Run-Time management of computation domains in field calculus

Run-time Management of Computation Domains in Field Calculus

Giorgio Audrito*, Ferruccio Damiani*, Mirko Viroli† and Roberto Casadei†
*Computer Science Department, University of Torino
Email: {giorgio.audrito, ferruccio.damiani}@unito.it
†DISI Department, University of Bologna
Email: mirko.viroli@unibo.it, roberto.casadei12@studio.unibo.it

***Abstract*—The field calculus is proposed as a foundational model for collective adaptive systems, capturing in a tiny language essential aspects of distributed interaction, restriction and evolution, as well as providing ground for engineering resiliency properties. In this paper, we investigate the interplay between interaction and restriction: known as "domain alignment" in field calculus, it is extremely powerful but can cause subtle bugs when not handled properly. We propose a disciplined programming approach based on the interplay between a weak and a strong version of alignment, mixing static and dynamic checks. This is exemplified to design a new reusable component dynamically updating the strategy by which a device can extract information from neighbours, which find applications, for instance, in the on-the-fly evolution of metrics in smart mobility applications.**

***Index Terms*—aggregate programming; computational field; dynamic software updating; formal properties;**

I. Introduction

In scenarios such as the Internet-of-Things, pervasive computing, and swarm robotics, the increasing density of computing devices we are witnessing make Collective Adaptive Systems (CASs) become a more and more interesting approach to face the challenge of scalability and adaptivity. However, it is often difficult to find a good trade-off between the need of building complex strategies to coordinate CASs, and the availability of engineering tools (languages, platforms) providing formal guarantees of functional and non-functional correctness. Traditionally, computer science addresses the problem by foundation calculi: λ-calculus [1] for functional programming, π-calculus [2] for interactive processes, Featherweight Java [2] for object-oriented programming, and the Linda calculus [3] or Klaim [4] for coordination models based on shared-spaces: they all bring few key mechanisms of their respective paradigm into a very small language (or calculus), over which several static and dynamic properties can be investigated—just to mention one, type-checking of Java generics in Java 5 has been implemented after FGJ extensions to FJ [5].

What would be a suitable core calculus for CASs? Recent works propose the *field calculus* [6], [7], a minimal model describing computations of CASs in terms of global-level manipulation of a collective data-structure, called *computational field* [8] (or "field" for short). A field is a time-varying map from devices to computational values, it is physically deployed in the computational environment, and can be used to model data produced by distributed sensors or feeding distributed actuators, intermediate results of distributed computations, and so on. The field calculus is at the root of the *aggregate computing* approach [9], which promotes the construction of reusable blocks of collective adaptive behaviour as field-to-field computations, and whole complex applications as layers of APIs constructed out of such blocks. Recently, the field calculus has been successfully exploited to investigate properties of self-stabilisation [10], device distribution independence [11] and universality [12], and to build programming languages on top, such as Protelis [13].

This paper follows the direction of such previous works, and investigates the subtleties of the construct of field calculus dealing with device-to-device interactions, operator `nbr`. It essentially specifies an observation mechanism by which a device gathers a map from neighbours to their value of a given intermediate computation: such "observation maps" (also called field values) are first-class data values, and are amenable to combinations with others, and to reduction to single data values. Crucially to the goals of aggregate computing, this mechanism is fully declarative in the sense that it supports compositionality of behaviour (interactions correctly occur even when `nbr` is nested into deep levels of functional composition) and restriction (`nbr` creates maps considering only the neighbour devices that executed the same intermediate program). One of the most basic and crucial safety features of `nbr` is *domain alignment*, already introduced in [7], [14], which ensures that observation maps are always "domain coherent" with each other, and hence no unwanted interactions between devices occur. However, checking domain alignment at compile-time is a complex task: it requires extensive coding effort, and can only be achieved at the expense of admitting some domain-aligned expressions to be refuted by the system. Often, when not treated carefully, this problem results in subtle bugs.

In this paper we first technically examine two possible strategies for run-time management of field domains: *Strict* (strong domain alignment), throwing exceptions on domain mismatch, ensuring run-time domain alignment; and *Permis-*

This work is partially supported by project HyVar which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 644298, by ICT COST Action IC1402 ARVI, and by Ateneo/CSP project RunVar.

P	::=	$\overline{\texttt{F}}$ e	program
F	::=	def d($\overline{\texttt{x}}$) {e}	function declaration
e	::=	x $\mid$ ϕ $\mid$ c($\overline{\texttt{e}}$) $\mid$ b $\mid$ d $\mid$ ($\overline{\texttt{x}}$) =>e	
		$\mid$ e($\overline{\texttt{e}}$) $\mid$ nbr{e} $\mid$ rep(e){e}	expression

Fig. 1. Syntax of field calculus.

sive (weak domain alignment), allowing fields with different domains to be combined by restricting them to the largest common domain (i.e. the intersection of their domains). Then, after examining several examples, we show a programming methodology that mixes these two strategies (strictness as default, and permissiveness on-demand) and a disciplined programming approach to selectively avoid alignment bugs and exploit the full potentials of nbr. As a final result, we put this approach in practice and develop a new building block called "updatable metric": during a system operation, it allows one to dynamically update the logical (metric) structure of the network so as to tweak system behaviour: for instance, one could stretch the notion of distance into a system to navigate people in complex environments, to take into account aspects such as crowds, traffic jams, pollution, and favourable areas.

The remainder of this paper is organised as follows: Section II presents the syntax and operational semantics of field calculus; Section III introduces and examines the definitions of *weak* and *strong* domain alignment and the restriction property, Section IV presents examples clarifying the advantages and disadvantages of strong domain alignment, Section V introduces the new *updatable metric* building block, and Section VI summarises the guidelines that emerge from the cases studied.

II. Field Calculus

Figure 1 presents the syntax of the field calculus in its higher-order version—we refer to [7], [13] for a more detailed presentation with examples. Following [5], the overbar notation denotes metavariables over sequences and the empty sequence is denoted by $\bullet$. E.g., for expressions, we let $\overline{\texttt{e}}$ range over sequences of expressions, written $\texttt{e}_1, \texttt{e}_2, \ldots \texttt{e}_n$ $(n \geq 0)$.

A program P consists of a sequence of function declarations and of a main expression e. A function declaration F defines a (possibly recursive) function, where d is the name, $\overline{\texttt{x}}$ are the parameters and e is the body. Expressions e are the main entities of the calculus, modelling a whole field (that is, an expression e evaluates to a value on every device in the network, thus producing a computational field). An expression can be:

- a variable x, used as function formal parameter;
- a field value ϕ (which is allowed to appear in intermediate computations but not in source programs);
- a data constructor c($\overline{\texttt{e}}$);
- a built-in function b;
- a declared function name d;
- an anonymous function ($\overline{\texttt{x}}$) =>e (where $\overline{\texttt{x}}$ are the formal parameters and e is the body);
- a function call e($\overline{\texttt{e}}$) (which is computed separately in clusters i.e. regions of space holding the same result for the functional expression e);
- a nbr-expression nbr{e}, modelling neighbourhood interaction via extraction of "observation maps";
- or a rep-expression rep(e){e}, modelling time evolution.

Figure 2 introduces the big-step operational semantics of field calculus, according to the "local" viewpoint. Devices undergo computation in fair and asynchronous rounds, performing evaluation of the program with respect to the messages recently received from neighbours, and broadcasting the outcome of the computation to neighbours for later use. The result of evaluation is a *value-tree* $\theta ::= \texttt{v}\langle\overline{\theta}\rangle$, which is an ordered tree of values tracking the result of any evaluated subexpression. The value-trees recently received from neighbours are collected together into a *value-tree environment* $\Theta ::= \overline{\delta} \mapsto \overline{\theta}$; which is navigated pointwise by the auxiliary functions $\pi_i(\Theta)$ (which selects the i-th branch corresponding to the relevant subexpression) and $\rho(\Theta)$ (which selects the outcome of the computation).

We use $\delta; \Theta \vdash \texttt{e} \Downarrow \theta$ to mean "expression e evaluates to value-tree θ on device δ with respect to the value-tree environment Θ", and $\delta; \overline{\pi}(\Theta) \vdash \overline{\texttt{e}} \Downarrow \overline{\theta}$ as a shorthand for $\delta; \pi_1(\Theta) \vdash \texttt{e}_1 \Downarrow \theta_1 \cdots \delta; \pi_n(\Theta) \vdash \texttt{e}_n \Downarrow \theta_n$.

Rule [E-LOC] and [E-FLD] model evaluation of local and field values. Rule [E-B-APP] and [E-D-APP] model application of built-in and user defined functions; where built-in application is abstracted away via the auxiliary function $(\!|\texttt{b}|\!)^{\Theta}_{\delta}$. Rule [E-REP] models state preservation and through repeated updates, and [E-NBR] models field construction from the values in the value-tree environment.

Each rule is designed to grant that the environment Θ corresponds to the expression currently being evaluated, by iteratively matching each subexpression with the corresponding branch in the value-trees. Two special precautions are taken in order to ensure that no interference between different subexpressions can happen:

- Rule [E-FLD] reduces the domain of a field ϕ (which might have been computed in a larger context) to the set of devices currently available in Θ;
- Rule [E-D-APP] reduces the value-tree environment Θ (for the evaluation of the actual application) to only those devices who applied the same function f in their last evaluation round, so that no matching is attempted between different expressions.

Together, these rules ensure that any field created through nbr{e} consists only of values that neighbours have evaluated for e, and not for other subexpressions e′.

III. Domain Alignment

Rule [E-D-APP] of field calculus together with the progressive alignment through $\pi_i(\Theta)$ ensures that fields created through nbr{e} consists only of values that neighbours have evaluated for the same subexpression e. Notice that for this property to

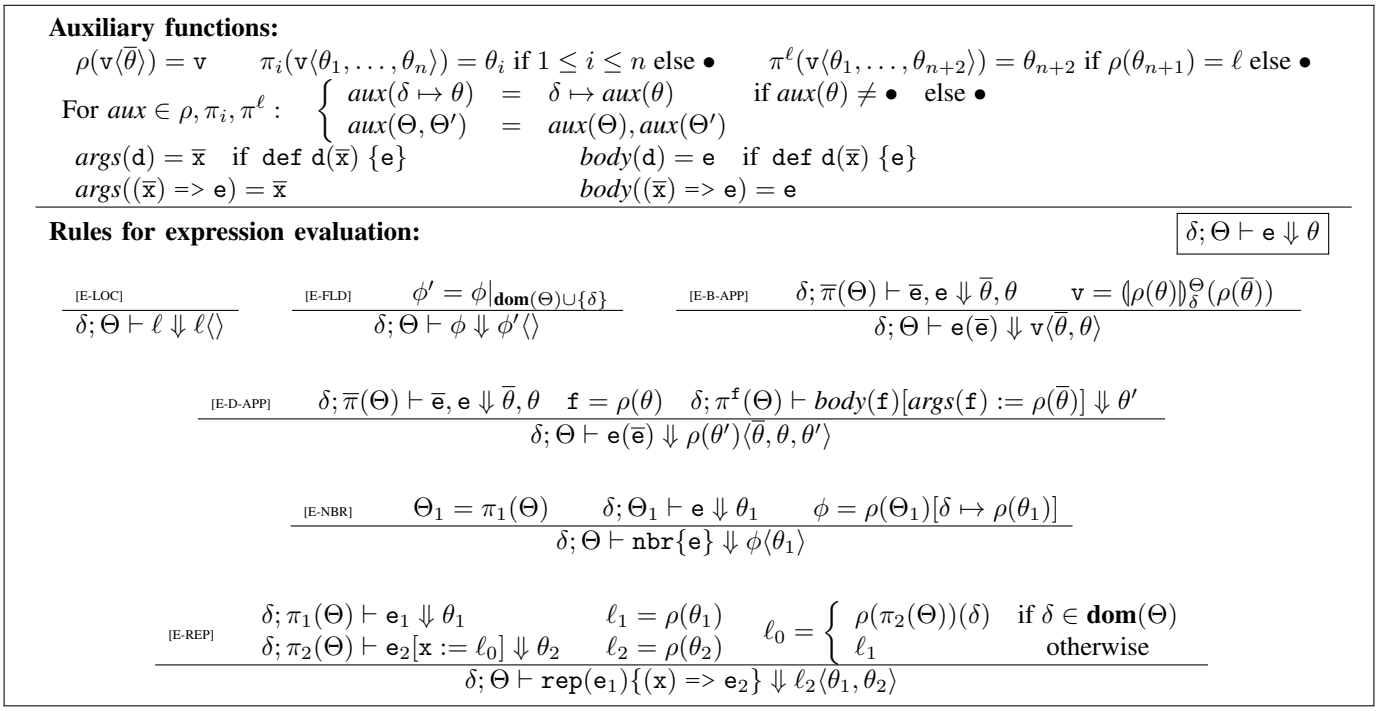

Fig. 2. Big-step operational semantics for expression evaluation

hold we don't need rule [E-FLD] to (possibly) reduce the domain of field values: however, this fact is needed in order to ensure the following property.

Definition 1. *We say that* strong domain alignment[1] *holds iff whenever a subexpression* e *evaluates in a device δ to a field value ϕ, **dom**(ϕ) is equal to the set S formed of δ and all of its neighbours which evaluated the same subexpression* e *in their last evaluation round.*

We say that weak domain alignment *holds iff **dom**(ϕ) is any subset of S containing δ.*

Weak domain alignment is enforced by construction by the evaluation rules of field calculus, thanks to rule [E-FLD] which ensures that any field value is first restricted to fit into the correct domain (together with the assumption that built-in operators return fields with domain included in the set **dom**(Θ) of aligned devices[2]).

Strong domain alignment could be statically enforced by advanced typing rules. However, the problem of determining exactly which expressions are guaranteed to comply with it is in general not decidable, so some degree of approximation is needed.

For example, consider an expression $\mathtt{e}(\overline{\mathtt{e}})$ where e is a functional expression of type $\overline{T} \to F$, where F is a field type (i.e., holding an observation map), and assume that e evaluates to a user-defined function f. By rule [E-D-APP], the function application is performed in the restricted environment of the devices which evaluated e to the same value f. Thus by weak domain alignment the field ϕ which is the result of $\mathtt{e}(\overline{\mathtt{e}})$ has domain contained in the same restricted set of devices. It follows that $\mathtt{e}(\overline{\mathtt{e}})$ complies with strong domain alignment whenever the expression e always evaluates to a same value on neighbour devices; and such property cannot be statically checked by a type system.

It follows that strong domain alignment can be enforced *without disallowing* some unrecognised compliant expressions only through dynamic run-time checks. We remark that this is in fact the practical approach that is taken in most existing implementations of field calculus (e.g. Protelis [13]). Domain alignment is also justified and supported by the following related property.

Definition 2. *We say that the* restriction property *holds iff whenever a (partially evaluated) subexpression* e *occurring at some stage of the computation is evaluated, the final outcome* v *is not influenced by the devices who did not evaluate the subexpression* e *in their last evaluation round.*

This property ensures predictable and correct composition of expressions. In particular, it allows to *(i)* evaluate an expression e in the restricted environment where a certain boolean condition b holds, via the construct if(b){e}{$\mathtt{e}_0$};[3] *(ii)* guarantee that fields arguments passed to a function f do not contain values for devices not accessible inside the function through nbr.

In other words, the restriction property is a feature of the language both allowing convenient expressibility of domain

[1] In [7], strong domain alignment is referred to as *domain alignment*.

[2] Which (by induction) is always a superset of the domain of any input fields passed to the built-in operator.

[3] In field calculus the if construct is defined in terms of mux and functional application, thus it is not included in Figure 1.

restrictions and ensuring (partly) predictable domains for field arguments. We can thus argue that the restriction property is a reasonable requirement for any safe and modular aggregate computing language. Interestingly enough, such property follows from weak domain alignment.

Theorem 1. *The restriction property follows from weak domain alignment.*

Proof sketch. Let `e` be a partially evaluated expression occurring at some stage of the computation. By weak domain alignment, any field subexpression of `e` has domain contained into the set of neighbours which computed `e`. Since any subexpression `e′` of `e` can be influenced by neighbours only via field values, the final outcome of `e` is not influenced by non-aligned neighbours. □

Since weak domain alignment is granted by field calculus alone (without static advanced type checks), we can conclude that the restriction property is also granted.

IV. Alignment strategies by examples

We now compare two possible implementations of field calculus based on run-time management of field domains:

- *Strict*: throwing exceptions on domain mismatch, ensuring run-time domain alignment;
- *Permissive*: allowing fields with different domains to be combined by restricting them to the largest common domain (i.e. the intersection of their domains).

In both cases, weak domain alignment and restriction property will be granted. In the strict case, strong domain alignment will also be checked at run-time, while in the permissive case fields will be combined by incrementally decreasing their domain. Thus in the permissive case there is no guarantee for a field value to contain values other than at the current device. The domain of such a field expression `e` can then be understood as the set of devices that both *(i)* computed `e` at their last evaluation round; *(ii)* computed the same subexpressions of `e` of field type.

We now present four examples aimed at showing that a strict approach is correctly able to rule out many incorrect computations, while also forbidding few important correct computations.

A. Average: Unpredictability of Composition

Consider the following function with a field argument:

```
def avg(x) { sum-hood(x)/sum-hood(nbr{1}) }
```

If `avg` is executed while ensuring strong domain alignment, `sum-hood(nbr{1})` gives 1 for each aligned neighbour and then sums them, hence it counts the number of aligned neighbours which is also equal to the size of the domain of field `x`. If instead `avg` is executed in a permissive environment, it will fail to correctly compute the average of `x` whenever `x` has a domain which is not as large as possible. A formulation of `avg` which is correct in a permissive setting is possible but definitely more cumbersome:

```
def avg2(x) { sum-hood(x)/sum-hood(0*x + nbr{1}) }
```

In this case, $0 \cdot$`x` is a constantly zero-valued field with the same domain as `x`, which added with `nbr{1}` produces a 1-valued field with the same domain. This example clearly shows that it is often convenient for a programmer to assume strong domain alignment in order to more easily produce correct programs.

B. Parametric Gradient: Vanishing of Domains

Consider the following function:

```
def G(source, metric) {
  rep (inf) {
    (dist) => mux( source, 0,
      min-hood(metric()+nbr{dist}) )
  }
}
```

Given a function returning fields `metric` and a boolean value `source`, function `G` computes the *gradient* [9], that is, the minimum distance from the current device to a source device computed through `metric` (a function returning an observation of estimated distances from neighbours).

In an environment with strong domain alignment, the field ϕ returned by `metric` will assign a distance to *every* neighbouring device. With an implementation of `metric` which violates strong domain alignment, however, ϕ might not be assigning a distance to any device, thus effectively disconnecting the network in unpredictable ways.

C. Combined Restriction: Code Redundancy

The restriction property ensures that a field parameter `x` can be easily restricted (before being processed by a function `f`[4]) on the subset of devices which agree on a certain boolean expression `b` via `if(b){f(x)}{f(x)}`. This "restriction operator" is in fact a common building block in aggregate programming [9]. However, if we want to restrict `x` to those devices which agree on a whole sequence of boolean expressions $\mathtt{b}_1, \ldots, \mathtt{b}_n$ in a strong domain aligned environment we need a program $\mathtt{e}_{\mathtt{main}}$ as the following:

```
if (b1 and b2 and ... bn) {f(x)}
 elif (not b1 and b2 and ... bn) {f(x)}
  elif (b1 and not b2 and ... bn) {f(x)} ...
```

whose size is $O(2^n)$ and which might be infeasible even for small values of n. On the other hand, if strong domain alignment is dropped the above program can be more concisely written as $\mathtt{e}'_{\mathtt{main}}$:

```
f(x + if (b1) {nbr{0}} {nbr{0}}
   + if (b2) {nbr{0}} {nbr{0}} ...)
```

whose size is $O(n)$. In this case, the domain of the i-th 0-valued field will be equal to the set of devices which agrees on $\mathtt{b}_i$; thus by intersecting all of them the resulting domain will be equal to the set of devices which computed the same values for each $\mathtt{b}_1, \ldots, \mathtt{b}_n$.

[4]Restricted fields cannot be directly returned by an expression without violating strong domain alignment: they first need to be processed into local values.

This rather abstract example might be concretely instantiated e.g. in case a function needs to be executed separately on devices with different configurations. Even though the number of possible configurations is exponential, $e'_{\texttt{main}}$ performs case distinction with a linear code size if strong domain alignment is not required.

D. Updatable Functions: Restricted Reusability

In field calculus [7] the following prototype of "updatable function" is suggested:

```
def up(injecter) {
  snd( rep(injecter()) {
    (x) => {max-hood(injecter(),nbr{x})}
  } )
}
```

where `injecter` is a function returning a pair ⟨`version number`, `function code`⟩, and the built-in operator `max-hood` selects the pair with the highest version number among a local pair and a field of pairs.

Then `up(injecter)` defines an "updatable function" by spreading functions with high version number throughout devices. However, in a strict domain aligned environment it can not be used to define an updatable function returning fields (such as a metric). In this case, whenever a new function is first injected it will be computed into the restricted environment consisting of only current device, thus returning a field that will not be aligned with respect to the calling context.

Even though `up` can still work under the assumptions that new versions are injected at a slow rate (so that most of the time no alignment is required), and an occasionally empty field domain does not produce critical effects, in the next section we shall seek for a better solution.

V. Case Study: Updatable Metric

We now present a new building block: a refined *updatable function* (see Figure 3) which can be applied to functions returning field values, while guaranteeing the fields returned to have full domain. This building block can be fruitfully applied also to functions `f` not returning fields, providing stability advantages whenever the execution of such `f` involves communication between neighbours.

To enhance code readability, we expand field calculus with `let` instructions and records (so that the elements of injected pairs can be accessed via `.ver`, `.fun`), which are nonetheless supported by Protelis [13] and can be simulated by function abstractions. The following variables are used:

- `procs`: list of functions ever executed on the network (as records with version number).
- `max_p`: the highest version number in `procs`.
- `cur_p`: the version number of the function used in the previous round.
- `n_num`: the number of neighbours which called `safeup`.
- `injecter`: a built-in sensor returning a record with function and version number.
- `d_cur`: the outcome of the currently considered version of the metric.
- `nxt`: the outcome field and version selected among the versions with higher version number.

In short, function `safeup` repeatedly updates the list `procs` of all versions ever executed, by retrieving lists from neighbours and possibly updating it with a new local version obtained through `injecter`. Then it calls function `exec` which in turn executes every function in the list which is not older than any version currently used by a neighbour. Finally, `exec` returns the outcome of the function with the highest version number that is shared by all neighbours.

Since `safeup` only returns fields that have been calculated by all neighbours, its output always satisfies strong domain alignment. However, its intermediate results necessarily violate it: whenever a new version is injected, it starts to be executed on a device with no aligned neighbours and is returned by nested `exec` calls (violating strong domain alignment) until an older shared version is met. Thus `safeup` cannot be formulated in a strict domain aligned environment, even though it is provable that the not-aligned part of its computation is always discarded.

We examine the behaviour of *updatable metric* in a case study. Consider an environment in which shortest paths to a `source` region are repeatedly computed through algorithm CRF-Gradient [15] (a refinement of *parametric gradient* allowing faster recovery from network changes) with parameter $v_0 = \infty$. The metric used by CRF-Gradient is implemented by the *updatable* paradigm, so that improved versions of the metric[5] can be injected dynamically without disrupting the system. We inject the following versions: ($t = 0$) `nbr-range` with a random 10% error in every device; ($t = 30$) `nbr-range` without error in a random device; ($t = 60$) density-weighted metric (discouraging crowded areas) in another random device.

We compare the behaviour of `up` and `safeup` for a network of 1000 nodes, placed randomly with uniform distribution on a unit disc in order to produce a network 32 hops wide. Rounds are carried out asynchronously, where subsequent firings of a same device are separated by a random 0.9 to 1.1 time units.

As shown in Figure 4, using algorithm `up` the values grow to infinity (cropped to the top) after any injection of a new function; while with algorithm `safeup` they change smoothly. In particular, when the metric injected is a refinement of the previous version ($t = 30$) the error of `safeup` does not rise. The error peak in $t = 60$ is due to the sudden change of target values, while computed values adapt gradually (as shown in the left picture). Similar graphs can be obtained with simple *parametric gradient* in place of CRF-Gradient, thus are not shown here.

VI. Conclusion

In this paper we compared the *Strict* and *Permissive* approach to run-time management of field domains in field calculus. In both cases, the basic safety guarantees of *weak*

[5] An updatable metric can correspond in practice to an updatable range-detection driver, which is able to accomodate for bugfixes, general improvements and addition of new functionalities.

```
def exec(procs, max_p, cur_p, n_num) {
 let d_cur = if (min-hood(nbr{cur_p}) <= head(procs).ver) {head(procs).fun()} {nbr{0}};
 let nxt = if (head(procs).ver < max_p) {exec(tail(procs), max_p, cur_p, n_num)} {pair(nbr{0},-1)};
 mux(snd(nxt) < 0 and n_num = sum-hood(nbr{1}), pair(d_cur, head(procs).ver), nxt)
}

def safeup(injecter) {
 4th(rep (nil, -1, -1, nbr{0}) {
  (procs, max_p, cur_p, field) => {
   let nmax_p, nprocs = max-hood(nbr{max_p, procs});
   let nnmax_p, nnprocs = if (injecter().ver > nmax_p) {injecter().ver, append(nprocs, injecter())}
                                                      {nmax_p, nprocs};
   let x = exec(nnprocs, nnmax_p, cur_p, sum-hood(nbr{1}));
   (nnprocs, nnmaxp, snd(x), fst(x))
  }
 })
}
```

Fig. 3. A refined implementation of updatable function.

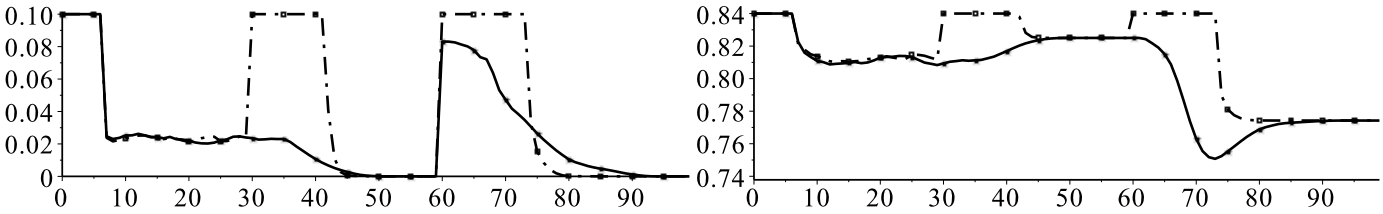

Fig. 4. Square mean relative error (left) and mean value (right) for `safeup` (solid line) and `up` (dashed line).

domain alignment and *restriction property* are granted. In the strict case, *strong domain alignment* holds which is able to prevent subtle errors occurring when composing different expressions. In the permissive case, some additional relevant functions are allowed, as the *updatable metric*.

Such expressions are needed often enough to leverage the full power of field computations—however, a wide majority of functions would benefit from strong domain alignment.

We then propose the following guidelines for development of practical implementations of field calculus: *(i)* exceptions on domain mismatch should be thrown by default; however *(ii)* it must be possible for a programmer to explicitly allow functions to handle domain mismatch in a *Permissive* fashion. These guidelines translate in practice into providing two overloads of built-in operators as pointwise operations on field values: a *throwing* form and a (discouraged) *non-throwing* form.

The results of this paper integrate smoothly with the workflow proposed in [10], where permissive functions are meant to be used mostly as building blocks (and seldom for application-specific internal features). The proposed examples (and in particular the *updatable metric*) expand on the work on safe code deployment in [7], and the properties studied (*weak* and *strong* domain alignment, restriction property) refine the corresponding notions introduced in [7], [14] and integrates the results on safety properties for aggregate computing (self-stabilisation [10], device distribution independence [11]).

References

[1] A. Church, "A set of postulates for the foundation of logic," *Annals of Mathematics*, vol. 33, no. 2, pp. 346–366, 1932.

[2] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes, part I," *Information and Computation*, vol. 100, no. 1, pp. 1–40, September 1992.

[3] N. Busi, R. Gorrieri, and G. Zavattaro, "On the expressiveness of linda coordination primitives," *Inf. Comput.*, vol. 156, no. 1-2, pp. 90–121, 2000.

[4] L. Bettini, V. Bono, R. D. Nicola, G. L. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri, "The Klaim project: Theory and practice," in *Global Computing 2003*, ser. Lecture Notes in Computer Science, vol. 2874. Springer, 2003, pp. 88–150.

[5] 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*, vol. 23, no. 3, 2001.

[6] F. Damiani, M. Viroli, and J. Beal, "A type-sound calculus of computational fields," *Science of Computer Programming*, vol. 117, pp. 17 – 44, 2016.

[7] F. Damiani, M. Viroli, D. Pianini, and J. Beal, "Code mobility meets self-organisation: A higher-order calculus of computational fields," in *Formal Techniques for Distributed Objects, Components, and Systems*, ser. LNCS. Springer, 2015, vol. 9039, pp. 113–128.

[8] M. Mamei and F. Zambonelli, "Programming pervasive and mobile computing applications: The tota approach," *ACM Trans. on Software Engineering Methodologies*, vol. 18, no. 4, pp. 1–56, 2009.

[9] J. Beal, D. Pianini, and M. Viroli, "Aggregate programming for the internet of things," *IEEE Computer*, vol. 48, no. 9, 2015.

[10] M. Viroli, J. Beal, F. Damiani, and D. Pianini, "Efficient engineering of complex self-organising systems by self-stabilising fields," in *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE, 2015, pp. 81–90.

[11] J. Beal, M. Viroli, D. Pianini, and F. Damiani, "Self-adaptation to device distribution changes in situated computing systems," in *IEEE Conference on Self-Adaptive and Self-Organising Systems (SASO 2016)*. IEEE, 2016, to appear.

[12] J. Beal, M. Viroli, and F. Damiani, "Towards a unified model of spatial computing," in *7th Spatial Computing Workshop (SCW 2014)*, AAMAS 2014, Paris, France, May 2014.

[13] D. Pianini, M. Viroli, and J. Beal, "Protelis: Practical aggregate programming," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15. ACM, 2015, pp. 1846–1853.

[14] M. Viroli, F. Damiani, and J. Beal, "A calculus of computational fields," in *Advances in Service-Oriented and Cloud Computing*, ser. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2013, vol. 393, pp. 114–128.

[15] J. Beal, J. Bachrach, D. Vickery, and M. Tobenkin, "Fast self-healing gradients," in *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 2008, pp. 1969–1975.