DAMIANI, Università di Torino, Torino, Italy, ferruccio.damiani@di.unito.it; DANILO PIANINI, Università di Bologna, Cesena, Italy, danilo.pianini@unibo.it.

XXXX-XXXX/2018/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Built-in Function	Type Signature	Meaning
`uid()`	$() \to \mathtt{num}$	*device identifier*
`+,-,*,/`	$(\mathtt{num},\mathtt{num}) \to \mathtt{num}$	*arithmetical operators*
`<,<=,=,>=,>`	$(\mathtt{num},\mathtt{num}) \to \mathtt{bool}$	*comparison operators*
`&&,||`	$(\mathtt{bool},\mathtt{bool}) \to \mathtt{bool}$	*boolean operators*
$\mathtt{mux}(\mathtt{b},\ell,\ell)$	$\forall t.(\mathtt{bool},t,t) \to t$	*multiplex selection*
$\mathtt{pair}(\ell,\ell)$	$\forall t_1 t_2.(t_1,t_2) \to \mathtt{tuple}(t_1,t_2)$	*pair construction*
$[\overline{\ell}]$	$\forall \overline{t}.(\overline{t}) \to \mathtt{tuple}(\overline{t})$	*tuple construction*
$\mathtt{1st}(\ell),\mathtt{2nd}(\ell),\mathtt{3rd}(\ell)$	$\forall \overline{t}.(\mathtt{tuple}(\overline{t})) \to t_i \ (i = 1,2,3)$	*tuple element access*
$\mathtt{pickHood}(\phi)$	$\forall t.(\mathtt{field}(t)) \to t$	*value in current device*
$\mathtt{foldHood}(\phi,\ell)(\mathtt{f})$	$\forall t.\ (\mathtt{field}(t),t,(t,t) \to t) \to t$	*general neighbour aggregation*
$\mathtt{meanHood}(\phi)$	$\forall t.(\mathtt{field}(t)) \to t$	*average of neighbour values*
$\mathtt{maxHood}(\phi),\mathtt{maxHood+}(\phi)$	$\forall t.(\mathtt{field}(t)) \to t$	*maximum of neighbour values*
$\mathtt{minHood}(\phi),\mathtt{minHood+}(\phi)$	$\forall t.(\mathtt{field}(t)) \to t$	*minimum of neighbour values*
$\mathtt{minHoodLoc}(\phi,\ell)$	$\forall t.(\mathtt{field}(t),t) \to t$	*minimum of neighbor & local values*
`nbrRange(),nbrLag()`	$() \to \mathtt{field}(\mathtt{num})$	*space-time distance from neighbours*
`snsNum()`	$() \to \mathtt{num}$	*generic numeric sensor*
`sns_interval()`	$() \to \mathtt{num}$	*interval with previous round*

Fig. 1. Built-in functions used throughout this paper, with types and meaning.

where the arguments $\overline{\mathtt{f}}$ can be either names of *plain* (i.e., non-extended) functions or functional parameters—names of extended functions are not allowed to be passed as arguments. By convention, we omit the second parentheses whenever no functional parameters are present; so that functions without functional parameters can be defined and called as usual. We also allow the presence of built-in functions admitting functional parameters (e.g., the field aggregator `foldhood(x,y)(z)` which combines values in a field `x` through an initial value `y` and a binary function `z` given as functional parameter).

We remark that a functional parameter `z` (like any other function name) is not an expression by itself, and it only constitutes one when provided with appropriate arguments or passed as argument to a function. This implies for instance that $(\mathtt{if}(\mathtt{e})\{\mathtt{z}_1\}\{\mathtt{z}_2\})(\overline{\mathtt{e}})$ is *not* a valid expression.

A program in the extended syntax can be converted to a program in plain first-order syntax by systematically substituting each call $\mathtt{d}(\overline{\mathtt{e}})(\overline{\mathtt{f}})$ to an extended function $\mathtt{def}\ \mathtt{d}(\overline{\mathtt{x}})(\overline{\mathtt{z}})\{\mathtt{e}\}$ (where the arguments $\overline{\mathtt{f}}$ do not contain functional parameters) by a call $\mathtt{d}_{\overline{\mathtt{f}}}(\overline{\mathtt{e}})$ to a plain function $\mathtt{d}_{\overline{\mathtt{f}}}$ defined as $\mathtt{def}\ \mathtt{d}_{\overline{\mathtt{f}}}(\overline{\mathtt{x}})\{\mathtt{e}[\overline{\mathtt{z}} := \overline{\mathtt{f}}]\}$—thus interpreting functional parameters as macro parameters.[1] For example, the following program (comparing minimum temperature and maximum threshold across a network):

[1]This rewriting process always terminates. Consider F as the set of distinct plain function names that are passed as parameters to extended functions in any point of the program. Then an extended function with k functional parameters can be instantiated at most once for each combination of functions in F, that is, at most n^k times where n is the cardinality of F.

```
def foldwithlocal(field, local, initial)(aggregate) {
  aggregate(foldHood(field, initial)(aggregate), local)
}

def gossip(null)(aggregate, sensor) {
  rep (initial) { (x) => foldwithlocal(nbr{x}, sensor(), initial)(aggregate) }
}

gossip(infinity)(min, sns_temp) < gossip(-infinity)(max, sns_threshold)
```

can be rewritten eliminating functional parameters in the following way:

```
def foldwithlocal_min(field, local, initial) {
  min(foldHood_min(field, initial), local)
}

def gossip_min_temp(initial) {
  rep (initial) { (x) => foldwithlocal_min(nbr{x}, sns_temp(), initial) }
}

def foldwithlocal_max(field, local, initial) {
  max(foldHood_max(field, initial), local)
}

def gossip_max_threshold(initial) {
  rep (initial) { (x) => foldwithlocal_max(nbr{x}, sns_threshold(),initial) }
}

gossip_min_temp(infinity) < gossip_max_threshold(-infinity)
```

where `foldHood_min` and `foldHood_max` can then be substituted with their equivalent versions `minHood`, `maxHood`.

C OPERATIONAL SEMANTICS

We here present a formal semantics that can serve both as a specification for implementation of programming languages based on the field calculus and for reasoning about its properties. Differently from models like BSP [1] that can enact system-wide synchronous rounds in which each device computes exactly once, in our model individual devices undergo computation in (local) rounds, which are sequential for each device, and interleaved among different devices. In each round, a device sleeps for some time, wakes up, gathers information about messages received from neighbours while sleeping, performs an evaluation of the program, and finally emits a message to all neighbours with information about the outcome of computation before going back to sleep. The scheduling of such rounds across the network is fair and asynchronous—the considered notion of fairness is explained in Section 4.1, and basically amounts to the eventual existence of another round for each device and for each moment of time. To simplify the notation, we shall assume a fixed program P. We say that "device δ *fires*", to mean that the main expression $\mathtt{e}_{\mathtt{main}}$ of P is evaluated on δ at a particular round.

Network evolution is modelled (in Section C.2) by a small-step semantics, given as a transition system $\xrightarrow{act}$ on network configurations N, where actions can either be firings of a device or network configuration changes. The semantics of a firing action is defined in terms of the computation that takes place on an individual device, which is modelled (in Section C.1) by a big-step semantics. Note that we use small-step semantics in network transitions to capture the step-by-step evolution of a network, while the more abstract big-step semantics is used in individual devices since in that case only the final result of round computation matters—and is in fact unique.

C.1 Device Semantics

The computation that takes place on a single device is formalised by a big-step semantics, expressed by the judgement $\delta;\Theta \vdash \mathsf{e}_{\mathtt{main}} \Downarrow \theta$, to be read "expression $\mathsf{e}_{\mathtt{main}}$ evaluates to θ on device δ with respect to environment Θ". The result of evaluation is a *value-tree* θ, which is an ordered tree of values that tracks the results of all evaluated subexpressions of $\mathsf{e}_{\mathtt{main}}$. Such a result is made available to δ's neighbours for their subsequent firing (including δ itself, so as to support a form of state across computation rounds). The recently-received value-trees of neighbours are then collected into a *value-tree environment* Θ, implemented as a map from device identifiers to value-trees (written $\overline{\delta} \mapsto \overline{\theta}$ as short for $\delta_1 \mapsto \theta_1, \ldots, \delta_n \mapsto \theta_n$). Intuitively, the outcome of the evaluation will depend on those value-trees. Figure 2 (top) defines value-trees and value-tree environments—the syntax of values v is given in Figure 1 in the main paper.

Example C.1. The graphical representation of the value trees $5\langle 2\langle\rangle, 3\langle\rangle\rangle$ and $5\langle 2\langle\rangle, 3\langle 7\langle\rangle, 1\langle\rangle, 4\langle\rangle\rangle\rangle$ is as follows:

```
   5                      5
  / \                    / \
 2   3                  2   3
                           /|\
                          7 1 4
```

In the following, for sake of readability, we sometimes write the value v as short for the value-tree $\mathsf{v}\langle\rangle$. Following this convention, the value-tree $5\langle 2\langle\rangle, 3\langle\rangle\rangle$ is shortened to $5\langle 2, 3\rangle$, and the value-tree $5\langle 2\langle\rangle, 3\langle 7\langle\rangle, 4\langle\rangle, 4\langle\rangle\rangle\rangle$ is shortened to $5\langle 2, 3\langle 7, 1, 4\rangle\rangle$.

Figure 2 (bottom) defines the judgement $\delta;\Theta \vdash \mathsf{e} \Downarrow \theta$, where: *(i)* δ is the identifier of the current device; *(ii)* Θ is the neighbouring field of the value-trees produced by the most recent evaluation of (an expression corresponding to) e on δ's neighbours; *(iii)* e is a closed run-time expression (i.e., a closed expression that may contain neighbouring field values); *(iv)* the value-tree θ represents the values computed for all the expressions encountered during the evaluation of e—in particular the root of the value tree θ, denoted by $\rho(\theta)$, is the value computed for expression e. The auxiliary function ρ is defined in Figure 2 (second frame).

The operational semantics rules are based on rather standard rules for functional languages, extended so as to be able to evaluate a subexpression e' of e with respect to the value-tree environment Θ' obtained from Θ by extracting the corresponding subtree (when present) in the value-trees in the range of Θ. This process, called *alignment*, is modelled by the auxiliary function π defined in Figure 2 (second frame). This function has two different behaviours (specified by its subscript or superscript): $\pi_i(\theta)$ extracts the i-th subtree of θ; while $\pi^{\ell}(\theta)$ extracts the last subtree of θ, *if* the root of the first subtree of θ is equal to the local (boolean) value ℓ (thus implementing a filter specifically designed for the `if` construct). Auxiliary functions ρ and π apply pointwise on value-tree environments, as defined in Figure 2 (second frame).

Rules [E-LOC] and [E-FLD] model the evaluation of expressions that are either a local value or a neighbouring field value, respectively: note that in [E-FLD] we take care of restructing the domain of a neighbouring field value to the only set of neighbour devices as reported in Θ.

Rule [E-LET] is fairly standard: it first evaluates e_1 and then evaluates the expression obtained from e_2 by replacing all the occurrences of the variable x with the value of e_1.

Rule [E-B-APP] models the application of built-in functions. It is used to evaluate expressions of the form $\mathsf{b}(\mathsf{e}_1 \cdots \mathsf{e}_n)$, where $n \geq 0$. It produces the value-tree $\mathsf{v}\langle\theta_1, \ldots, \theta_n\rangle$, where $\theta_1, \ldots, \theta_n$ are the value-trees produced by the evaluation of the actual parameters $\mathsf{e}_1, \ldots, \mathsf{e}_n$ and v is the value returned by the function. The rule exploits the special auxiliary function $(\!|\mathsf{b}|\!)_{\delta}^{\Theta}$, whose actual

Value-trees and value-tree environments:

$$\theta \;::=\; \mathsf{v}\langle\overline{\theta}\rangle \qquad \text{value-tree}$$
$$\Theta \;::=\; \overline{\delta} \mapsto \overline{\theta} \qquad \text{value-tree environment}$$

Auxiliary functions:

$$\rho(\mathsf{v}\langle\overline{\theta}\rangle) = \mathsf{v}$$
$$\pi_i(\mathsf{v}\langle\theta_1,\ldots,\theta_n\rangle) = \theta_i \quad \text{if } 1 \le i \le n \qquad\qquad \pi^{\ell}(\mathsf{v}\langle\theta_1,\theta_2\rangle) = \theta_2 \quad \text{if } \rho(\theta_1) = \ell$$
$$\pi_i(\theta) = \bullet \quad \text{otherwise} \qquad\qquad \pi^{\ell}(\theta) = \bullet \quad \text{otherwise}$$
$$\text{For } aux \in \rho,\pi_i,\pi^{\ell}: \begin{cases} aux(\delta \mapsto \theta) &= \delta \mapsto aux(\theta) & \text{if } aux(\theta) \ne \bullet \\ aux(\delta \mapsto \theta) &= \bullet & \text{if } aux(\theta) = \bullet \\ aux(\Theta,\Theta') &= aux(\Theta), aux(\Theta') \end{cases}$$
$$args(\mathsf{d}) = \overline{\mathsf{x}} \quad \text{if } \mathtt{def}\ \mathsf{d}(\overline{\mathsf{x}})\ \{\mathsf{e}\} \qquad\qquad body(\mathsf{d}) = \mathsf{e} \quad \text{if } \mathtt{def}\ \mathsf{d}(\overline{\mathsf{x}})\ \{\mathsf{e}\}$$

Syntactic shorthands:

$$\delta;\overline{\pi}(\Theta) \vdash \overline{\mathsf{e}} \Downarrow \overline{\theta} \quad \text{where } |\overline{\mathsf{e}}| = n \quad \text{for } \delta;\pi_1(\Theta) \vdash \mathsf{e}_1 \Downarrow \theta_1 \quad \cdots \quad \delta;\pi_n(\Theta) \vdash \mathsf{e}_n \Downarrow \theta_n$$
$$\rho(\overline{\theta}) \quad \text{where } |\overline{\theta}| = n \quad \text{for } \rho(\theta_1),\ldots,\rho(\theta_n)$$
$$\overline{\mathsf{x}} := \rho(\overline{\theta}) \quad \text{where } |\overline{\mathsf{x}}| = n \quad \text{for } \mathsf{x}_1 := \rho(\theta_1) \;\ldots\; \mathsf{x}_n := \rho(\theta_n)$$

Rules for expression evaluation: $\boxed{\delta;\Theta \vdash \mathsf{e} \Downarrow \theta}$

$$\text{[E-LOC]}\ \frac{}{\delta;\Theta \vdash \ell \Downarrow \ell\langle\rangle} \qquad \text{[E-FLD]}\ \frac{\phi' = \phi|_{\mathbf{dom}(\Theta)\cup\{\delta\}}}{\delta;\Theta \vdash \phi \Downarrow \phi'\langle\rangle}$$

$$\text{[E-LET]}\ \frac{\delta;\pi_1(\Theta) \vdash \mathsf{e}_1 \Downarrow \theta_1 \qquad \delta;\pi_2(\Theta) \vdash \mathsf{e}_2[\mathsf{x} := \rho(\theta_1)] \Downarrow \theta_2}{\delta;\Theta \vdash \mathtt{let}\ \mathsf{x} = \mathsf{e}_1\ \mathtt{in}\ \mathsf{e}_2 \Downarrow \rho(\theta_2)\langle\theta_1,\theta_2\rangle}$$

$$\text{[E-B-APP]}\ \frac{\delta;\overline{\pi}(\Theta) \vdash \overline{\mathsf{e}} \Downarrow \overline{\theta} \qquad \mathsf{v} = (\!|\mathsf{b}|\!)_{\delta}^{\Theta}(\rho(\overline{\theta}))}{\delta;\Theta \vdash \mathsf{b}(\overline{\mathsf{e}}) \Downarrow \mathsf{v}\langle\overline{\theta}\rangle}$$

$$\text{[E-D-APP]}\ \frac{\delta;\overline{\pi}(\Theta) \vdash \overline{\mathsf{e}} \Downarrow \overline{\theta} \qquad \delta;\Theta \vdash body(\mathsf{d})[args(\mathsf{d}) := \rho(\overline{\theta})] \Downarrow \theta'}{\delta;\Theta \vdash \mathsf{d}(\overline{\mathsf{e}}) \Downarrow \rho(\theta')\langle\overline{\theta},\theta'\rangle}$$

$$\text{[E-NBR]}\ \frac{\delta;\pi_1(\Theta) \vdash \mathsf{e} \Downarrow \theta \qquad \phi = \rho(\pi_1(\Theta))[\delta \mapsto \rho(\theta)]}{\delta;\Theta \vdash \mathtt{nbr}\{\mathsf{e}\} \Downarrow \phi\langle\theta\rangle}$$

$$\text{[E-REP]}\ \frac{\begin{array}{l}\delta;\pi_1(\Theta) \vdash \mathsf{e}_1 \Downarrow \theta_1 \\ \delta;\pi_2(\Theta) \vdash \mathsf{e}_2[\mathsf{x} := \ell_0] \Downarrow \theta_2\end{array} \qquad \ell_0 = \begin{cases} \rho(\pi_2(\Theta))(\delta) & \text{if } \delta \in \mathbf{dom}(\Theta) \\ \rho(\theta_1) & \text{otherwise}\end{cases}}{\delta;\Theta \vdash \mathtt{rep}(\mathsf{e}_1)\{(\mathsf{x}) \Rightarrow \mathsf{e}_2\} \Downarrow \rho(\theta_2)\langle\theta_1,\theta_2\rangle}$$

$$\text{[E-IF]}\ \frac{\delta;\pi_1(\Theta) \vdash \mathsf{e} \Downarrow \theta_1 \qquad \rho(\theta_1) \in \{\mathtt{True},\mathtt{False}\} \qquad \delta;\pi^{\rho(\theta_1)}(\Theta) \vdash \mathsf{e}_{\rho(\theta_1)} \Downarrow \theta}{\delta;\Theta \vdash \mathtt{if}(\mathsf{e})\{\mathsf{e}_{\mathtt{True}}\}\{\mathsf{e}_{\mathtt{False}}\} \Downarrow \rho(\theta)\langle\theta_1,\theta\rangle}$$

Fig. 2. Big-step operational semantics for expression evaluation.

definition is abstracted away. This is such that $(\!|\mathsf{b}|\!)_{\delta}^{\Theta}(\overline{\mathsf{v}})$ computes the result of applying built-in function b to values $\overline{\mathsf{v}}$ in the current environment of the device δ. In particular: the built-in 0-ary function `uid` gets evaluated to the current device identifier (i.e., $(\!|\mathtt{uid}|\!)_{\delta}^{\Theta}() = \delta$), and mathematical operators have their standard meaning, which is independent from δ and Θ (e.g., $(\!|+|\!)_{\delta}^{\Theta}(2,3) = 5$).

Example C.2. Evaluating the expression $+(2,3)$ produces the value-tree $5\langle 2,3\rangle$. The value of the whole expression, 5, has been computed by using rule [E-B-APP] to evaluate the application of the sum operator + to the values 2 (the root of the first subtree of the value-tree) and 3 (the root of the second subtree of the value-tree).

The $(\!|\mathsf{b}|\!)_{\delta}^{\Theta}$ function also encapsulates measurement variables such as `nbrRange` and interactions with the external world via sensors and actuators.

Rule [E-D-APP] models the application of a user-defined function. It is used to evaluate expressions of the form $\mathtt{d}(\mathtt{e}_1 \ldots \mathtt{e}_n)$, where $n \geq 0$. It resembles rule [E-B-APP] while producing a value-tree with one more subtree θ', which is produced by evaluating the body of the function d (denoted by *body*(d)) substituting the formal parameters of the function (denoted by *args*(d)) with the values obtained evaluating $\mathtt{e}_1, \ldots \mathtt{e}_n$.

Rule [E-REP] implements internal state evolution through computational rounds: $\mathtt{rep(e_1)\{(x)\texttt{=>}e_2\}}$ evaluates to $\mathtt{e}_2[\mathtt{x} := \mathtt{v}]$ where v is obtained from $\mathtt{e}_1$ on the first firing of a device, from the previous value of the whole rep-expression otherwise.

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

Rule [E-NBR] models device interaction. It first collects neighbour's values for expressions e as $\phi = \rho(\pi_1(\Theta))$, then evaluates e in δ and updates the corresponding entry in ϕ.

Example C.4. To illustrate rule [E-NBR], consider the program $\mathtt{e}' = \mathtt{minHood(nbr\{snsNum()\})}$, where the 1-ary built-in function `minHood` returns the lower limit of values in the range of its neighbouring field argument, and the 0-ary built-in function `snsNum` returns the numeric value measured by a sensor. Suppose that the program runs on a network of three devices δ_A, δ_B, and δ_C where:

- δ_B and δ_A are mutually connected, δ_B and δ_C are mutualy connected, while δ_A and δ_C are not connected;
- `snsNum` returns 1 on δ_A, 2 on δ_B, and 3 on δ_C; and
- all devices have an initial empty tree-environment $\emptyset$.

Suppose that device δ_A is the first device that fires: the evaluation of `snsNum()` on δ_A yields 1 (by rules [E-LOC] and [E-B-APP], since $(\!|\mathtt{snsNum}|\!)_{\delta_A}^{\emptyset}() = 1$); the evaluation of `nbr{snsNum()}` on δ_A yields $(\delta_A \mapsto 1)\langle 2\rangle$ (by rule [E-NBR]); and the evaluation of e′ on δ_A yields

$$\theta_A \quad = \quad 1\langle(\delta_A \mapsto 1)\langle 1\rangle\rangle$$

(by rule [E-B-APP], since $(\!|\mathtt{minHood}|\!)_{\delta_A}^{\emptyset}(\delta_A \mapsto 1) = 1$). Therefore, at its first fire, device δ_A produces the value-tree θ_A. Similarly, if device δ_C is the second device that fires, it produces the value-tree

$$\theta_C \quad = \quad 3\langle(\delta_C \mapsto 3)\langle 3\rangle\rangle$$

Suppose that device δ_B is the third device that fires. Then the evaluation of e' on δ_B is performed with respect to the value tree environment $\Theta_B = (\delta_A \mapsto \theta_A,\ \delta_C \mapsto \theta_C)$ and the evaluation of its subexpressions `nbr{snsNum()}` and `snsNum()` is performed, respectively, with respect to the following value-tree environments obtained from Θ_B by alignment:

$$\begin{array}{lllll}\Theta'_B & = & \pi_1(\Theta_B) & = & (\delta_A \mapsto (\delta_A \mapsto 1)\langle 1\rangle,\ \ \delta_C \mapsto (\delta_C \mapsto 3)\langle 3\rangle)\\ \Theta''_B & = & \pi_1(\Theta'_B) & = & (\delta_A \mapsto 1,\ \ \delta_C \mapsto 3)\end{array}$$

We thus have that $(\!|\mathtt{snsNum}|\!)^{\Theta''_B}_{\delta_B}() = 2$; the evaluation of `nbr{snsNum()}` on δ_B with respect to Θ'_B produces the value-tree $\phi\langle 2\rangle$ where $\phi = (\delta_A \mapsto 1, \delta_B \mapsto 2, \delta_C \mapsto 3)$; and $(\!|\mathtt{minHood}|\!)^{\Theta_B}_{\delta_B}(\phi) = 1$. Therefore the evaluation of e' on δ_B produces the value-tree $\theta_B = 1\langle\phi\langle 2\rangle\rangle$. Note that, if the network topology and the values of the sensors will not change, then: any subsequent fire of device δ_B will yield a value-tree with root 1 (which is the minimum of `snsNum` across δ_A, δ_B and δ_C); any subsequent fire of device δ_A will yield a value-tree with root 1 (which is the minimum of `snsNum` across δ_A and δ_B); and any subsequent fire of device δ_C will yield a value-tree with root 2 (which is the minimum of `snsNum` across δ_B and δ_C).

Rule [E-IF] is almost standard, except that it performs domain restriction $\pi^{\mathtt{True}}(\Theta)$ (resp. $\pi^{\mathtt{False}}(\Theta)$) in order to guarantee that subexpression $\mathsf{e}_{\mathtt{True}}$ is not matched against value-trees obtained from $\mathsf{e}_{\mathtt{False}}$ (and vice-versa).

C.2 Network Semantics

The overall network evolution is formalised by the small-step operational semantics given in Figure 3 as a transition system on network configurations N. Figure 3 (top) defines key syntactic elements to this end. Ψ models the overall status of the devices in the network at a given time, as a map from device identifiers to value-tree environments. From it, we can define the state of the field at that time by summarising the current values held by devices. τ models *network topology*, namely, a directed neighbouring graph, as a map from device identifiers to set of identifiers (denoted as I). Σ models *sensor (distributed) state*, as a map from device identifiers to (local) sensors (i.e., sensor name/value maps denoted as σ). Then, *Env* (a couple of topology and sensor state) models the system's environment. So, a whole network configuration N is a couple of a status field and environment.

We use the following notation for status fields. Let $\overline{\delta} \mapsto \Theta$ denote a map from device identifiers $\overline{\delta}$ to the same value-tree environment Θ. Let $\Theta_0[\Theta_1]$ denote the value-tree environment with domain $\mathbf{dom}(\Theta_0) \cup \mathbf{dom}(\Theta_1)$ coinciding with Θ_1 in the domain of Θ_1 and with Θ_0 otherwise. Let $\Psi_0[\Psi_1]$ denote the status field with the *same domain* as Ψ_0 made of $\delta \mapsto \Psi_0(\delta)[\Psi_1(\delta)]$ for all δ in the domain of Ψ_1, $\delta \mapsto \Psi_0(\delta)$ otherwise.

We consider transitions $N \xrightarrow{act} N'$ of two kinds: firings, where *act* is the corresponding device identifier, and environment changes, where *act* is the special label *env*. This is formalised in Figure 3 (bottom). Rule [N-FIR] models a computation round (firing) at device δ: it takes the local value-tree environment filtered out of old values $F(\Psi)(\delta)$;[2] then by the single device semantics it obtains the device's value-tree θ,[3] which is used to update the system configuration of δ and of δ's neighbours.

[2]Function $F(\Psi)$ in rule [N-FIR] models a filtering operation that clears out old stored values from the value-tree environments in Ψ, implicitly based on space/time tags.

[3]We shall assume that any device firing is guaranteed to terminate in any environmental condition. Termination of a device firing is clearly not decidable, but we shall assume—without loss of generality for the results of this paper—that a decidable subset of the termination fragment can be identified (e.g., by ruling out recursive user-defined functions or by applying standard static analysis techniques for termination).

System configurations and action labels:

$$
\begin{array}{lcll}
\Psi & ::= & \overline{\delta} \mapsto \overline{\Theta} & \text{status field} \\
\tau & ::= & \overline{\delta} \mapsto \overline{I} & \text{topology} \\
\Sigma & ::= & \overline{\delta} \mapsto \overline{\sigma} & \text{sensors-map} \\
Env & ::= & \tau, \Sigma & \text{environment} \\
N & ::= & \langle Env; \Psi \rangle & \text{network configuration} \\
act & ::= & \delta \;\big|\; env & \text{action label}
\end{array}
$$

Environment well-formedness:

$WFE(\tau,\Sigma)$ holds if τ,Σ have same domain, and τ's values do not escape it.

Transition rules for network evolution: $\boxed{N \xrightarrow{act} N}$

$$
\text{[N-FIR]}\ \frac{Env = \tau,\Sigma \quad \tau(\delta) = \overline{\delta} \quad \delta; F(\Psi)(\delta) \vdash \mathtt{e}_{\mathtt{main}} \Downarrow \theta \text{ (w.r.t. } \Sigma(\delta)) \quad \Psi_1 = \delta\overline{\delta} \mapsto \{\delta \mapsto \theta\}}{\langle Env; \Psi \rangle \xrightarrow{\delta} \langle Env; F(\Psi)[\Psi_1] \rangle}
$$

$$
\text{[N-ENV]}\ \frac{WFE(Env') \quad Env' = \tau, \overline{\delta} \mapsto \overline{\sigma} \quad \Psi_0 = \overline{\delta} \mapsto \emptyset}{\langle Env; \Psi \rangle \xrightarrow{env} \langle Env'; \Psi_0[\Psi] \rangle}
$$

Fig. 3. Small-step operational semantics for network evolution.

Rule [N-ENV] takes into account the change of the environment to a new *well-formed* environment Env'—environment well-formedness is specified by the predicate $WFE(Env)$ in Figure 3 (middle). Let $\overline{\delta}$ be the domain of Env'. We first construct a status field Ψ_0 associating to all the devices of Env' the empty context $\emptyset$. Then, we adapt the existing status field Ψ to the new set of devices: $\Psi_0[\Psi]$ automatically handles removal of devices, map of new devices to the empty context, and retention of existing contexts in the other devices.

Example C.5. Consider a network of devices with the program $\mathtt{e}' = \mathtt{minHood(nbr\{snsNum()\})}$ introduced in Example C.4. The network configuration illustrated at the beginning of Example C.4 can be generated by applying rule [N-ENV] to the empty network configuration. I.e., we have

$$\langle \emptyset, \emptyset; \emptyset \rangle \xrightarrow{env} \langle Env_0; \Psi_0 \rangle$$

where

- $Env_0 = \tau_0, \Sigma_0$,
- $\tau_0 = (\delta_A \mapsto \{\delta_B\}, \delta_B \mapsto \{\delta_A, \delta_C\}, \delta_C \mapsto \{\delta_B\})$,
- $\Sigma_0 = (\delta_A \mapsto (\mathtt{snsNum} \mapsto 1), \delta_B \mapsto (\mathtt{snsNum} \mapsto 2), \delta_C \mapsto (\mathtt{snsNum} \mapsto 3))$, and
- $\Psi_0 = (\delta_A \mapsto \emptyset, \delta_B \mapsto \emptyset, \delta_C \mapsto \emptyset)$.

Then, the tree fires of devices δ_A, δ_C and δ_B illustrated in Example C.4 correspond to the following transitions, respectively.

(1) $\langle Env_0; \Psi_0 \rangle \xrightarrow{\delta_A} \langle Env_0; \Psi' \rangle$, where
   - $\Psi' = (\delta_A \mapsto (\delta_A \mapsto \theta_A),\ \delta_B \mapsto (\delta_A \mapsto \theta_A),\ \delta_C \mapsto \emptyset)$, and
   - $\theta_A = 1\langle(\delta_A \mapsto 1)\langle 1 \rangle\rangle$;

(2) $\langle Env_0; \Psi' \rangle \xrightarrow{\delta_C} \langle Env_0; \Psi'' \rangle$, where
- $\Psi'' = (\delta_A \mapsto (\delta_A \mapsto \theta_A),\ \delta_B \mapsto (\delta_A \mapsto \theta_A, \delta_C \mapsto \theta_C),\ \delta_C \mapsto (\delta_C \mapsto \theta_C))$, and
- $\theta_C = 1\langle(\delta_C \mapsto 3)\langle 3\rangle\rangle$;

(3) $\langle Env_0; \Psi'' \rangle \xrightarrow{\delta_B} \langle Env_0; \Psi''' \rangle$, where
- $\Psi''' = (\delta_A \mapsto (\delta_A \mapsto \theta_A, \delta_B \mapsto \theta_B),$
  $\delta_B \mapsto (\delta_A \mapsto \theta_A, \delta_B \mapsto \theta_B, \delta_C \mapsto \theta_C),$
  $\delta_C \mapsto (\delta_B \mapsto \theta_B, \delta_C \mapsto \theta_C)),$
- $\theta_B = 1\langle\phi\langle 2\rangle\rangle$, and
- $\phi = (\delta_A \mapsto 1, \delta_B \mapsto 2, \delta_C \mapsto 3)$.

D PROOF OF EVENTUAL BEHAVIOUR PRESERVING EQUIVALENCES

Restatement of Proposition 1 (Eventual behaviour preserving equivalences).

(1) Let $\mathtt{e}_1$, $\mathtt{e}_2$ be self-stabilising expressions with the same eventual behaviour. Then given a self-stabilising expression $\mathtt{e}$, swapping $\mathtt{e}_1$ for $\mathtt{e}_2$ in $\mathtt{e}$ does not change the eventual outcome of its computation.

(2) Let $\mathtt{f}_1$, $\mathtt{f}_2$ be self-stabilising functions with the same eventual behaviour. Then given a self-stabilising expression $\mathtt{e}$, swapping $\mathtt{f}_1$ for $\mathtt{f}_2$ in $\mathtt{e}$ does not change the eventual outcome of its computation.

(3) Let $\mathtt{e}$ be a self-stabilising expression calling a user-defined self-stabilising function $\mathtt{d}$ such that in $body(\mathtt{f})$ no $\mathtt{x} \in args(\mathtt{f})$ occurs in the branch of an `if`. Let $\mathtt{e}'$ be the expression obtained from $\mathtt{e}$ by substituting each function application of the kind $\mathtt{f}(\overline{\mathtt{e}})$ with $body(\mathtt{f})\ [args(\mathtt{f}) := \overline{\mathtt{e}}]$. Then $\mathtt{e}'$ is self-stabilising and has the same eventual behaviour as $\mathtt{e}$ (i.e. $[\![\mathtt{e}]\!] = [\![\mathtt{e}']\!]$).

Proof. (1) By straightforward induction on the structure of an expression. The base case is given by expressions without occurrences of $\mathtt{e}_1$ and $\mathtt{e}_2$, and by expressions $\mathtt{e}_i$ for $i = 1, 2$. The inductive step follows by compositionality of the operational semantics.

(2) For the same reasoning as in point (1), where the base case is given by expressions without occurrences of $\mathtt{f}_1$ and $\mathtt{f}_2$ and by expressions $\mathtt{f}_i(\overline{\mathtt{e}})$ for $i = 1, 2$.

(3) Recall that no expressions with side effects are contemplated in the present calculus. Since no argument of `f` occurs in the branch of an `if`, each of those arguments is evaluated in the same environment as the whole function application $\mathtt{f}(\overline{\mathtt{e}})$. It follows that $\mathtt{e}_1 = \mathtt{f}(\overline{\mathtt{e}})$ and $\mathtt{e}_2 = body(\mathtt{f})\ [args(\mathtt{f}) := \overline{\mathtt{e}}]$ have the same behaviour (hence the same eventual behaviour). The thesis follows then by applying point (1) to expressions $\mathtt{e}_1$ and $\mathtt{e}_2$. □

E PROOF OF SELF-STABILISATION FOR THE FRAGMENT

We report complete proofs for the statements given in Section 5.3. We first prove self-stabilisation for the minimising `rep` pattern (Lemma 1), since it is technically more involved than the proof of self-stabilisation for the remainder of the fragment. We then prove self-stabilisation through a variation of the goal results (Lemma 2) more suited for inductive reasoning. Theorems 1 and 2 will then follow by inspecting the proof of those lemmas.

Let $\mathtt{s}_{\mathtt{min}}$ = $\mathtt{rep(e)\{(x)\text{=>}f^{R}(minHoodLoc(f^{MP}(nbr\{x\},\overline{s}),s),x,\overline{e})\}}$ be a minimising `rep` expression such that $[\![\overline{\mathtt{s}}]\!] = \overline{\Phi}$, $[\![\mathtt{s}]\!] = \Phi$. Let $P = \overline{\delta}$ be a path in the network (a sequence of pairwise connected devices), and define its *weight* in $\mathtt{s}_{\mathtt{min}}$ as the result of picking the eventual value $\ell_1 = \Phi(\delta_1)$ of

$\mathtt{s}$ in the first device δ_1, and repeatedly passing it to subsequent devices through the monotonic progressive function, so that $\ell_{i+1} = \mathtt{f}^{\mathsf{MP}}(\ell_i, \overline{\mathtt{v}})$ where $\overline{\mathtt{v}}$ is the result of projecting fields in $\overline{\Phi}(\delta_{i+1})$ to their δ_i component (leaving local values untouched). Notice that the weight is well-defined since function $\mathtt{f}^{\mathsf{MP}}$ is required to be stateless.

Lemma 1. *Let* $\mathtt{s}$ *be a minimising* `rep` *expression. Then* $\mathtt{s}$ *self-stabilises in each device* δ *to the minimal weight in* $\mathtt{s}$ *for a path* P *ending in* δ*.*

Proof. Let ℓ_δ be the minimal weight for a path P ending in δ, and let $\delta^0, \delta^1, \ldots$ be the list of all devices δ ordered by increasing ℓ_δ. Notice that the path P of minimal weight ℓ_{δ^i} for device i can only pass through nodes such that $\ell_{\delta^j} \leq \ell_{\delta^i}$ (thus s.t. $j < i$). In fact, whenever a path P contains a node j the weight of its prefix until j is at least ℓ_{δ^j}; thus any longer prefix has weight strictly greater than ℓ_{δ^j} since $\mathtt{f}^{\mathsf{MP}}$ is progressive.

Let $N_0 \xrightarrow{\delta_0} N_1 \xrightarrow{\delta_1} \ldots$ be a fair evolution[4] and assume w.l.o.g. that all subexpressions of $\mathtt{s}$ not involving $\mathtt{x}$ have already self-stabilised to computational fields $\overline{\Phi}, \Phi$ (as in the definition of weight) in the initial state N_0. We now prove by complete induction on i that device δ^i stabilises to ℓ_{δ^i} after a certain step t_i.

Assume that devices δ^j with $j < i$ are all self-stabilised (from a certain step t_{i-1}), and consider the evaluation of expression $\mathtt{s}$ in a device δ^k with $k \geq i$. Since the local argument ℓ of `minHoodLoc` is also the weight of the single-node path $P = \delta^k$, it has to be at least $\ell \geq \ell_{\delta^k} \geq \ell_{\delta^i}$. Similarly, the restriction ϕ' of the field argument ϕ of `minHoodLoc` to devices δ^j with $j < i$ has to be at least $\phi' \geq \ell_{\delta^k} \geq \ell_{\delta^i}$ since it corresponds to weights of (not necessarily minimal) paths P ending in δ^k (obtained by extending a minimal path for a device δ^j with $j < i$ with the additional node δ^k). Finally, the complementary restriction ϕ'' of ϕ to devices δ^j with $j \geq i$ is strictly greater than the minimum value for $\mathtt{x}$ among those devices, since $\mathtt{f}^{\mathsf{MP}}$ is progressive.

It follows that as long as the minimum value for $\mathtt{x}$ among non-stable devices is lower than ℓ_{δ^i}, the result of the `minHoodLoc` subexpression is strictly greater than this minimum value. Since the overall value of $\mathtt{s}$ is obtained by combining the output of `minHoodLoc` with the previous value for $\mathtt{x}$ through the rising function $\mathtt{f}^{\mathsf{R}}$ (and a rising function does not drop below the minimum of its arguments), the minimum value for $\mathtt{x}$ among non-stable devices cannot decrease as long as it is lower than ℓ_{δ^i}, and it cannot drop below ℓ_{δ^i} if it is already greater than that.

Furthermore, the minimum has to eventually increase until it reaches at least ℓ_{δ^i}. Recall that a rising function selects its first argument infinitely often (since the order $\lhd$ is noetherian). Thus each device realising a minimum for $\mathtt{x}$ among non-stable devices has to eventually evaluate $\mathtt{s}$ to the output of the `minHoodLoc` subexpression, which is strictly higher than the previous minimum, and it will not be able to reach the previous minimum afterwards.

Let $t' \geq t_{i-1}$ be the first step in which the minimum for $\mathtt{x}$ among non-stable devices is at least ℓ_{δ^i}, and consider device δ^i. Let P be a path of minimum weight for δ^i, then either:

- $P = \delta^i$, so that ℓ_{δ^i} is exactly the local argument of the `minHoodLoc` operator, hence also the output of it (since the field argument is greater than ℓ_{δ^i}).
- $P = Q, \delta^i$ where Q ends in δ^j with $j < i$. Since $\mathtt{f}^{\mathsf{MP}}$ is monotonic non-decreasing, the weight of Q', δ^i (where Q' is minimal for δ^j) is not greater than that of P; in other words, $P' = Q', \delta^i$ is also a path of minimum weight. It follows that $\phi(\delta^j)$ (where ϕ is the field argument of the `minHoodLoc` operator) is exactly ℓ_{δ^i}.

[4]Notice that δ_0 is the first device firing while δ^0 is the device with minimal weight.

In both cases, the output of `minHoodLoc` in δ^i stabilises to ℓ_{δ^i} from t' on. Let t_i be the first step after t' in which the rising function $\mathtt{f}^{\mathsf{R}}$ selects its first argument ℓ_{δ^i}. Then expression s in device δ^i is self-stabilised to ℓ_{δ^i} from t_i on, concluding the inductive step and the proof. □

Let Φ be a computational field as defined in Section 4.2. We write $\mathtt{s}[\mathtt{x} := \Phi]$ to indicate an aggregate process in which each device is computing a possibly different substitution $\mathtt{s}[\mathtt{x} := \Phi(\delta)]$ of the same expression.

Lemma 2. *Assume that every built-in operator is self-stabilising. Let* s *be an expression with free variables* $\overline{\mathtt{x}}$ *in the self-stabilising fragment, and* $\overline{\Phi}$ *be a sequence of computational fields of the same length. Then* $\mathtt{s}[\overline{\mathtt{x}} := \overline{\Phi}]$ *is self-stabilising.*

Proof. The proof proceeds by induction on the syntax of expressions and programs. Let s be an expression in the fragment, then it can be:

- A variable $\mathtt{x}_i$, so that $\mathtt{s}[\overline{\mathtt{x}} := \overline{\Phi}] = \Phi_i$ is already self-stabilised.
- A value v, so that $\mathtt{s}[\overline{\mathtt{x}} := \overline{\Phi}] = \mathtt{v}$ is already self-stabilised.
- A let-expression `let x = s`$_1$ `in s`$_2$. Fix an environment *Env*, in which expression $\mathtt{s}_1$ self-stabilises to Φ after fire t. After t, `let x = s`$_1$ `in s`$_2$ evaluates to the same value of the expression $\mathtt{s}_2[\mathtt{x} := \Phi]$ which is self-stabilising by inductive hypothesis.
- A functional application $\mathtt{f}(\overline{\mathtt{s}})$. Fix an environment *Env*, in which all expressions $\overline{\mathtt{s}}$ self-stabilise to $\overline{\Phi}$ after fire t. After t, if f is a built-in function then $\mathtt{f}(\overline{\mathtt{s}})$ is already self-stabilised. Otherwise, if f is a user-defined function then $\mathtt{f}(\overline{\mathtt{s}})$ evaluates to the same value of the expression $body(\mathtt{f})[args(\mathtt{f}) := \overline{\Phi}]$ which is self-stabilising by inductive hypothesis.
- A conditional $\mathtt{s} = \mathtt{if}(\mathtt{s}_1)\{\mathtt{s}_2\}\{\mathtt{s}_3\}$. Fix an environment *Env*, in which expression $\mathtt{s}_1$ self-stabilises to Φ_{guard}. Let $Env_{\mathtt{True}}$ be the sub-environment consisting of devices δ such that $\Phi_{guard}(\delta) = \mathtt{True}$, and analogously $Env_{\mathtt{False}}$. Assume that $\mathtt{s}_2$ self-stabilises to $\Phi_{\mathtt{True}}$ in $Env_{\mathtt{True}}$ and $\mathtt{s}_3$ to $\Phi_{\mathtt{False}}$ in $Env_{\mathtt{False}}$. Since a conditional is computed in isolation in the above defined sub-environments, s self-stabilises to $\Phi = \Phi_{\mathtt{True}} \cup \Phi_{\mathtt{False}}$.
- A neighbourhood field construction `nbr{s}`. Fix an environment *Env*, in which expression s self-stabilises to Φ after fire t. Then `nbr{s}` self-stabilises to the corresponding Φ' after one more firing of each device, where $\Phi'(\delta)$ is Φ restricted to $\tau(\delta)$.
- A converging rep: $\mathtt{s} = \mathtt{rep(e)\{(x)\text{=>}f^C(nbr\{x\},nbr\{s\},\overline{e})\}}$. Fix an environment *Env* and a fair evolution of the network $N_0 \xrightarrow{\delta_0} N_1 \xrightarrow{\delta_1} \ldots$, and let t be such that all subexpressions of s not containing x have self-stabilised after t. Assume that s self-stabilises to Φ; we prove that s stabilises as well to the same Φ.
  Given any index $i \geq t$, let d^i be the maximum distance $\mathtt{x} - \Phi(\delta^i)$ of x from s realised by a device δ^i in the network. Let $t_0 = t$ and t_{i+1} be the first firing of device δ^{t_i} after t_i. Since δ^{t_i} realises the maximum distance in the whole network N_{t_i}, no device firing between t_i and t_{i+1} can assume a value more distant than d^{t_i} without violating the converging property. Thus d^i, δ^i remains the same in the whole interval from t_i to t_{i+1} (excluded).
  Finally, in fire t_{i+1} device δ^{t_i} recomputes its value, necessarily obtaining a closer value to $\Phi(\delta^{t_i})$ (by the converging property) thus forcing the overall maximal distance in the network to reduce: $d^{t_{i+1}} < d^{t_i}$. Since the set of possible values is finite, so are the possible distances and eventually the maximal distance d^i will reach 0.

- An acyclic `rep`: $\mathtt{s} = \mathtt{rep(e)\{(x)\texttt{=>}f(mux(nbrlt(s}_p\mathtt{),nbr\{x\},s),\overline{s})\}}$. Fix an environment *Env* and a fair evolution of the network $N_0 \xrightarrow{\delta_0} N_1 \xrightarrow{\delta_1} \ldots$, and let t be such that all subexpressions of `s` not containing `x` have self-stabilised after t.
  Let $t_0 \geq t$ be any fire of the device δ^0 of minimal potential $\mathtt{s}_p$ in the network after t. Since δ^0 is minimal, $\mathtt{mux(nbrlt(s}_p\mathtt{),nbr\{x\},s)}$ reduces to `s` and the whole `s` to $\mathtt{f(s,\overline{s})}$, which is self-stabilising (after some $t_0' \geq t_0$) for inductive hypothesis.
  Let $t_1 \geq t_0'$ be any fire of the device δ^1 of second minimal potential after t_0'. Then the value of $\mathtt{mux(nbrlt(s}_p\mathtt{),nbr\{x\},s)}$ in δ^1 only (possibly) depends on the value of the device of minimal potential, which is already self-stabilised. Thus by inductive hypothesis `s` self-stabilises also in δ^1 after some index $t_1' \geq t_1$. By repeating the same reasoning on all devices in order of increasing potential, we obtain a final t_n' after which all devices have self-stabilised.
- A minimising `rep`: this case is proved for closed expressions in Lemma 1, and its generalisation to open expressions is straightforward. □

Restatement of Theorem 1 (Fragment Stabilisation). Let `s` be a closed expression in the self-stabilising fragment, and assume that every built-in operator is self-stabilising. Then `s` is self-stabilising.

Proof. Follows directly from Lemma 2 when `s` has no free variables. □

Restatement of Theorem 2 (Substitutability). The following three equivalences hold: *(i)* each `rep` in a self-stabilising fragment self-stabilises to the same value under arbitrary substitution of the initial condition; *(ii)* the *converging* `rep` pattern self-stabilises to the same value as the single expression `s` occurring in it; *(iii)* the *minimising* `rep` pattern self-stabilises to the same value as the analogous pattern where $\mathtt{f}^{\mathsf{R}}$ is the identity on its first argument.

Proof. Follows from inspecting the proof of Lemmas 1 and 2. □

REFERENCES

[1] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.