Predictable Self-Organization with Computational Fields
Pt 2
Computational Fields: calculus, examples, self-stabilisation

Mirko Viroli ¹ Jacob Beal ²

¹ALMA MATER STUDIORUM—Università di Bologna, Italy
mirko.viroli@unibo.it

²BBN Technologies, USA
jakebeal@bbn.com

1. Why a calculus of fields

2. The Field Calculus – “global viewpoint”

3. Semantic details – “local viewpoint”

4. Bootstrapping property verification: Self-stabilisation
Outline

1. Why a calculus of fields
2. The Field Calculus – “global viewpoint”
3. Semantic details – “local viewpoint”
4. Bootstrapping property verification: Self-stabilisation
Scope of the research

Towards a foundational approach

- Many languages/modes incorporate features of aggregate programming
  (see a survey in [Beal et al., 2013])
- No general formalisation approaches exist
Scope of the research

Towards a foundational approach

- Many languages/modes incorporate features of aggregate programming (see a survey in [Beal et al., 2013])
- No general formalisation approaches exist

Our research steps

1. Identify a minimal set of ingredients of spatial computing
2. Devise a core calculus of computational fields [Viroli et al., 2013]
3. Provide a full formalisation in the proglangs/concurrency style
4. Isolate fragments enjoying certain properties
Expected impact

Perspective

- Paving the way towards advanced behavioural analysis
  - Sufficient conditions for self-stabilisation and density independence
  - Type-checking correctness properties
  - Formal local-to-global and global-to-local connections
- Devising tools for proper system engineering
  - building blocks, languages, patterns, APIs, infrastructure
An example: a “channel” deployed in physical space

- Computational field (or field): a mapping from nodes to values
- Can be: static, dynamic, or require time to stabilise
- Sometimes abstracted to a continuous space-time domain
How to achieve this structure in a self-organised way?

Local viewpoint
- What is the single-node behaviour?
- How should a node interact with neighbours and locally compute?

Global viewpoint
- What is the global structure that emerges, what are its properties?
- Can we create it compositionally?
How to achieve this structure in a self-organised way?

Local viewpoint

- What is the single-node behaviour?
- How should a node interact with neighbours and locally compute?

Global viewpoint

- What is the global structure that emerges, what are its properties?
- Can we create it compositionally?

Computing with fields as first-class abstractions

- “Inputs” (dynamically coming from sensors) are fields!
- “Outputs” (data produced, signal to actuators) are fields!
- Computation is a function from inputs to outputs as usual

⇒ use a functional style to manipulate field expressions
How would we create a “channel” compositionally?

Using a functional style and mathematical operators

\[
\text{channel}(src, dest, width) = \\
\text{dilate}(\text{distance-to}(src) + \text{distance-to}(dest) \leq \text{distance}(src, dest), width)
\]
From global back to local viewpoint

Working at the global level is key

- Provides a more abstract framework
- Captures desired concepts more directly
- Facilitates design, specification, understanding

We need to map back to locality

- How do we “compile” the global specification back in the single node behaviour?
A core calculus of computational fields

What do we need to tackle the problem solidly?

- A handy formalisation framework
- Built on top of known and well-studied techniques
- Quickly leading to tool implementation

The notion of core calculus

- A tiny language: syntax + operational semantics (+ typing)
- Strives for the best tradeoff between:
  ▶ compactness, simplicity (fewest constructs/concepts)
  ▶ expressiveness (many program/behaviours)

Popular examples:

- $\lambda$-calculus [Barendregt, 1984]: a core for functional programming
- $\pi$-calculus [Milner, 1999]: a core for interactive programming
- FJ [Igarashi et al., 1999]: a core for object-oriented programming

$\Rightarrow$ virtually any programming constructs is formulated in that way.
A core calculus of computational fields

What do we need to tackle the problem solidly?

- A handy formalisation framework
- Built on top of known and well-studied techniques
- Quickly leading to tool implementation

The notion of core calculus

- A tiny language: syntax + operational semantics (+ typing)
- Strives for the best tradeoff between:
  - compactness, simplicity (fewest constructs/concepts)
  - expressiveness (many program/behaviours)
- Popular examples:
  - $\lambda$-calculus [Barendregt, 1984]: a core for functional programming
  - $\pi$-calculus [Milner, 1999]: a core for interactive programming
  - FJ [Igarashi et al., 1999]: a core for object-oriented programming
  - Virtually any programming constructs is formulated in that way..
The case of $\lambda$-calculus

Formalisation
- Syntax: lambdas, variables, application
- Semantics: one-step execution of function application

\[
L ::= \lambda x. L \mid x \mid LL \\
(\lambda x. L)M \rightarrow L[M/x]
\]
Outline

1. Why a calculus of fields
2. The Field Calculus – “global viewpoint”
3. Semantic details – “local viewpoint”
4. Bootstrapping property verification: Self-stabilisation
The calculus of computational fields [Viroli et al., 2013]

Main constructs

- Retains the functional style (recursive function definitions, one main)
- Built-in functions model sensor acquisition and any local algorithm
- Key computational field constructs
  - \texttt{rep} — Repetition: state evolving over time
  - \texttt{nbr} — Neighbouring: get information from neighbours
  - \texttt{if} — Restriction: a space-time branch

Other facts

- Syntax is LISP-like, in fact, a core of Proto [MIT Proto, 2012]
- Will use MIT Proto implementation for demoing
- Nodes all execute the same program, in asynchronous rounds
- Env. information (topology, sensors) extracted by built-in functions
Syntax

\[
e ::= \begin{array}{ll}
\text{(field) expression:} & \\
1 & \text{local value (boolean, float, tuple ...)} \\
| & \\
x & \text{variable} \\
| & \\
(b \ e_1 \ e_2 \ldots \ e_n) & \text{functional composition (b is built-in)} \\
| & \\
(f \ e_1 \ e_2 \ldots \ e_n) & \text{function call (f is user-defined)} \\
| & \\
(rep \ x \ l \ e) & \text{time evolution} \\
| & \\
(nbr \ e) & \text{neighbourhood field construction} \\
| & \\
(if \ e_b \ e_t \ e_f) & \text{restriction}
\end{array}
\]

\[
F ::= (\text{def } f(x_1 \ x_2 \ldots \ x_n) \ e) \quad \text{user-defined function}
\]
Functional composition: \((b \ e_1 \ldots \ e_n)\)

**Goal: extending standard computation mechanisms to whole fields**

- \(b\) is a built-in operator applied to \(e_1 \ldots e_n\) pointwise (node per node)
- A means to perform local operations: algorithms/sensing/acting
- They are pervasively used, shall use the following color

**Basic example**

- \((+ \ e_1 \ e_2)\): sums the two fields \(e_1\) and \(e_2\)
Built-in functions as environment bindings

Examples

- \((\text{red} \ 1)\) send value 1 to red actuator (a led)
- \((\text{sense} \ \text{"temperature"})\) gives a field of temperature values
- \((\text{sense} \ 1)\) gives a field of values from sensor n. 1
- \((\text{dt})\) gives the length of last round in each node
- \((\text{mid})\) gives the field of device identifiers
How do we deal with more articulated data types

- they are still seen as locals
- built-in functions are ADT operations, they can be used to create and operate on such complex values

Examples using tuples

- \((\text{tup } 10 \ 20)\) gives everywhere a tuple of two values, 10 and 20
- \((1\text{st } (\text{tup } 10 \ 20))\) accesses first component, i.e., 10
- \((2\text{nd } (\text{tup } 10 \ 20))\) accesses second component, i.e., 20
A peculiar data-type we handle is a neighbourhood field \( \phi \)

A map \( \{\sigma_1 \mapsto l_1, \ldots, \sigma_n \mapsto l_n\} \) from neighbourhood to some local value
- its domain never escapes a node’s neighbourhood
- it can’t be the final result of computation, just an intermediate one
- some special built-in sensor could return one such field (\( \text{nbr-star} \))
- some built-in function (\( \text{*-hood}, \text{*-hood+} \)) can turn it into a value

Examples using fields
- \( \text{nbr-range} \) gives a map \( \phi \) from neighbours to their distance
- \( \text{min-hood+ (nbr-range)} \) gives the minimum of neighbour distances (+ means “myself excluded”)
- \( < (\text{min-hood+ (nbr-range)})) 5 \) gives nodes with some short proximity
Function calls: \((f \ e_1 \ldots \ e_n)\)

**Goal:** providing standard name abstraction and recursive behaviour

- \(f\) is the name of a function to be defined as: \((\text{def} \ f(x_1 \ldots x_n) \ e_{body})\)
- call-by-value semantics, possibly with recursion
- for user-defined functions shall use the following color

**Example**

**Def** \((\text{def} \ double(x) \ (* \ x \ 2))\)

**Use** \((double (+ e1 e2))\): doubles the field \((+ e1 e2)\)
Syntax

\[ e ::= \]

1 \quad \text{(local value (boolean, float, tuple . . .))}

| x \quad \text{variable} |
| (b \ e_1 \ e_2 \ldots \ e_n) \quad \text{functional composition (b is built-in)} |
| (f \ e_1 \ e_2 \ldots \ e_n) \quad \text{function call (f is user-defined)} |
| (\text{rep} \ x \ l \ e) \quad \text{time evolution} |
| (\text{nbr} \ e) \quad \text{neighbourhood field construction} |
| (\text{if} \ e_b \ e_t \ e_f) \quad \text{restriction} |

\[ F ::= (\text{def} \ f(x_1 \ x_2 \ldots \ x_n) \ e) \quad \text{user-defined function} \]
Field construct n.1: Time evolution (\texttt{rep x 1 e})

**Goal: supporting field evolution over time**

- Initially it is globally 1 (or any expression..)
- At each new step a point in space updates to (local value of) \( e \)
- \( e \) can mention \( x \), which stands for “previous field (state)”

**Example**

- \((\texttt{rep } x \ 0 \ (+ \ x \ 1))\): counts the number of rounds in each device
- \((\texttt{rep } t \ 0 \ (+ \ t \ (dt)))\): the field of time
Field construct n.2: Neigh. field construction ($\text{nbr } e$)

Goal: enabling declarative node-to-node interaction

- The resulting field maps each node $n$ to a neighbourhood field $\phi_n$
- $\phi_n$ maps neighbours of $n$ to their local value of $e$
- Is to be flattened by a *-hood built-in operator

Example

- Assume $\text{bsns}$ be a special operator modelling a boolean sensor
- ($\text{any-}^{\text{hood}}(\text{nbr } (\text{bsns})))$: has any neighbour positive $\text{bsns}$?
Few field constructions examples

;;; the field of number of neighbours
(def count-neighbours () (sum-hood (nbr 1)))

;;; average distance of neighbours
(/ (sum-hood (nbr-range)) (count-neighbours))

;;; most connected neighbour’s id?
(def most-connected-neighbour ()
     (2nd (max-hood (nbr (tup (count-neighbours) (mid)))))
)

;;; what’s the highest connection in the network?
???
Neighbourhood chaining: nesting \texttt{nbr} into \texttt{rep}

\begin{verbatim}
;; what's the greatest value of F?
(def goss-max (F) (rep x F (max-hood (nbr x))))

;; which node is the highest connected?
(def most-connected-node ()
  (2nd (goss-max (tup (count-neighbours) (mid))))))

;; what's the minimum hop-count distance to (any) source?
(def hop-count (source)
  (rep x (inf) (mux source 0 (min-hood+ (+ 1 (nbr x))))))

;; distance-to, aka, gradient --> note it is robust and flexible!!
(def distance-to (source)
  (rep d (inf) (mux source 0 (min-hood+ (+ (nbr d) (nbr-range)))))
\end{verbatim}
Building a channel

;; broadcast, aka gradcast, broadcasting the value of \( f \) at \( src \)
(def broadcast (src f)
  (rep t (tup (inf) f)
    (mux src (tup 0 f)
      (min-hood+ (tup (+ (nbr-range) (1st (nbr t)))
        (2nd (nbr t)))))))

;; minimum distance between \( a \) and \( b \)
(def distance (a b)
  (2nd (broadcast b (distance-to a))))

;; channel between \( src \) and \( dest \) of thickness \( width \)
(def channel (src dest width)
  (< (+ (distance-to src) (distance-to dest))
    (+ width (distance src dest))))
Field construct n.3: Restriction \((\text{if } e_b e_t e_f)\)

**Goal:** providing a distributed branch

- \(e_b\) should be a boolean field (defining two restricted domains)
- Construct \(e_t\) where \(e_b\) is positive, and \(e_f\) where \(e_b\) is negative
- Not a mere superposition of \(e_t\) and \(e_f\), but a true domain restriction!
  - i.e., nbrs inside \(e_t\) should not escape into where \(e_b\) is false

**Example**

- \((\text{if } (\text{bsns}) e 0)\) creates \(e\) where \(\text{bsns}\) is true, and 0 elsewhere
Channel avoiding obstacles

;; automatically circumventing the obstacle
(def channel-avoiding-obstacle (obstacle source destination width)
  (if (not obstacle) (channel source destination width) 0))

;; mux applies the obstacle
;; but it does not interfere with channel computation
(def wrong-channel-avoiding-obstacles (obstacle source destination width)
  (mux (not obstacle) (channel source destination width) 0))
Channel in action – a simulation in Proto
Outline

1. Why a calculus of fields
2. The Field Calculus – “global viewpoint”
3. Semantic details – “local viewpoint”
4. Bootstrapping property verification: Self-stabilisation
Key aspects of the semantics: network model

**Network model**

- A node has a state $S$ (evaluation tree) updated at asynchronous rounds.
- At the end of the round, $S$ is spread to the (current) neighbourhood.
- State is updated “against” the neighbour states just received.

Nodes send their evaluation tree to neighbours at the end of each computation round.
Key aspects of the semantics: node (abstract) model

Inside a node: the evaluation tree $S$

- $S$ is an annotated version of the expression to evaluate
  - annotations are used to keep track of the next evaluation site
  - some annotations are persistent, and used to interact in space/time
- $S$ may dynamically expand due to (recursive) calls
- field constructs semantics impact shape of annotations
  - \texttt{rep}: last result is stored in an annotation and reminded at next round
  - \texttt{nbr}: observes annotations in the same position in neighbour trees
  - \texttt{if}: discards the neighbour trees which took a different branch

Main issue (and contribution wrt previous formalisation attempts)

Accommodate the interplay between \texttt{rep}, \texttt{if} and \texttt{nbr} – even in the presence of (recursive) calls
A node state is an annotated evaluation tree

Round 1  
\[(\text{rep } x \ 0 \ (\ + \ x \ 1)) \rightarrow (\text{rep } x \ 0 \ (\ + \ x \cdot 0 \ 1)) \rightarrow (\text{rep } x \ 0 \ (\ + \ x \cdot 0 \ 1 \cdot 1)) \rightarrow (\text{rep } x \ 0 \ (\ + \ x \cdot 0 \ 1 \cdot 1) \cdot 1) \rightarrow (\text{rep}^1 \ x \ 0 \ (\ + \ x \cdot 0 \ 1 \cdot 1) \cdot 1) \cdot 1\]
A node state is an annotated evaluation tree

Round 1  \[(\text{rep } x \ 0 \ (\ + \ x \ 1)) \rightarrow (\text{rep } x \ 0 \ (\ + \ x\cdot0 \ 1)) \rightarrow (\text{rep } x \ 0 \ (\ + \ x\cdot0 \ 1\cdot1)) \rightarrow (\text{rep } x \ 0 \ (\ + \ x\cdot0 \ 1\cdot1)\cdot1) \rightarrow (\text{rep}^1 \ x \ 0 \ (\ + \ x\cdot0 \ 1\cdot1)\cdot1)\cdot1\]

Round 2  \[(\text{rep}^1 \ x \ 0 \ (\ + \ x \ 1)) \rightarrow .. \rightarrow (\text{rep}^2 \ x \ 0 \ (\ + \ x\cdot1 \ 1\cdot1)\cdot2)\cdot2\]
A node state is an annotated evaluation tree

Round 1 \[(\text{rep} \ x \ 0 \ (+ \ x \ 1)) \rightarrow (\text{rep} \ x \ 0 \ (+ \ x \cdot 0 \ 1)) \rightarrow (\text{rep} \ x \ 0 \ (+ \ x \cdot 0 \ 1 \cdot 1)) \rightarrow (\text{rep} \ x \ 0 \ (+ \ x \cdot 0 \ 1 \cdot 1) \cdot 1) \rightarrow (\text{rep}^1 \ x \ 0 \ (+ \ x \cdot 0 \ 1 \cdot 1) \cdot 1) \cdot 1\]

Round 2 \[(\text{rep}^1 \ x \ 0 \ (+ \ x \ 1)) \rightarrow .. \rightarrow (\text{rep}^2 \ x \ 0 \ (+ \ x \cdot 1 \ 1 \cdot 1) \cdot 2) \cdot 2\]

Round 3 \[(\text{rep}^2 \ x \ 0 \ (+ \ x \ 1)) \rightarrow .. \rightarrow (\text{rep}^3 \ x \ 0 \ (+ \ x \cdot 2 \ 1 \cdot 1) \cdot 3) \cdot 3\]
A node state is an annotated evaluation tree

Round 1
(rep x 0 (+ x 1)) → (rep x 0 (+ x · 0 1)) →
(rep x 0 (+ x · 0 1 · 1)) → (rep x 0 (+ x · 0 1 · 1) · 1)
→ (rep x 0 (+ x · 0 1 · 1) · 1)

Round 2
(rep x 0 (+ x 1)) → .. → (rep x 0 (+ x · 1 1 · 1) · 2) · 2

Round 3
(rep x 0 (+ x 1)) → .. → (rep x 0 (+ x · 2 1 · 1) · 3) · 3
Evaluation aligns with neighbours to support nbr

Neighbour trees:

\[
\begin{align*}
\sigma_1 & \mapsto (\text{min-hood} (\text{nbr} (\text{sns}) \cdot 4)) \cdot \phi_1) \cdot 4 \\
\sigma_2 & \mapsto (\text{min-hood} (\text{nbr} (\text{sns}) \cdot 9)) \cdot \phi_2) \cdot 9 
\end{align*}
\]

Evaluation (assume here sns gives 7):

\[
\begin{align*}
(\text{min-hood} (\text{nbr} (\text{sns}))) & \mapsto \\
(\text{min-hood} (\text{nbr} (\text{sns}) \cdot 7)) & \mapsto \\
(\text{min-hood} (\text{nbr} (\text{sns}) \cdot 7) \cdot (\sigma \mapsto 7, \sigma_1 \mapsto 4, \sigma_2 \mapsto 9)) & \mapsto \\
(\text{min-hood} (\text{nbr} (\text{sns}) \cdot 7) \cdot (\sigma \mapsto 7, \sigma_1 \mapsto 4, \sigma_2 \mapsto 9)) \cdot 4
\end{align*}
\]
Evaluation aligns with neighbours to support $\text{nbr}$

**Neighbour trees:**

\[
\sigma_1 \mapsto (\text{min-hood} (\text{nbr} (\text{sns}) \cdot 4) \cdot \phi_1) \cdot 4 \\
\sigma_2 \mapsto (\text{min-hood} (\text{nbr} (\text{sns}) \cdot 9) \cdot \phi_2) \cdot 9
\]

**Evaluation (assume here sns gives 7):**

\[
(\text{min-hood} (\text{nbr} (\text{sns}))) \rightarrow \\
(\text{min-hood} (\text{nbr} (\text{sns}) \cdot 7)) \rightarrow \\
(\text{min-hood} (\text{nbr} (\text{sns}) \cdot 7) \cdot (\sigma \mapsto 7, \sigma_1 \mapsto 4, \sigma_2 \mapsto 9)) \rightarrow \\
(\text{min-hood} (\text{nbr} (\text{sns}) \cdot 7) \cdot (\sigma \mapsto 7, \sigma_1 \mapsto 4, \sigma_2 \mapsto 9)) \cdot 4
\]
Restriction prevents escaping a domain

Neighbour trees:

\[ \sigma_1 \mapsto (\text{if } (\text{bsns}) \cdot f (\text{min-hood} (\text{nbr}(\text{sns}))) \cdot 0 \cdot 0) \cdot 0 \]

\[ \sigma_2 \mapsto (\text{if } (\text{bsns}) \cdot t (\text{min-hood} (\text{nbr}(\text{sns}) \cdot 9 \cdot \phi_2) \cdot 0)) \cdot 9 \]

Evaluation (assume here sns gives 7 and bsns gives t):

\[ (\text{if } (\text{bsns}) (\text{min-hood} (\text{nbr} (\text{sns}))) \cdot 0) \mapsto \]

\[ (\text{if } (\text{bsns}) \cdot t (\text{min-hood} (\text{nbr} (\text{sns}))) \cdot 0) \mapsto \ldots \mapsto \]

\[ (\text{if } (\text{bsns}) \cdot t (\text{min-hood} (\text{nbr} (\text{sns}) \cdot 7) (\sigma \mapsto 7, \sigma_2 \mapsto 9)) \cdot 7 \cdot 0) \]
Restriction prevents escaping a domain

Neighbour trees:

\[
\sigma_1 \mapsto (\text{if } (\text{bsns}) \cdot f (\text{min-hood (nbr (sns)))) 0 \cdot 0) \cdot 0
\]

\[
\sigma_2 \mapsto (\text{if } (\text{bsns}) \cdot t (\text{min-hood (nbr (sns) \cdot 9)} \cdot \phi_2 0)) \cdot 9
\]

Evaluation (assume here sns gives 7 and bsns gives t):

\[
(\text{if } (\text{bsns}) (\text{min-hood (nbr (sns)))) 0) \rightarrow \ldots \rightarrow (\text{if } (\text{bsns}) \cdot t (\text{min-hood (nbr (sns) \cdot 7)})(\sigma \mapsto 7, \sigma_2 \mapsto 9)) \cdot 7 0)
\]

Recursively, neighbour trees that are not aligned are temporarily discarded.
Function calls dynamically expand the evaluation tree

Take definition \( \text{def fun (x) (min-hood (nbr x))} \).

When evaluating \((\text{if (bsns) (fun sns) 0})\), function body is expanded:

\[
\begin{align*}
(\text{if (bsns) (fun sns) 0}) & \rightarrow (\text{if (bsns) \cdot t (fun sns) 0}) \\
(\text{if (bsns) \cdot t (fun sns) 0}) & \rightarrow \\
(\text{if (bsns) \cdot t (fun (min-hood (nbr x)) sns) 0}) & \rightarrow \\
(\text{if (bsns) \cdot t (fun (min-hood (nbr x) sns) 7) 0}) & \rightarrow \\
... \\
\end{align*}
\]
Function calls dynamically expand the evaluation tree

Take definition (\texttt{def fun(x) (min-hood (nbr x)))..}

When evaluating \((\texttt{if (bsns) (fun sns) 0)})\), function body is expanded:

\[
\begin{align*}
(\texttt{if (bsns) (fun sns) 0}) & \rightarrow (\texttt{if (bsns) \cdot t (fun (sns)) 0}) \\
(\texttt{if (bsns) \cdot t (fun (sns) \cdot 7) 0}) & \rightarrow \\
(\texttt{if (bsns) \cdot t (fun}^{\texttt{min-hood (nbr x)}} (\texttt{sns) \cdot 7}) 0) & \rightarrow \\
(\texttt{if (bsns) \cdot t (fun}^{\texttt{min-hood (nbr x \cdot 7)}} (\texttt{sns) \cdot 7}) 0) & \rightarrow \ldots
\end{align*}
\]

Expansion at function call, contraction on non-taken branches
## The pillars of the operational semantics

### Elements

- A node’s computation is about annotating the evaluation tree
- Such a tree is spread to neighbours after being cleaned up
- Neighbour trees affect evaluation
- Annotations to **nbr**
  - persist over “space” (sent to neighbours)
  - used to remotely reconstruct a neighbourhood field at **nbr** sites
- Annotations to **if**
  - persist over “space” (sent to neighbours)
  - used to filter out neighbours that do not match at **if** sites
- Annotations to **rep**
  - persist over “time” (remembered at next round)
  - used to recompute at **rep** sites
Tree evaluation: pictorial semantics

Neighbour trees

Evaluation tree

rep
if
nbr fun
Tree evaluation: pictorial semantics

Neighbour trees

Evaluation tree

domain restriction

time persistence

rep

if

nbr

fun

neighbour alignment

tree expansion
### The operational semantics of node model

**Runtime Expression Syntax:**

- $e ::= \alpha \overline{\nu}$  
  - runtime expression (rte)
- $a ::= x \mid v \mid \text{nbr } e \mid \text{if } e \text{ or } e' \mid \text{rep } x \text{ we} \mid \text{f } \overline{\nu}$  
  - auxiliary rte
- $v ::= 1 \mid \phi$  
  - runtime value
- $s ::= \overline{\mu}$  
  - superscript
- $w ::= x \mid 1$  
  - variable or local value
- $\phi ::= \overline{\sigma} \mapsto 1$  
  - field value
- $\Theta ::= \overline{\sigma} \mapsto \overline{e}$  
  - tree environment
- $\Gamma ::= \overline{x} ::= \overline{v}$  
  - variable environment

**Congruence Contexts:**

- $\mathbb{C} ::= (\text{nbr } []) \mid (\text{f } \overline{\nu} \mid \overline{e}) \mid (\text{o } \overline{\nu} \mid \overline{e}) \mid (\text{if } [] \text{ e} \mid \text{if } a \text{ t } \overline{e} \mid \text{if } a \text{ f } \overline{e} [])$

**Alignment contexts:**

- $\mathbb{A} ::= \mathbb{C} \mid (\text{rep } x \text{ w } []) \mid (\text{f } \overline{a} \overline{v})$

**Auxiliary functions:**

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

**Reduction Rules:**

- **[VAL]** \(\Theta; \Gamma \vdash v \rightarrow \nu \overline{v}\)
- **[VAR]** \(\Theta; \Gamma \vdash x \rightarrow x \Gamma(x)_{\text{dom}(\Theta), \text{e}(\text{self})}\)
- **[NBR]** \(\Theta; \Gamma \vdash \text{nbr } a \overline{l} \rightarrow \text{nbr } a \overline{l} \phi\)
- **[OP]** \(\Theta; \Gamma \vdash (o \overline{a} \overline{v}) \rightarrow (o \overline{a} \overline{v}) \text{e}(o, \overline{v})\)
- **[CONG]** \(\pi_{\mathbb{C}}(\Theta); \Gamma \vdash a \rightarrow e\)
- **[REP]** \(\pi_{\text{rep } x \text{ w } []}(\Theta); \Gamma \vdash (\text{f } \overline{a} \overline{v}) (\text{args } (f) := \overline{v}) \rightarrow (\text{body } (f) \triangleright s) \rightarrow a \overline{v}\)
- **[FUN]** \(\Theta; \Gamma \vdash (\text{f } \overline{a} \overline{v}) \rightarrow (\text{f } \overline{a} \overline{v}) \phi\)

- **[THEN]** \(\Theta; \Gamma \vdash (\text{if } a \text{ t } a \text{ t } e \rightarrow (\text{if } a \text{ t } a \text{ t } e) \overline{1})\)
- **[ELSE]** \(\Theta; \Gamma \vdash (\text{if } a \text{ f } e \overline{a} \rightarrow (\text{if } a \text{ f } e \overline{a}) \overline{1})\)
Outline

1. Why a calculus of fields
2. The Field Calculus – “global viewpoint”
3. Semantic details – “local viewpoint”
4. Bootstrapping property verification: Self-stabilisation
What is the field calculus good at?

- Giving feedbacks on existing/new models/implementations
- Helping designing new features (e.g., typing, higher-order functions)
- Assisting development of surface languages, APIs
- Predicting the behaviour of fragments of the language
What is the field calculus good at?

- Giving feedbacks on existing/new models/implementations
- Helping designing new features (e.g., typing, higher-order functions)
- Assisting development of surface languages, APIs
- Predicting the behaviour of fragments of the language

History and Roadmap

- PAST: Started in SAPERE EU FP7 project [Zambonelli et al., 2011]
  - self-stabilisation for a rule-based field language
- PRESENT: A self-stabilisation result for a fragment of the field calculus achieved [Viroli and Damiani, 2014]
- PRESENT: Universality of field calculus [Beal et al., 2014]
- ONGOING: Density-independence / approximability
- FUTURE: Extending the fragments of investigation
Self-stabilisation

**Traditional self-stabilisation [Dolev, 2000]**

A distributed system that is self-stabilising will end up in a *correct* state (in finite steps) no matter what state it is initialised with.

**Superstabilisation**

A distributed system is superstabilising if it is self-stabilising and it recovers *fast* from (single) topological changes.

**Considerations for the field-calculus**

- Self-stabilisation is of course a key fault-tolerant property
- The definition should somehow be adapted to work with fields
- One still needs to (implicitly) define what are the correct states
- Knowing that (and which) correct states are reached is indeed about predictability!
- What about: (def flip (a b) (rep x a (- (+ a b) x)))
We considered a notion of \textit{(unique, super-)} self-stabilisation

If a field expression is self-stabilising, then it necessarily reaches a unique configuration (unique fixpoint), recovering from \textit{any} change

- assume execution of rounds is asynchronous but \textit{fair}
- fixpoint is independent of initial state
- fixpoint is reached in finite time
- fixpoint depends on environment \( E \) (sensors/topology)
- if \( E \) (arbitrarily) changes the system recovers

\textbf{Implication}

Any self-stabilising field expression \( e \) has denotational semantics:

\[
\phi_{\text{final}} = \phi_{E}(e)
\]

..i.e., a program \( e \) has predictable global/final outcome \( \phi_{\text{final}} \)
Sufficient conditions: a self-stabilising fragment

Ideas

- Generalise over a key block, the hop-count distance (gradient):

  \[(\text{rep } x (\text{inf}) (\text{mux } \text{source } 0 (\text{min-\text{hood}+ (+ 1 (\text{nbr } x))}))))\]

- \text{nbr} should stay inside a \text{*-hood}, itself inside a \text{rep}
- \text{*-hood} should implement an order-independent function (e.g. \text{min})
- Should generalise over the \text{“+ 1”} function
- Should guarantee that propagation terminates
- Strict layering of self-stabilising fields is self-stabilising
Some preliminaries

Rework a distance-to (gradient) specification

⇒ (rep x (inf) (mux source 0 (min-hood+ (+ 1 (nbr x)))))
  ● .. assume source is 0 on sources, and ∞ elsewhere
⇒ (rep x (inf) (min source (min-hood+ (+ 1 (nbr x)))))
  ● .. generalise over function +, which could also use sensor-dependent expressions
⇒ (def spr-aggr (E0 FUN E1 .. En)
  (rep x (inf)
    (min E0 (FUN (min-hood+ x) E1 .. En))))
  ● .. abstract this block into a syntactic construct
⇒ {e0 : g(@, e1, ..., en)}
The Syntax – 3 main ingredients

| e ::= x | variable   |
| v     | value      |
| s     | sensor     |
| g(e₁, ⋯, eₙ) | pointwise functional composition |
| {e : g(@, e₁, ⋯, eₙ)} | spreading-aggregation (nbr, rep, min-hood+) |

| g ::= f | user-defined function |
| b     | built-in function |

Additional constraints

- User-defined function definitions have no cycles
- Built-in functions are environment independent
- Basic typing properties are satisfied
A function $g$ over type $T$ is a *stabilising progression* if:

1. $g$ is a *pure operator* — it calls no sensor or spreading-aggregation;
2. $T$ is *locally noetherian* — $[T]$ is equipped with a total order relation $\leq_T$, and for every element $v \in [T]$, there are no infinite ascending chains of elements $v_0 <_T v_1 <_T v_2 \cdots$ such that (for every $n \geq 0$) $v_n <_T v$;
3. $g$ is *monotone* in its first argument — $v \leq_T v'$ implies $[g](v, \overline{v}) \leq_T [g](v', \overline{v})$ for any $\overline{v}$;
4. $g$ is *progressive* in its first argument — $v <_T [g](v, \overline{v})$ (for $v \neq \text{top}(T)$).

Self-stabilisation result

A field expression whose spreading aggregation constructs feature only self-stabilising progressions is self-stabilising.
def int distanceto(int i) is \{ i : \emptyset + \#dist \}
Distance-to circumventing an obstacle

\[
\begin{align*}
def \text{int} & \ \text{distanceto}(\text{int } i) \ \text{is} \ \{ \ i : \od + \#\text{dist} \} \\
def \text{int} & \ \text{restrict}(\text{int } i, \ \text{bool } b) \ \text{is} \ b ? i : \ \text{INF} \\
def \text{int} & \ \text{distobs}(\text{int } i, \ \text{bool } b) \ \text{is} \ \{ \ i : \ \text{restrict}(\od + \#\text{dist}, \ b) \} \\
\end{align*}
\]
def int distanceto(int i) is { i : @ + #dist }
def int restrict(int i, bool b) is b ? i : INF
def int distobs(int i, bool b) is { i : restrict(@ + #dist, b) }
def <int,int> add_to_1st(<int,int> x, int y) is <1st(x)+ y, 2nd(x)>
def <int,int> broadcast(int i, int j) is { <i, j>: add_to_1st(@, #dist) }
def int dist(int i, int j) is broadcast(restrict(j,j==0),distanceto(i))
def bool path(int i, int j, int w) is
distanceto(i) + distanceto(j) < dist(i,j) + w
def int channel(int i, int j, int w) is
broadcast(distanceto(j),not path(i,j,w))
The proof

Structure of the proof

- Because of layering can reason inductively on expression structure
- The only non-trivial case is to prove that \( \{v_0 : g(@, v_1, \ldots, v_n)\} \) self-stabilises (\( v_i \) are immutable fields)

Key idea

Inductively on the size of the subnetwork \( S \) that already self-stabilised

- Base case: the node holding the minimum value of \( v_0 \) self-stabilises immediately (at its first round)
- Inductive case: after a small transient, the minimum value outside \( S \) necessarily increases until \( S \neq \emptyset \), so there’s surely another node that will self-stabilise:
  - either one reaching \( \text{top}(T) \),
  - or one in the neighbourhood of a node in \( S \)
... in both cases the result is independent of the initial state
Non-self-stabilising fields

;; previous definition
(def goss-max (F) (rep x F (max-hood (nbr x))))

;; non-self-stabilising
(goss-max (sense 1))

;; still not working
(def goss-max2 (F) (rep x F (max-hood (nbr x))))

;; where’s the bug?
(def wrong-distance-to (SRC)
  (rep x (inf) (mux SRC 0 (min-hood (+ (nbr-range) (nbr x)))))
)
Conclusions

Other predictability studies (in the following)
- Eventual consistency, i.e., network density independence
- Universality of field calculus constructs

Open issues
- There are more self-stabilising fields than our condition can check
- The notion of self-stabilisation can be generalised
- What about correct dynamic fields?
- There’s room for some more foundational work
- You do not like the language itself? Others are under study.
*The Lambda Calculus.*
North Holland, revised edition.

Organizing the aggregate: Languages for spatial computing.

Towards a unified model of spatial computing.

*Self-Stabilization.*
MIT Press.

Featherweight Java: A minimal core calculus for Java and GJ.

*Communicating and Mobile Systems: The π-calculus.*
Cambridge University Press.

MIT Proto.
A calculus of self-stabilising computational fields.
In eva Kühn and Pugliese, R., editors, Coordination Languages and Models, volume 8459 of LNCS, pages 163–178.
Springer-Verlag.
Proceedings of the 16th Conference on Coordination Models and Languages (Coordination 2014), Berlin (Germany), 3-5 June.

A calculus of computational fields.

Self-aware pervasive service ecosystems.
Procedia CS, 7.