Logical Methods in Computer Science
Vol. 13(1:13)2017, pp. 1–46
www.lmcs-online.org

Submitted Feb. 5, 2016
Published Mar. 22, 2017

ASYNCHRONOUS DISTRIBUTED EXECUTION OF FIXPOINT-BASED COMPUTATIONAL FIELDS

ALBERTO LLUCH LAFUENTE [a], MICHELE LORETI [b], AND UGO MONTANARI [c]

[a] DTU Compute, Technical University of Denmark, Denmark
e-mail address: albl@dtu.dk

[b] University of Florence, Italy
e-mail address: michele.loreti@unifi.it

[c] Computer Science Department, University of Pisa, Italy
e-mail address: ugo@di.unipi.it

Abstract. Coordination is essential for dynamic distributed systems whose components exhibit interactive and autonomous behaviors. Spatially distributed, locally interacting, propagating computational fields are particularly appealing for allowing components to join and leave with little or no overhead. Computational fields are a key ingredient of *aggregate programming*, a promising software engineering methodology particularly relevant for the Internet of Things. In our approach, space topology is represented by a fixed graph-shaped field, namely a network with attributes on both nodes and arcs, where arcs represent interaction capabilities between nodes. We propose a SMuC calculus where μ-calculus-like modal formulas represent how the values stored in neighbor nodes should be combined to update the present node. Fixpoint operations can be understood globally as recursive definitions, or locally as asynchronous converging propagation processes. We present a distributed implementation of our calculus. The translation is first done mapping SMuC programs into normal form, purely iterative programs and then into distributed programs. Some key results are presented that show convergence of fixpoint computations under fair asynchrony and under reinitialization of nodes. The first result allows nodes to proceed at different speeds, while the second one provides robustness against certain kinds of failure. We illustrate our approach with a case study based on a disaster recovery scenario, implemented in a prototype simulator that we use to evaluate the performance of a recovery strategy.

1. Introduction

Coordination is essential in all the activities where an ensemble of agents interacts within a distributed system. Particularly interesting is the situation where the ensemble is dynamic, with agents entering and exiting, and when the ensemble must adapt to new situations and

Key words and phrases: distributed computing, computational fields, distributed graph algorithms, coordination, modal logics.

Research supported by the European projects IP 257414 ASCENS and STReP 600708 QUANTICOL, and the Italian PRIN 2010LHT4KM CINA.

DOI:10.23638/LMCS-13(1:13)2017
© A. Lluch Lafuente, M. Loreti, and Ugo Montanari
Creative Commons

must have in general an autonomic behavior. Several models of coordination have been proposed and developed in the past years. Following the classification of [21], we mention: (i) point-to-point direct coordination, (ii) connector-based coordination, (iii) shared data spaces, (iv) shared deductive knowledge bases, and (v) spatially distributed, locally interacting, propagating computational fields.

Computational Fields. Among them, computational fields are particularly appealing for their ability of allowing new interactions with little or no need of communication protocols for initialization. Computational fields are analogous to fields in physics: classical fields are scalars, vectors or tensors, which are functions defined by partial differential equations with initial and/or boundary conditions. Analogously, computational fields consist of suitable space dependent data structures, where interaction is possible only between neighbors.

Computational fields have been proposed as models for several coordination applications, like amorphous computing, routing in mobile ad hoc and sensor networks, situated multi agent ecologies, like swarms, and finally for robotics applications, like coordination of teams of modular robots. Physical fields, though, have the advantage of a regular structure of space, e.g. the one defined by Euclidean geometry, while computational fields are sometimes based on specific (logical) networks of connections. The topology of such a network may have little to do with Euclidean distance, in the sense that a node can be directly connected with nodes which are far away, e.g. for achieving a logarithmic number of hops in distributed hash tables. However, for several robotics applications, and also for swarms and ad hoc networking, one can reasonably assume that an agent can directly interact only with peers located within a limited radius. Thus locality of interaction and propagation of effects become reasonable assumptions.

The computational fields approach has a main conceptual advantage: it offers to the analyst/programmer a high level interface for the collection of possibly inhomogeneous components and connectors which constitute the distributed system under consideration. This view is not concerned with local communication and computation, but only with the so called *emergent behavior* of the system. Coordination mechanisms should thus be resilient and self-stabilizing, they should adjust to network structure and should scale to large networks. As for physical fields, this approach distinguishes clearly between local parameters, which express (relatively) static initial/boundary/inhomogeneity conditions, and field values, which are computed in a systematic way as a result of the interaction with neighbour components. Analogously to the very successful *map-reduce* strategy [12], typical operations are propagation and accumulation of values, with selection primitives based on distance and gradient. However, actual computation may require specific control primitives, guaranteeing proper sequentialization of possibly different fields. In particular, if coordination primitives are executed in asynchronous form, suitable termination/commit events should be available in the global view.

Aggregate Programming. Recently, computational fields have been integrated in a software engineering methodology called *aggregate programming* [4]. Several abstraction layers are conceptualized, from component/connector capabilities, to coordination primitives, to API operations for global programming. As a main application, the *Internet of Things* (IoT) has been suggested, a context where the ability to offer a global view is particularly appreciated.

Another programming style where the advantages of aggregate programming could be felt significantly is *fog* or *edge computing* [34]. In this approach, a conceptual level below (or at the edge) of the cloud level is envisaged, where substantial computing and storage resources are available in a located form. Organizing such resources in an aggregated, possibly hierarchical, style might combine the positive qualities of pattern-based distributed programming with the abstract view of cloud virtualization.

A third potential application domain for aggregate programming is Big Data analytics, where many analyses are essentially based on the computation of fixpoints over a graph or network. Typical examples are centrality measures like PageRank or reachability properties like shortest paths. Entire parallel graph analysis frameworks like Google's Pregel [19] and Apache's Giraph [8] are built precisely around this idea, originally stemming from the bulk synchronous parallel model of computation [30]. Those frameworks include features for propagating and aggregating values from and among neighbour nodes, as well as termination detection mechanisms.

Contributions. This paper introduces SMuC, the *Soft Mu-calculus for Computational fields*, and presents some fundamental results. In particular the main contributions of the paper are (i) a detailed presentation of the SMuC calculus, (ii) results on robustness against node unavailability, (iii) results on robustness against node failures, (iv) a distributed implementation, (v) a case study.

(i) SMuC. In SMuC execution corresponds to sequential computation of fixpoints in a computational field that represents a fixed graph-shaped topology. Fields are essentially networks with attributes on both nodes and arcs, where arcs represent interaction capabilities between nodes. We originally introduced SMuC in [16] as based on the *semiring μ-calculus* [17], a constraint semiring-valued generalisation of the modal μ-calculus, which provides a flexible mechanism to specify the neighbor range (according to path formulae) and the way attributes should be combined (through semiring operators). Constraint semirings are semirings where the additive operation is idempotent and the multiplicative operation is commutative. The former allows one to define a partial ordering as $a \sqsubseteq b$ iff $a + b = a$, under which both additive and multiplicative operations are monotone. The diamond modality corresponds, in the ordinary μ-calculus, to disjunction of the logical values on the nodes reached by all the outgoing arcs. In soft μ-calculus the values are semiring values and the diamond modality corresponds to the additive operation of the semiring. Similarly for the box modality and the multiplicative operation of the semiring. In the present version of SMuC there is no distinction between the two modalities: we have only a parametric modality labeled by monotone associative and commutative operations. More precisely, we have a forward and a backward modality, referring to outgoing and ingoing arcs. This generalisation allows us to cover more cases of domains and operations.

We believe that our approach based on μ-calculus-like modalities can be particularly convenient for aggregate programming scenarios. In fact, the μ-calculus, both in its original and in its soft version, offers a high level, global meaning expressed by its recursive formulas, while their interpretation in the evaluation semantics computes the fixpoints via iterative approximations which can be interpreted as propagation processes. Thus the SMuC calculus provides a well-defined, general, expressive link bridging the gap between the component/connector view and the emergent behavior view.

(ii) Robustness against node unavailability. Under reasonable conditions, fixpoints can be computed by asynchronous iterations, where at each iteration certain node attributes are updated based on the attributes of the neighbors in the previous iteration. Not necessarily all nodes must be updated at every iteration: to guarantee convergence it is enough that every node is updated infinitely often. Furthermore, the fixpoint does not depend on the particular sequence of updates. If the partial ordering has only finite chains, the unique (minimal or maximal) fixpoint is reached in a finite number of iterations. In order to guarantee convergence, basic constructs must be monotone. Theorem 3.17 formalises this key result.

(iii) Robustness against node failures. Another concern is about dependability in the presence of failure. In our model, only a limited kind of failure is taken care of: nodes and links may fail, but then they regularly become active, and the underlying mechanism guarantees that they start from some previous back up state or are restarted. Robustness against such failures is precisely provided by Theorem 3.19, which guarantees that if at any step some nodes are updated with the values they had in previous steps, possibly the initialization value, but then from some time on they work correctly, the limit value is still the fixpoint. In fact, from a semantical point of view the equivalence class of possible computations for a given formula is characterized as having the same set of upper bounds (and thus the same least fixpoint).

A more general concern is about possible changes in the structure of the network. The meaning of a μ-calculus formula is supposed to be independent of the network itself: for instance a formula expressing node assignment as the minimal distance of every node from some set of final nodes is meaningful even when the network is modified: if the previous network is not changed, and an additional part, just initialised, is connected to it, the fixpoint computation can proceed without problems: it just corresponds to the situation where the additional part is added at the very beginning, but its nodes have never been activated according to the chosen asynchronous computation policy. In general, however, specific recovery actions must be foreseen for maintaining networks with failures, which apply to our approach just as they concern similar coordination styles. Some remedies to this can be found for example in [24] where overlapping fields are used to adapt to network changes.

(iv) Distributed implementation. We present a possible distributed implementation of our calculus. The translation is done in two phases, from SMuC programs into normal form SMuC programs (a step which explicits communication and synchronisation constraints) and then into distributed programs. The correctness of translations exploits the above mentioned results on asynchronous computations.

A delicate issue in the distributed implementation is how to detect termination of a fixpoint computation. Several approaches are possible. We considered the Dijkstra-Scholten algorithm [13], based on the choice of a fixed spanning tree. We do not discuss how to construct and deploy such a tree, or how to maintain it in the presence of certain classes of failures and of attackers. However, most critical aspects are common to all the models based on computational fields. On a related issue, spanning trees can be computed in SMuC as illustrated in one of the several examples we provide in this paper.

(v) Case study. As a meaningful case study, we present a novel disaster recovery coordination strategy. The goal of the coordination strategy is to direct several rescuers present in the network to help a number of victims, where each victim may need more than one rescuer.

While an optimal solution is not required, each victim should be reached by its closest rescuers, so to minimise intervention time. Our proposed approach may need several iterations of a sequence of three propagations: the first to determine the distance of each rescuer from his/her closest victim, the second to associate to every victim v the list of rescuers having v as their closest victim, so to select the best k of them, if k helpers are needed for v; finally, the third propagation is required for notifying each selected rescuer to reach its specific victim.

We have also developed a prototype tool for our language, equipped with a graphical interface that provides useful visual feedback. Indeed we employ those visual features to illustrate the application of our approach to the aforementioned case study.

Previous work. A first, initial version of SMuC was presented in [16]. However the present version offers important improvements in many aspects.

- Graph-based fields and SMuC formulas are generalised here to ω-chain-complete partial orders, with constraint semirings (and their underlying partial orders) being a particularly interesting instance. The main motivation behind such extension is that some of the values transmitted and updated, as data and possibly SMuC programs themselves, can be given a partial ordering structure relatively easily, while semirings require lots of additional structure, which sometimes is not available and not fully needed.
- We have formalised the notion of asynchronous computation of fixpoints in our fields and have provided results ensuring that, under reasonable conditions, nodes can proceed at different speeds without synchronising at each iteration, while still computing the same, desired fixpoint.
- We have formalised a notion of safe computation, that can handle certain kinds of failures and have shown that fixpoint computations are robust against such failures.
- The simple imperative language on which SMuC is embedded has been simplified. In particular it is now closer to standard while [23]. The motivations, besides simplicity and adherence to a well-known language, is that it becomes easier to define control flow based on particular agreements and not just any agreement (as it was in [16]). Of course, control flow based on any agreement can still be achieved, as explained in the paper.
- The distributed realisation of SMuC programs has been fully re-defined, refined and improved. Formal proofs of correctness have been added. Moreover, the global agreement mechanism is now related to the *Dijkstra-Scholten* algorithm [13] for termination detection.

Structure of the paper. The rest of the paper is structured as follows. Sect. 2 recalls some basic definitions related to partial orders and semirings. Sect. 3 presents the SMuC calculus and the results related to robustness against unavailability and failures. Sect. 4 presents the SMuC specification of our disaster recovery case study, which is illustrated with figures obtained with our prototypical tool. Sect. 5 discusses several performance and synchronisation issues related to distributed implementations of the calculus. Sect. 6 discusses related works. Sect. 7 concludes the paper, describes our current work and identifies opportunities for future research. Formal proofs can be found in the appendix, together with a table of symbols.

2. Background

Our computational fields are essentially networks of inter-connected agents, where both agents and their connections have attributes. One key point in our proposal is that the domains of attributes are partially ordered and possibly satisfy other properties. Attributes, indeed, can be natural numbers (e.g. ordered by $\leq$ or $\geq$), sets of nodes (e.g. ordered by containement), paths in the graph (e.g. lexicographically ordered), etc. We call here such domains *field domains* since node-distributed attributes form, in a certain sense, a computational field of values. The basic formal underlying structure we will consider for attributes is that of complete partial orders (CPOs) with top and bottom elements, but throughout the paper we will see that requiring some additional conditions is fundamental for some results.

Definition 2.1 (field domain)**.** Our *field domains* are tuples $\langle A, \sqsubseteq, \bot, \top \rangle$ such that $\langle A, \sqsubseteq \rangle$ is an ω-chain complete partially $\sqsubseteq$-ordered set A with bottom element $\bot \in A$ and top element $\top \in A$, and $\langle A, \sqsupseteq \rangle$ is an ω-chain complete partially $\sqsupseteq$-ordered set A with bottom element $\top \in A$ and top element $\bot \in A$.

Recall that an ω-chain in a complete partially $\sqsubseteq$-ordered (resp. $\sqsupseteq$-ordered) set A is an infinite sequence $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq \dots$ (resp. $a_0 \sqsupseteq a_1 \sqsupseteq a_2 \sqsupseteq \dots$) and that in such domains all ω-chains have a least upper (resp. greatest lower) bound.

With an abuse of notation we sometimes refer to a field domain $\langle A, \sqsubseteq, \bot, \top \rangle$ with the carrier A and to its components by subscripting them with the carrier, i.e. $\sqsubseteq_A$, $\bot_A$ and $\top_A$. For the sake of a lighter notation we drop the subscripts if clear from the context.

Example 2.2. Some typical examples of field domains are:

- Boolean and quasi-boolean partially ordered domains such as the classical Boolean domain $\langle \{\mathit{true}, \mathit{false}\}, \rightarrow, \mathit{false}, \mathit{true} \rangle$, Belnap's 4-valued domains, etc.
- Totally ordered numerical domains such as $\langle A, \leq, 0, +\infty \rangle$, with A being $\mathbb{N} \cup \{+\infty\}$, $\mathbb{R}^+ \cup \{+\infty\}$, or $\langle [a..b], \leq, a, b \rangle$ with $a, b \in \mathbb{R}$ and $a \leq b$, etc.;
- Sets with containment relations such as $\langle 2^A, \subseteq, \emptyset, A \rangle$;
- Words with lexicographical relations such as $\langle A^* \cup \{\bullet\}, \sqsubseteq, \epsilon, \bullet \rangle$, with A being a partially ordered alphabet of symbols, A^* denoting possibly empty sequences of symbols of A and $\sqsubseteq$ being a lexicographical order with the empty word ϵ as bottom and $\bullet$ as top element (an auxiliary element that dominates all words).

Many other domains can be constructed by *reversing* the domains of the above example. For example, $\langle \{\mathit{true}, \mathit{false}\}, \leftarrow, \mathit{true}, \mathit{false} \rangle$, $\langle A, \geq, +\infty, 0 \rangle$, $\langle 2^A, \supseteq, A, \emptyset \rangle$, $\langle (A^* \cup \{\bullet\}, \sqsupseteq, \bullet, \epsilon \rangle$... can be considered as domains as well. Moreover, additional domains can be constructed by composition of domains, e.g. based on Cartesian products and power domains. The Cartesian product is indeed useful whenever one needs to combine two different domains.

Definition 2.3 (Cartesian product)**.** Let $\langle A_1, \sqsubseteq_1, \bot_1, \top_1 \rangle$ and $\langle A_2, \sqsubseteq_2, \bot_2, \top_2 \rangle$ be two field domains. Their *Cartesian product* $\langle A_1, \sqsubseteq_1, \bot_1, \top_1 \rangle \times \langle A_2, \sqsubseteq_2, \bot_2, \top_2 \rangle$ is the field domain $\langle A_1 \times A_2, \sqsubseteq, (\bot_1, \bot_2), (\top_1, \top_2) \rangle$ where $\sqsubseteq$ is defined as $(a_1, a_2) \sqsubseteq (a_1', a_2')$ iff $a_1 \sqsubseteq_1 a_1'$ and $a_2 \sqsubseteq_1 a_2'$.

In some of the examples we shall use a variant of the Cartesian product where pairs of values are ordered lexicographically, corresponding to the case in which there is a priority between the two dimensions being combined.

Definition 2.4 (Lexicographical Cartesian product)**.** Let $\langle A_1, \sqsubseteq_1, \bot_1, \top_1 \rangle$ and $\langle A_2, \sqsubseteq_2, \bot_2, \top_2 \rangle$ be two field domains. Their *lexicographical cartesian product* $\langle A_1, \sqsubseteq_1, \bot_1, \top_1 \rangle \times_1 \langle A_2, \sqsubseteq_2, \bot_2, \top_2 \rangle$ is the field domain $\langle A_1 \times A_2, \sqsubseteq, (\bot_1, \bot_2), (\top_1, \top_2) \rangle$ where $\sqsubseteq$ is defined as $(a_1, a_2) \sqsubseteq (a_1', a_2')$ iff $a_1 \sqsubset_1 a_1'$ or ($a_1 = a_1'$ and $a_2 \sqsubseteq_1 a_2'$).

Sometimes one needs to deal with sets of non-dominated values, for example when considering multi-criteria optimisation problems. A suitable construction in this case is to consider the Hoare Power Domain [29].

Definition 2.5 (Hoare Power Domain)**.** Let $\langle A, \sqsubseteq, \bot, \top \rangle$ be a field domain. The *Hoare Power Domain* $P^H(\langle A, \sqsubseteq, \bot, \top \rangle)$ is the field domain $\langle \{B \subseteq A \mid a \in B \wedge b \sqsubseteq a \Rightarrow b \in B\}, \subseteq, \emptyset, A \rangle$.

In words, the obtained domain contains downward closed sets of values, ordered by set inclusion.

Our agents will use arbitrary functions to operate on attributes and to coordinate. Among other things, agents will compute (least or greatest) fixpoints of functions on field domains. Of course, for fixpoints to be well-defined some restrictions may need to be imposed, in particular regarding monotonicity and continuity properties.

A sufficient condition for the least and greatest fixpoints of a function $f : A \to A$ on an ω-chain complete field domain A to be well-defined is for f to be continuous and monotone. Recall that our domains are such that all infinite chains of partially ordered (respectively reverse-ordered) elements have a least upper bound (respectively a greatest lower bound). Indeed, under such conditions the least upper bound of the chain $\bot \sqsubseteq f \bot \sqsubseteq f^2 \bot \sqsubseteq \dots$ is the least fixpoint of f. Similarly, the greatest lower bound of chain $\top \sqsupseteq f \top \sqsupseteq f^2 \top \sqsupseteq \dots$ is the greatest fixpoint of f.

Another desirable property is for fixpoints to be computable by iteration. This means that the least and greatest fixpoints of f are equal to $f^n \top$ and $f^m \bot$, respectively, for some $n, m \in \mathbb{N}$. In some cases, we will indeed require that all chains of partially ordered elements are finite. In that case we say that the chains *stabilize*, which refers to the fact that, for example, $f^n \bot = f^{n+k} \bot$, for all $k \in \mathbb{N}$. This guarantees that the computation of a fixpoint by successive approximations eventually terminates since every iteration corresponds to an element in the chain. If this is not the case the fixpoint can only be approximated or solved with some alternative method that may depend on the concrete field domain and the class of functions under consideration.

To guarantee some of those properties, we will often instantiate our approach on algebraic structures based on a class of semirings called *constraint semirings* (just *semirings* in the following). Such class of semirings has been shown to be very flexible, expressive and convenient for a wide range of problems, in particular for optimisation and solving in problems with soft constraints and multiple criteria [7].

Definition 2.6 (semiring)**.** A *semiring* is a tuple $\langle A, +, \times, \bot, \top \rangle$ composed by a set A, two operators $+$, $\times$ and two constants $\bot$, $\top$ such that:

- $+ : 2^A \to A$ is an associative, commutative, idempotent operator to "choose" among values;
- $\times : A \times A \to A$ is an associative, commutative operator to "combine" values;
- $\times$ distributes over $+$;
- $\bot + a = a$, $\top + a = \top$, $\top \times a = a$, $\bot \times a = \bot$ for all $a \in A$;

- $\sqsubseteq$, which is defined as $a \sqsubseteq b$ iff $a+b=b$, provides a field domain of preferences $\langle A, \sqsubseteq, \bot, \top \rangle$ (which is actually a complete lattice [7]).

Recall that *classical* semirings are algebraic structures that are more general than the (constraint) semirings we consider here. In fact, classical semirings do not require the additive operation $+$ to be idempotent or the multiplicative operation $\times$ to be commutative. Such axiomatic properties, however, turn out to yield many useful and interesting features (e.g. in constraint solving [7] and model checking [17]) and are actually provided by many semirings, such as the ones in Example 2.7.

Again, we shall use the notational convention for semirings that we mentioned for field domains, i.e. we sometimes denote a semiring by its carrier A and the rest of the components by subscripting them with A. Note also that since the underlying field domain of a semiring is a complete lattice, all partially ordered chains have least upper and greatest lower bounds.

Example 2.7. Typical examples of semirings are:

- the Boolean semiring $\langle \{true, false\}, \vee, \wedge, false, true \rangle$;
- the tropical semiring $\langle \mathbb{R}^+ \cup \{+\infty\}, min, +, +\infty, 0 \rangle$;
- the possibilistic semiring: $\langle [0..1], max, \cdot, 0, 1 \rangle$;
- the fuzzy semiring $\langle [0..1], max, min, 0, 1 \rangle$;
- and the set semiring $\langle 2^A, \cup, \cap, \emptyset, A \rangle$.

All these examples have an underlying domain that can be found among the examples of field domains in Example 2.2. As for domains, additional semirings can be obtained in some cases by reversing the underlying order. For instance, $\langle \{true, false\}, \wedge, \vee, true, false \rangle$, $\langle [0..1], min, max, 1, 0 \rangle$, ... are semirings as well. A useful property of semirings is that Cartesian products and power constructions yield semirings, which allows one for instance to lift techniques for single criteria to multiple criteria.

3. SMuC: A Soft μ-calculus for Computations fields

3.1. **Graph-based Fields.** We are now ready to provide our notion of field, which is essentially a fixed graph equipped with field-domain-valued node and edge labels. The idea is that nodes play the role of agents, and (directed) edges play the role of (directional) connections. Labels in the graph are of two different natures. Node labels are used as the names of attributes of the agents. On the other hand, edge labels correspond to functions associated to the connections, e.g. representing how attribute values are transformed when traversing a connection.

Definition 3.1 (field)**.** A *field* is a tuple $\langle N, E, A, L = L_N \uplus L_E, I = I_N \uplus I_E \rangle$ formed by

- a set N of nodes;
- a relation $E \subseteq N \times N$ of edges;
- a set L of node labels L_N and edge labels L_E;
- a field domain $\langle A, \sqsubseteq, \bot, \top \rangle$;
- an interpretation function $I_N : L_N \to N \to A$ associating a function from nodes to values to every node label in L_N;
- an interpretation function $I_E : L_E \to E \to A \to A$ associating a function from edges to functions from values to values to every edge label in P;

where node, edge, and label sets are drawn from a corresponding universe, i.e. $N \subseteq \mathcal{N}$, $E \subseteq \mathcal{E}$, $L_N \subseteq \mathcal{L}$, $L_E \subseteq \mathcal{L}'$.

As usual, we may refer to the components of a field F using subscripted symbols (i.e. N_F, E_F, ...). We will denote the set of all fields by $\mathcal{F}$.

It is worth to remark that while standard notions of computational fields tend to be restricted to nodes (labels) and their mapping to values, our notion of field includes the topology of the network and the mapping of edge (labels) to functions. As a matter of fact, the topology plays a fundamental role in our field computations as it defines how agents are connected and how their attributes are combined when communicated. Note that we consider directed edges since there are many cases in which the direction of the connection matters as we shall see in applications based on spanning trees or shortest paths. On the other hand, the role of node and edge labels is different in our approach. In fact, some node labels are computed as the result of a fixpoint approximation which corresponds to a propagation procedure. They thus represent the genuine computational fields. Edge labels, instead, are assigned directly in terms of the data of the problem (e.g. distances) or in terms of the results of previous propagations. They thus represent more properly equation coefficients and boundary conditions as one can have in *partial differential equations* in physical fields.

3.2. **SMuC Formulas.** SMuC (*Soft μ-calculus for Computations fields*) is meant to specify global computations on fields. One key aspect of our calculus are atomic computations denoted with expressions reminiscent of the semiring modal μ-calculus proposed in [17]. The semiring μ-calculus departed from the modal μ-calculus, a very flexible and expressive calculus that subsumes other modal temporal logics such as CTL* (and hence also CTL and LTL). The semiring μ-calculus inherits essentially the same syntax as the modal μ-calculus (i.e. predicate logic enriched with temporal operators and fixpoint operators) but changes the domain of interpretation from Booleans (i.e. set of states that satisfy a formula) to semiring valuations (i.e. mappings of states to semiring values), and the semantic interpretation of operators, namely disjunction and existential quantification are interpreted as the semiring addition, while conjunction and universal quantification are interpreted as semiring multiplication. In that manner, the semiring μ-calculus captures the ordinary μ-calculus for the Boolean semiring but, in addition, allows one to reason about quantitative properties of graph-based structures like transition systems (i.e. quantitative model checking) and network topologies (e.g. shortest paths and similar properties).

In SMuC similar expressions will be used to specify the functions being calculated by global computations, to be recorded by updating the interpretation functions of the nodes.

Given a field domain A, we shall call functions $f, g, \dots$ on A *attribute* operations. Functions $f : A^* \to A$ will be used to combine values, while functions $g : mset(A) \to A$ will be used to aggregate values, where $mset(A)$ denotes the domain of finite multisets on A. The latter hence have finite multisets of A-elements as domain. The idea is that they are going to be used to aggregate values from neighbour nodes using associative and commutative functions, so that the order of the arguments does not matter. A function $N \to A$ is called a *node valuation* that we typically range over by $\mathsf{f}, \mathsf{g}, \dots$. Note that we use the same symbols but a different font since we sometimes lift an attribute operation to a set of nodes. For instance a zero-adic attribute operation $f : \to A$ can be lifted to a node valuation $\mathsf{f} : N \to A$ in the obvious way, i.e. $\mathsf{f} = \lambda n.f$. A function $\psi : (N \to A) \to N \to A$

is called an *update function* and is typically ranged over by $\psi, \psi_1, \psi_2, \ldots$. As we shall see, the computation of fixpoints of such update functions is at the core of our approach. Such fixpoints do not refer to functions on the field domain of attribute values $\langle A, \sqsubseteq_A, \bot_A, \top_A \rangle$ but to the field domain of node valuations $\langle (N \to A), \sqsubseteq_{N\to A}, \bot_{N\to A}, \top_{N\to A} \rangle$. That field domain is obtained by lifting $\langle A, \sqsubseteq_A, \bot_A, \top_A \rangle$ to set N, i.e. the carrier of the new field domain is the set of node valuations $N \to A$, the partial ordering relation $\sqsubseteq_{N\to A}$ is such that $\mathsf{f}_1 \sqsubseteq_{N\to A} \mathsf{f}_2$ iff $\forall n. \mathsf{f}_1\, n \sqsubseteq_A \mathsf{f}_2\, n$ and the bottom and top elements $\bot_{N\to A}$, $\top_{N\to A}$ are such that $\bot_{N\to A}\, n = \bot_A$ and $\top_{N\to A}\, n = \top_A$.

Given a set $\mathcal{Z}$ of formula variables, an environment is a partial function $\rho : \mathcal{Z} \to \mathcal{N} \to A$. We shall also use a set $\mathcal{M}$ of function symbols for attribute operations, of functional types $f : A^* \to A$ for combining values or $g : mset(A) \to A$ for aggregating values.

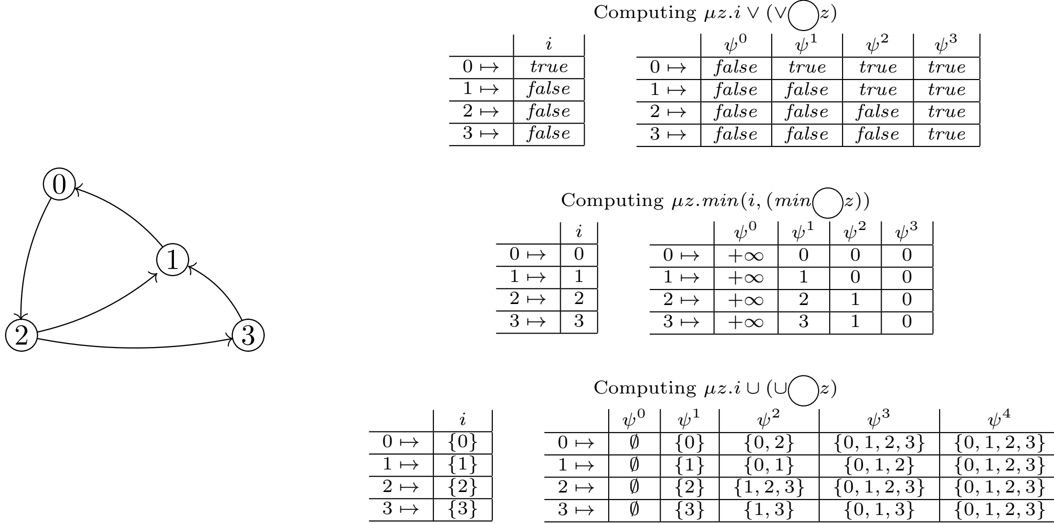
Definition 3.2 (syntax of SMuC formulas)**.** The syntax of SMuC formulas is as follows:

$$\Psi ::= i \mid z \mid f(\Psi, \ldots, \Psi) \mid g \textcircled{\alpha} \Psi \mid g \CIRCLE_{\alpha} \Psi \mid \mu z.\Psi \mid \nu z.\Psi$$

with $i \in \mathcal{L}$, $\alpha \in \mathcal{L}'$, $f, g \in \mathcal{M}$ and $z \in \mathcal{Z}$.

The formulas allow one to combine atomic node labels i, functions f, classical least (μ) and greatest (ν) fixpoint operators and the modal operators $\bigcirc$ and $\CIRCLE$. Including both least and greatest fixpoints is needed since we consider cases in which it is not always possible to express one in terms of the other. It is also useful to consider both since they provide two different ways of describing computations: recursive (in the case of least fixpoints) and co-recursive (in the case of greatest fixpoints). The operational view of formulas can provide a useful intuition of when to use least or greatest fixpoints. Informally, least fixpoints are useful when we conceive the computation being described as starting from none or few information that keeps being accumulated until enough (a fixpoint) is reached. The typical such property in a graph is the reachability of a node satisfying some property. Conversely, the computation corresponding to a greatest fixpoint starts with a lot (possibly irrelevant) amount of information that keeps being refined until no irrelevant information is present. The typical such property in a graph is the presence of an infinite path where all nodes have some feature. Usually infinite paths can be easily represented when they traverse finite cycles in the graph, but in some practical cases the greatest fixpoint approach may be difficult to implement when it requires a form of global information to be available to all nodes. The modal operators are used to aggregate (with function g) values obtained from neighbours following outgoing ($\bigcirc$) or incoming ($\CIRCLE$) edges and using the edge capability α (i.e. the function transforming values associated to label α). We sometimes use id as the identity edge capability and abbreviate $\textcircled{id}$ and $\CIRCLE_{id}$ with $\bigcirc$ and $\CIRCLE$, respectively. The choice of the symbol is reminiscent of modal temporal logics with past operators.

If we choose a semiring as our field domain, the set of function symbols may include, among others, the semiring operator symbols $+$ and $\times$ and possibly some additional ones, for which an interpretation on the semiring of interest can be given. In that case, for instance we can instantiate modal operators $\bigcirc$ and $\CIRCLE$ to "choose" or "combine" values from neighbour nodes as we did in [16], i.e. by using box and diamond operators $[\alpha]\Psi \equiv +\textcircled{\alpha}\Psi$, $\langle\alpha\rangle\Psi \equiv \times\textcircled{\alpha}\Psi$, $[[\alpha]]\Psi \equiv +\CIRCLE_{\alpha}\Psi$ and $\langle\langle\alpha\rangle\rangle\Psi \equiv \times\CIRCLE_{\alpha}\Psi$. Those operators were inspired by classical operators of modal logics: in modal logics the box ($\Box$) and diamond ($\Diamond$) modalities are used to universally or existentially quantify among all possible next worlds, which amounts to aggregate values with logical conjunction and disjunction, respectively.



Computing $\mu z.i \vee (\vee\bigcirc z)$

	i
$0 \mapsto$	*true*
$1 \mapsto$	*false*
$2 \mapsto$	*false*
$3 \mapsto$	*false*

	ψ^0	ψ^1	ψ^2	ψ^3
$0 \mapsto$	*false*	*true*	*true*	*true*
$1 \mapsto$	*false*	*false*	*true*	*true*
$2 \mapsto$	*false*	*false*	*false*	*true*
$3 \mapsto$	*false*	*false*	*false*	*true*

Computing $\mu z.min(i, (min\bigcirc z))$

	i
$0 \mapsto$	0
$1 \mapsto$	1
$2 \mapsto$	2
$3 \mapsto$	3

	ψ^0	ψ^1	ψ^2	ψ^3
$0 \mapsto$	$+\infty$	0	0	0
$1 \mapsto$	$+\infty$	1	0	0
$2 \mapsto$	$+\infty$	2	1	0
$3 \mapsto$	$+\infty$	3	1	0

Computing $\mu z.i \cup (\cup\bigcirc z)$

	i
$0 \mapsto$	$\{0\}$
$1 \mapsto$	$\{1\}$
$2 \mapsto$	$\{2\}$
$3 \mapsto$	$\{3\}$

	ψ^0	ψ^1	ψ^2	ψ^3	ψ^4
$0 \mapsto$	$\emptyset$	$\{0\}$	$\{0,2\}$	$\{0,1,2,3\}$	$\{0,1,2,3\}$
$1 \mapsto$	$\emptyset$	$\{1\}$	$\{0,1\}$	$\{0,1,2\}$	$\{0,1,2,3\}$
$2 \mapsto$	$\emptyset$	$\{2\}$	$\{1,2,3\}$	$\{0,1,2,3\}$	$\{0,1,2,3\}$
$3 \mapsto$	$\emptyset$	$\{3\}$	$\{1,3\}$	$\{0,1,3\}$	$\{0,1,2,3\}$

Figure 1: The underlying graph of a field (left) and the computation of three SMuC formulas in detail (right).

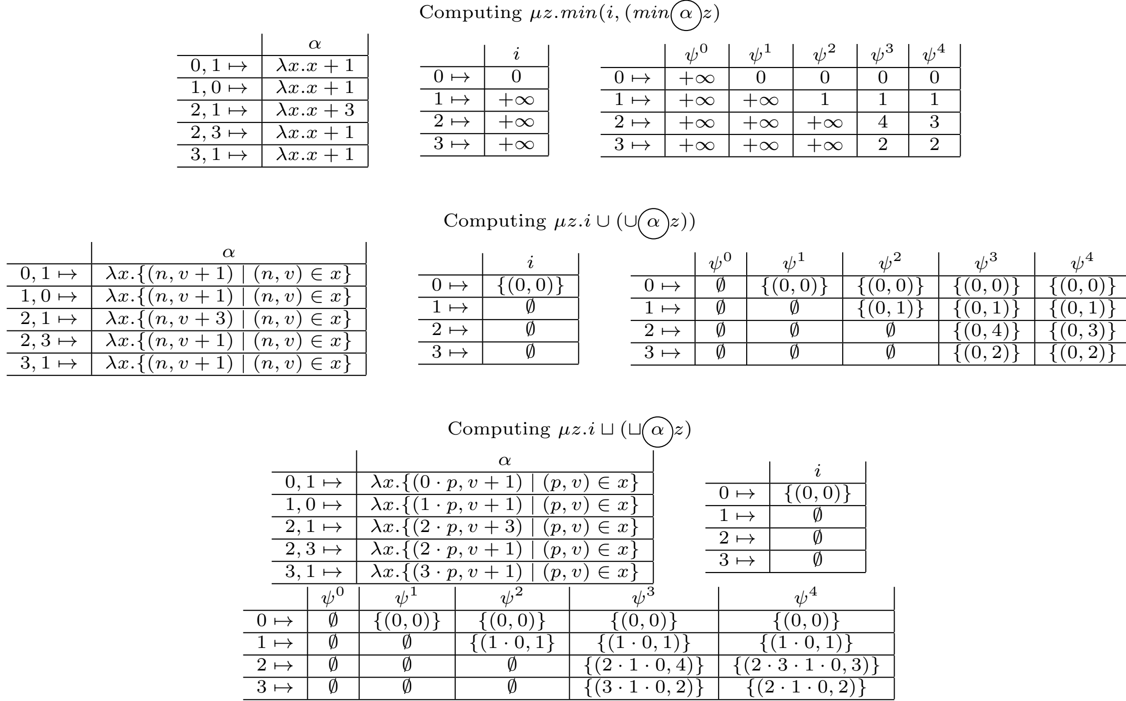
Generalising conjunction and disjunction to the multiplicative and additive operations of a semiring yields the modalities in [16].

A set of typical example formulas can be obtained by instantiating the simple pattern formula $\mu z.i \sqcup (\sqcup\bigcirc z)$ in different domains that happen to be complete lattices, such as the ones in Example 2.2 and 2.7, where $\sqcup$ is a well-defined join operation.

Example 3.3. Formula $\mu z.i \vee (\vee\bigcirc z)$ is equivalent to the temporal property *"eventually i"* if we use the Boolean domain. Indeed the formula amounts to the fixpoint characterization of the Computation Tree Logic (CTL) formula $\mathbf{EF}i$ stating that, starting from the current state (represented as a node of the graph), there is an execution path in the transition system (represented as a graph) and some state on that path that has property i. Other well-known temporal properties can be similarly obtained and used for agents to check, for example, complex reachability properties. Recall that the entire CTL and CTL* temporal logic languages can be encoded in the μ-calculus. For example, the CTL formula $\mathbf{AG}i$ stating that *"starting from the current state (represented as a node of the graph), all execution paths in the transition system (represented as a graph) are such that all states along those paths have property i"* can be easily expressed as $\nu z.i \wedge (\wedge\bigcirc z)$, and similarly for other properties.

Example 3.4. Formula $\mu z.min(i, (min\bigcirc z))$ yields the minimal value in a totally ordered numerical domain, like the tropical semiring. This can be used by agents to discover the best value for some attribute. A typical example could be the discovery of a leader agent, in case totally ordered agent identifiers are used. In the same setting the maximal value could be obtained with $\nu z.max(i, (max\bigcirc z))$.

Example 3.5. In a set-based domain, $\mu z.i \cup (\cup\bigcirc z)$ can provide the union of all elements in a graph. For example, if the set A coincides with the set N of nodes and i records each node's identifier as a singleton, then the formula can be used for each node to compute the



Computing $\mu z.min(i,(min\,\textcircled{\alpha}\,z)$

	α
$0,1 \mapsto$	$\lambda x.x+1$
$1,0 \mapsto$	$\lambda x.x+1$
$2,1 \mapsto$	$\lambda x.x+3$
$2,3 \mapsto$	$\lambda x.x+1$
$3,1 \mapsto$	$\lambda x.x+1$

	i
$0 \mapsto$	0
$1 \mapsto$	$+\infty$
$2 \mapsto$	$+\infty$
$3 \mapsto$	$+\infty$

	ψ^0	ψ^1	ψ^2	ψ^3	ψ^4
$0 \mapsto$	$+\infty$	0	0	0	0
$1 \mapsto$	$+\infty$	$+\infty$	1	1	1
$2 \mapsto$	$+\infty$	$+\infty$	$+\infty$	4	3
$3 \mapsto$	$+\infty$	$+\infty$	$+\infty$	2	2

Computing $\mu z.i \cup (\cup\textcircled{\alpha}z))$

	α
$0,1 \mapsto$	$\lambda x.\{(n,v+1) \mid (n,v) \in x\}$
$1,0 \mapsto$	$\lambda x.\{(n,v+1) \mid (n,v) \in x\}$
$2,1 \mapsto$	$\lambda x.\{(n,v+3) \mid (n,v) \in x\}$
$2,3 \mapsto$	$\lambda x.\{(n,v+1) \mid (n,v) \in x\}$
$3,1 \mapsto$	$\lambda x.\{(n,v+1) \mid (n,v) \in x\}$

	i
$0 \mapsto$	$\{(0,0)\}$
$1 \mapsto$	$\emptyset$
$2 \mapsto$	$\emptyset$
$3 \mapsto$	$\emptyset$

	ψ^0	ψ^1	ψ^2	ψ^3	ψ^4
$0 \mapsto$	$\emptyset$	$\{(0,0)\}$	$\{(0,0)\}$	$\{(0,0)\}$	$\{(0,0)\}$
$1 \mapsto$	$\emptyset$	$\emptyset$	$\{(0,1)\}$	$\{(0,1)\}$	$\{(0,1)\}$
$2 \mapsto$	$\emptyset$	$\emptyset$	$\emptyset$	$\{(0,4)\}$	$\{(0,3)\}$
$3 \mapsto$	$\emptyset$	$\emptyset$	$\emptyset$	$\{(0,2)\}$	$\{(0,2)\}$

Computing $\mu z.i \sqcup (\sqcup\textcircled{\alpha}z)$

	α
$0,1 \mapsto$	$\lambda x.\{(0 \cdot p,v+1) \mid (p,v) \in x\}$
$1,0 \mapsto$	$\lambda x.\{(1 \cdot p,v+1) \mid (p,v) \in x\}$
$2,1 \mapsto$	$\lambda x.\{(2 \cdot p,v+3) \mid (p,v) \in x\}$
$2,3 \mapsto$	$\lambda x.\{(2 \cdot p,v+1) \mid (p,v) \in x\}$
$3,1 \mapsto$	$\lambda x.\{(3 \cdot p,v+1) \mid (p,v) \in x\}$

	i
$0 \mapsto$	$\{(0,0)\}$
$1 \mapsto$	$\emptyset$
$2 \mapsto$	$\emptyset$
$3 \mapsto$	$\emptyset$

	ψ^0	ψ^1	ψ^2	ψ^3	ψ^4
$0 \mapsto$	$\emptyset$	$\{(0,0)\}$	$\{(0,0)\}$	$\{(0,0)\}$	$\{(0,0)\}$
$1 \mapsto$	$\emptyset$	$\emptyset$	$\{(1 \cdot 0,1\}$	$\{(1 \cdot 0,1)\}$	$\{(1 \cdot 0,1)\}$
$2 \mapsto$	$\emptyset$	$\emptyset$	$\emptyset$	$\{(2 \cdot 1 \cdot 0,4)\}$	$\{(2 \cdot 3 \cdot 1 \cdot 0,3)\}$
$3 \mapsto$	$\emptyset$	$\emptyset$	$\emptyset$	$\{(3 \cdot 1 \cdot 0,2)\}$	$\{(2 \cdot 1 \cdot 0,2)\}$

Figure 2: Computation of SMuC optimal path formulas in detail: cost of the optimal path (above), optimal cost and actual goal node (center), optimal path and its cost (bottom).

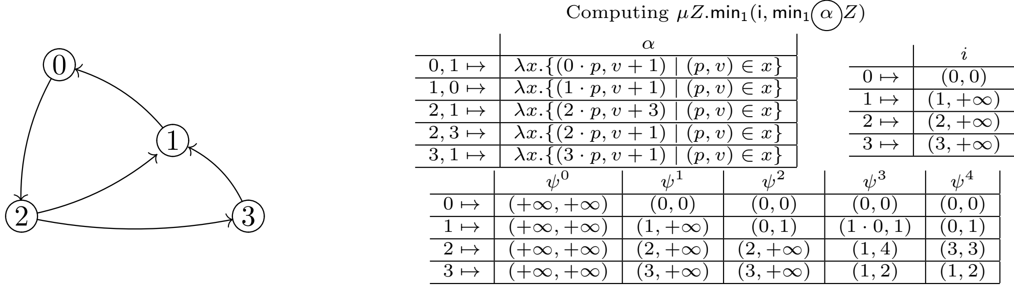
set of nodes that it can reach. Assume now that i records some arbitrary set, say the set of nodes that every node happens to know. Then the formula $\nu z.i \cap (\cap\bigcirc z)$ can be used to compute the set of nodes that every node knows.

The computation of some of the above formulas can be found in Fig. 1. The figure includes a simple graph (underlying a field), the instance of each formula ψ on the considered domain of interpretation, and the details of the computation of each of the formulas. For each computation a table is used to represent the value of i on each node and the evaluation of the (fixpoint) formula by iteration, as a sequence $\psi^0, \psi^1, \psi^2, \ldots$

Another interesting family of properties can be obtained with the slightly extended pattern formula $\mu z.i \sqcup (\sqcup\textcircled{\alpha}z)$. This pattern formula can be used for several shortest-path related properties, considering i to be a label providing information related to each node being or not a goal node and α providing a function that takes care of composing the cost of traversing an edge.

Example 3.6. Considering $\langle \mathbb{R}^+ \cup \{+\infty\}, \geq, +\infty, 0\rangle$ as domain, i to yield 0 for goal nodes and $+\infty$ for the rest of the nodes, and α being a function that adds the cost of traversing an edge, formula $\mu z.min(i,(min\,\textcircled{\alpha}\,z)$ yields the shortest distance to a goal node.

Example 3.7. The actual sets of reachable goal nodes with their distances can be obtained if we consider the evaluation of $\mu z.i \cup (\cup\textcircled{\alpha}z))$ under the Hoare power domain of the Cartesian product of nodes and costs, i.e. $P^H(N \times (\mathbb{R}^+ \cup \{+\infty\}))$. In words, the domain



Computing $\mu Z.\mathsf{min}_1(\mathsf{i}, \mathsf{min}_1 \textcircled{\alpha} Z)$

	α
$0,1 \mapsto$	$\lambda x.\{(0 \cdot p, v+1) \mid (p,v) \in x\}$
$1,0 \mapsto$	$\lambda x.\{(1 \cdot p, v+1) \mid (p,v) \in x\}$
$2,1 \mapsto$	$\lambda x.\{(2 \cdot p, v+3) \mid (p,v) \in x\}$
$2,3 \mapsto$	$\lambda x.\{(2 \cdot p, v+1) \mid (p,v) \in x\}$
$3,1 \mapsto$	$\lambda x.\{(3 \cdot p, v+1) \mid (p,v) \in x\}$

	i
$0 \mapsto$	$(0,0)$
$1 \mapsto$	$(1,+\infty)$
$2 \mapsto$	$(2,+\infty)$
$3 \mapsto$	$(3,+\infty)$

	ψ^0	ψ^1	ψ^2	ψ^3	ψ^4
$0 \mapsto$	$(+\infty,+\infty)$	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$
$1 \mapsto$	$(+\infty,+\infty)$	$(1,+\infty)$	$(0,1)$	$(1 \cdot 0,1)$	$(0,1)$
$2 \mapsto$	$(+\infty,+\infty)$	$(2,+\infty)$	$(2,+\infty)$	$(1,4)$	$(3,3)$
$3 \mapsto$	$(+\infty,+\infty)$	$(3,+\infty)$	$(3,+\infty)$	$(1,2)$	$(1,2)$

Figure 3: Computation of shortest path spanning tree.

consists of non-dominated sets of pairs (node,cost). In this case i should be $\{(n,0)\}$ for every goal node n and $\emptyset$ for the rest of the nodes, and α should be similar as before (pointwise applied to every pair, only on the second component of each pair).

Example 3.8. The actual set of paths to the goal nodes can be obtained in a similar way, by considering formula $\mu z.i \sqcup (\sqcup \textcircled{\alpha} z)$ under the Hoare power domain of the Cartesian product of paths and costs, i.e. $P^H((N^* \cup \{\bullet\}) \times (\mathbb{R}^+ \cup \{+\infty\}))$. In words, the domain consists of non-dominated sets of pairs (path,cost). In this case i should be $\{(n,0)\}$ for every goal node n and $\emptyset$ for the rest of the nodes, and α should be such that $\alpha(n,n')$ is a function that prefixes n to a path. Since the Hoare power domain deals with non-dominated paths, loops (that would require special treatment with an ordinary power construct) are implicitly dealt (i.e. extending a set of paths can only consist of adding non-dominated paths and loops can only worsen the cost of existing ones).

The computation of the above formulas can be found in Fig. 2, which follows a similar schema as Fig. 1 but includes in addition the interpretation of edge label α.
The set of shortest path formulas we have discussed above is very flexible and can be used indeed to build useful field structures. A typical example are spanning trees. Indeed, a spanning tree can be computed as follows.

Example 3.9. Consider as domain the lexicographical Cartesian product of domains $\langle \mathbb{N} \cup \{+\infty\}, \leq, +\infty, 0\rangle$ and $N_{\sqsubseteq} = \langle N, \sqsubseteq_N, n_{|N|}, n_1\rangle$ given by some total ordering $n_1 \sqsubseteq_N n_2 \sqsubseteq_N .. \sqsubseteq_N n_{|N|}$ on nodes. Then, the formula for computing a (shortest path-based) spanning tree is $\mu Z.\mathsf{min}_1(\mathsf{i}, \mathsf{min}_1 \textcircled{\alpha} Z)$, where label i is $(n,0)$ for the root n and $(n' + \infty)$ for any other node n' (i.e. the root points at itself with cost 0, while all other nodes point at themselves with infinite cost) and edge label α is used to append the source of an edge to the path component of a tuple (p,v). The computation of the formula on a simple example is illustrated in Fig. 3.

The exhaustive presentation of our case study in Section 4 exploits some of the above examples to solve a complex task.

Now that we have provided an illustrative set of examples, we are ready to formalise the meaning of formulas. Given a formula Ψ and an environment ρ we say that Ψ is ρ *closed* if ρ is defined for the free formula variables of Ψ.

Definition 3.10 (semantics of SMuC formulas)**.** Let F be a field and ρ be an enviroment. The semantics of ρ-closed SMuC formulas is given by the interpretation function $[\![\cdot]\!]^F_\rho : \Psi \to N_F \to A_F$ defined by

$$\begin{array}{rcl}
[\![i]\!]^F_\rho & = & I_F(i) \\
[\![z]\!]^F_\rho & = & \rho(z) \\
[\![f(\Psi_1,\ldots,\Psi_k)]\!]^F_\rho & = & \lambda n.[\![f]\!]_{A_F}([\![\Psi_1]\!]^F_\rho\, n,..,[\![\Psi_k]\!]^F_\rho\, n) \\
[\![g \textcircled{\alpha} \Psi]\!]^F_\rho & = & \lambda n.[\![g]\!]_{A_F}(\{I_F(\alpha)(n,n')([\![\Psi]\!]^F_\rho(n')) \mid (n,n') \in E_F\}) \\
[\![g \text{\ding{108}}_{\alpha} \Psi]\!]^F_\rho & = & \lambda n.[\![g]\!]_{A_F}(\{I_F(\alpha)(n',n)([\![\Psi]\!]^F_\rho(n')) \mid (n',n) \in E_F\}) \\
[\![\mu z.\Psi]\!]^F_\rho & = & \mathit{lfp}\ \lambda \mathsf{f}.[\![\Psi]\!]^F_{\rho[^{\mathsf{f}}/_z]} \\
[\![\nu z.\Psi]\!]^F_\rho & = & \mathit{gfp}\ \lambda \mathsf{f}.[\![\Psi]\!]^F_{\rho[^{\mathsf{f}}/_z]}
\end{array}$$

where *lfp* and *gfp* stand for the least and greatest fixpoint, respectively[1].

As usual for such fixpoint formulas, the semantics is well defined if so are all fixpoints. As we mentioned in the previous section we require that all functions $\lambda \mathsf{f}.[\![\Psi]\!]^F_{\rho[^{\mathsf{f}}/_z]}$ are monotone and continuous.

It is worth to remark that if we restrict ourselves to a semiring-valued field, then all SMuC formulas provide such guarantees.

Lemma 3.1 (semiring monotony)**.** Let F be a field, where F_A is a semiring, I_F is such that $I_F(\alpha)(e)$ is monotone for all $\alpha \in L_A, e \in E_A$, $\mathcal{M}$ contains only function symbols that are obtained by composing additive and multiplicative operations of the semiring, ρ be an environment and Ψ be a ρ-closed formula. Then, every function $\lambda \mathsf{f}.[\![\Psi]\!]^F_{\rho[^{\mathsf{f}}/_z]}$ is monotone and continuous.

3.3. **Robustness against unavailabilty.**

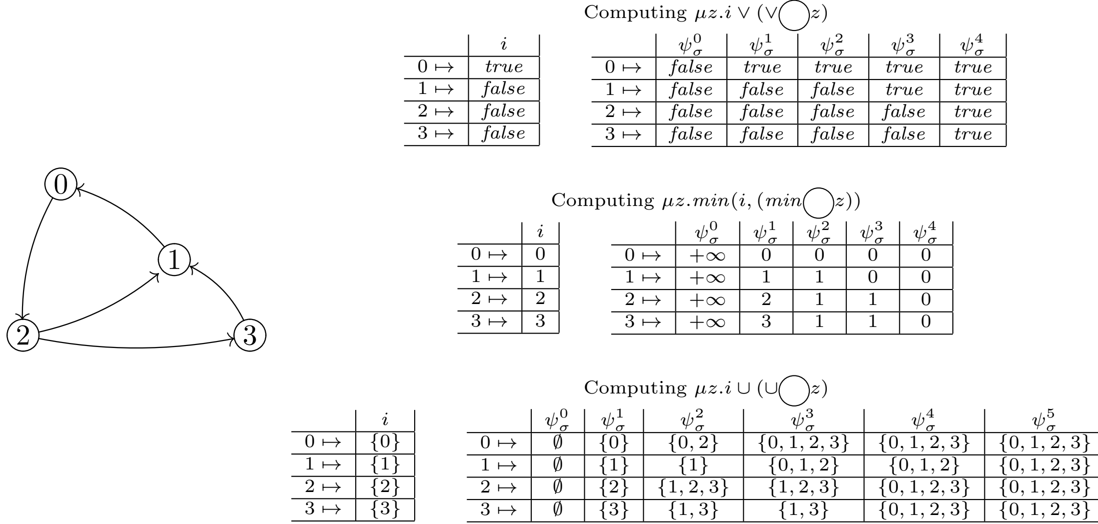
This section provides a formal characterisation of unavailability and robustness results against situations where nodes are allowed to proceed at different speeds. For this purpose we introduce the notions of a *pattern* and a *strategy* which formalise the ability of nodes to participate to an iteration in the computation of a fixpoint. A pattern and the corresponding pattern-restricted application formalise which nodes will participate in an iteration. Note that the unavailability of a node n does not mean that other nodes will ignore n when aggregating values. We assume that the underlying system will ensure that the last known attributes of n will be available (e.g. through a cache-based mechanism).

Definition 3.11 (pattern)**.** Let F be a field. A *pattern* is a subset $\pi \subseteq N_F$ of the nodes of F.

Definition 3.12 (pattern-restricted application)**.** Let F be a field, $\pi \subseteq N_F$ be a pattern and $\psi : (N \to A) \to N \to A$ be an update function. The π-restricted application of ψ, denoted ψ_π is a function $\psi__ : 2^{N_F} \to (N_F \to A_F) \to N_F \to A_F$ such that:

$$\psi_\pi\, \mathsf{f}\, n = \begin{cases} \psi\, \mathsf{f}\, n & \text{if } n \in \pi \\ \mathsf{f}\, n & \text{otherwise} \end{cases}$$

[1]Notice that the value $I_F(\alpha)(n,n')([\![\Psi]\!]^F_\rho(n'))$ will be passed to function $[\![g]\!]_{A_F}$ with a multiplicity which depends on the number of nodes n'.



Computing $\mu z.i \vee (\vee\bigcirc z)$

	i
$0 \mapsto$	*true*
$1 \mapsto$	*false*
$2 \mapsto$	*false*
$3 \mapsto$	*false*

	ψ_σ^0	ψ_σ^1	ψ_σ^2	ψ_σ^3	ψ_σ^4
$0 \mapsto$	*false*	*true*	*true*	*true*	*true*
$1 \mapsto$	*false*	*false*	*false*	*true*	*true*
$2 \mapsto$	*false*	*false*	*false*	*false*	*true*
$3 \mapsto$	*false*	*false*	*false*	*false*	*true*

Computing $\mu z.min(i, (min\bigcirc z))$

	i
$0 \mapsto$	0
$1 \mapsto$	1
$2 \mapsto$	2
$3 \mapsto$	3

	ψ_σ^0	ψ_σ^1	ψ_σ^2	ψ_σ^3	ψ_σ^4
$0 \mapsto$	$+\infty$	0	0	0	0
$1 \mapsto$	$+\infty$	1	1	0	0
$2 \mapsto$	$+\infty$	2	1	1	0
$3 \mapsto$	$+\infty$	3	1	1	0

Computing $\mu z.i \cup (\cup\bigcirc z)$

	i
$0 \mapsto$	$\{0\}$
$1 \mapsto$	$\{1\}$
$2 \mapsto$	$\{2\}$
$3 \mapsto$	$\{3\}$

	ψ_σ^0	ψ_σ^1	ψ_σ^2	ψ_σ^3	ψ_σ^4	ψ_σ^5
$0 \mapsto$	$\emptyset$	$\{0\}$	$\{0,2\}$	$\{0,1,2,3\}$	$\{0,1,2,3\}$	$\{0,1,2,3\}$
$1 \mapsto$	$\emptyset$	$\{1\}$	$\{1\}$	$\{0,1,2\}$	$\{0,1,2\}$	$\{0,1,2,3\}$
$2 \mapsto$	$\emptyset$	$\{2\}$	$\{1,2,3\}$	$\{1,2,3\}$	$\{0,1,2,3\}$	$\{0,1,2,3\}$
$3 \mapsto$	$\emptyset$	$\{3\}$	$\{1,3\}$	$\{1,3\}$	$\{0,1,2,3\}$	$\{0,1,2,3\}$

Figure 4: Computation of the three SMuC formulas of Fig. 4 under a fair strategy.

The intuition is that ψ_π applies a node valuation f on the nodes in π and ignores the rest. Note that $\psi_{N_F} = \psi$.

The concepts of pattern and pattern-restricted application are extended to sequences of executions that we call here *strategies*. They can be seen as schedules determining which processes will be able to update their values in each step of an execution.

Definition 3.13 (strategy)**.** Let F be a field. A *strategy* σ is a possibly infinite sequence of patterns $\pi_1, \pi_2, \ldots$ of F.

As usual, we use ϵ for the empty sequence and, given a possibly infinite sequence $\sigma = \pi_1, \pi_2, \ldots$, we use σ_i for the i-th element (i.e. π_i), σ^i for the suffix starting from the i-th element (i.e. $\pi_i, \pi_{i+1}, \ldots$) and $\sigma[i..j]$ for the sub-sequence that starts from the i-th element and ends at the j-th element (i.e. $\pi_i, \ldots, \pi_j$ if $i \leq j$ and ϵ otherwise).

Definition 3.14 (strategy-restricted application)**.** Let F be a field, σ be a finite strategy and $\psi : (N \to A) \to N \to A$ be an update function. The σ-restricted application of ψ, denoted ψ_σ is defined as:

$$\psi_\sigma = \begin{cases} \bot & \text{if } \sigma = \epsilon \\ \psi_\pi\, \psi_{\sigma'} & \text{if } \sigma = \sigma', \pi \end{cases}$$

The intuition is that update function ψ is applied to bottom $k = |\sigma|$ times, every time according to the i-th element σ_i of the strategy σ, for i ranging from 1 to k. As an example, Fig. 4 shows how the computations of Fig. 1 would be carried out under the strategy σ where node 1 participates in odd rounds only, i.e. σ is such that $\sigma_i = \{0, 1, 2, 3\}$ if i is odd and $\sigma_i = \{0, 2, 3\}$ otherwise. One can see that the main effect of a strategy is to delay the achievement of the fixpoint.

Clearly, not all strategies correspond to realistic executions in practice. The situation under consideration here is that the underling middleware guarantees that nodes are able to participate infinitely often in the computations of formulas. We formalise this by considering a class of *fair* strategies. The term *fair* is inspired by classical notions of *fairness* such as those used in concurrency theory, operating systems and formal verification.

Definition 3.15 (fair strategy)**.** Let F be a field. A strategy $\sigma = \pi_1, \pi_2, \dots$ is *fair* with respect to a field F iff $\forall n \in N_F$ the set $\{k \mid n \in \pi_k\}$ is infinite.

Intuitively, a fair strategy allows every node to execute infinitely often. Clearly, only infinite strategies can be fair, but when the fixpoint is reached in a finite number of steps, the strategy can be considered as finite/terminated. The example strategy discussed above is fair.

In the following we shall present lemmas and theorems related to the robustness of least fixpoints under fair strategies. Analogous results can be obtained for greatest fixpoints.

Lemma 3.2 (monotony of pattern-restricted application)**.** Let F be a field, $\psi : (N \to A) \to N \to A$ be a monotone update function, f, $\mathsf{f1}$, $\mathsf{f2} : N \to A$ be node valuations, π, π_1, π_2 be patterns, and $n \in N_F$ be a node. Then, the following holds:

(i) Function ψ_π is monotone;
(ii) $n \in \pi_1 \Leftrightarrow n \in \pi_2$ implies $\psi_{\pi_1} \mathsf{f}\, n = \psi_{\pi_2} \mathsf{f}\, n$;
(iii) $n \in \pi_1 \Leftrightarrow n \in \pi_2$ and $\mathsf{f}_1 \sqsubseteq \mathsf{f}_2$ implies $\psi_{\pi_1} \mathsf{f}_1\, n \sqsubseteq \psi_{\pi_2} \mathsf{f}_2\, n$.

We recall that, for the sake of a lighter notation we drop subscripts when they are clear from the context. This means that in the above definition and in what follows, for example, something like $\bot \sqsubseteq \psi$ abbreviates $\bot_{N \to A} \sqsubseteq_{N \to A} \psi$ in an unambigous manner (our notational convention for ψ determines the field domain under consideration).

Lemma 3.3 (pattern-restricted application bounds)**.** Let F be a field, $\psi : (N \to A) \to N \to A$ be a monotone update function, σ be a finite strategy and π be a pattern. Then it holds $\psi_\sigma \sqsubseteq \psi_\pi\, \psi_\sigma \sqsubseteq \psi\, \psi_\sigma$

A consequence of the lemma is that strategy-restricted applications yield partially ordered chains.

Corollary 3.1 (strategy-restricted applications yield chains)**.** *Let F be a field, $\psi : (N \to A) \to N \to A$ be a monotone update function and σ be an infinite strategy. Then, the sequence $\bot \sqsubseteq \psi_{\sigma[1..1]} \sqsubseteq \psi_{\sigma[1..2]} \sqsubseteq \dots$ is actually a partially ordered chain.*

The above results allow us to state now one of the main results of the paper, i.e. that fair strategies and the ideal situation (all nodes are always available) have the same bounds.

Theorem 3.16 (bounds under fair strategies)**.** *Let F be a field, with N_F finite, $\psi : (N \to A) \to N \to A$ be a monotone update function and σ be a fair strategy. Then all elements of the partially ordered chains $\bot \sqsubseteq \psi_{\sigma[1..1]} \sqsubseteq \psi_{\sigma[1..2]} \sqsubseteq \dots$ and $\bot \sqsubseteq \psi \bot \sqsubseteq \psi^2 \bot \sqsubseteq \dots$ have the same set of upper bounds and hence the same least upper bound, namely the least fixpoint of ψ.*

The final result is the formalisation of robustness against node unavailability.

Theorem 3.17 (robustness against unavailability)**.** *Let F be a field with finite set of nodes N_F and field domain A with finite partially ordered chains only, σ be a fair strategy and $\psi : (N \to A) \to N \to A$ be a monotone update function. Then the partially ordered chain $\bot \sqsubseteq \psi_{\sigma[1..1]} \sqsubseteq \psi_{\sigma[1..2]} \sqsubseteq \dots$*

(i) *stabilizes to its least upperbound f;*
(ii) *its least upper bound f does not depend on the fair strategy: we always have $f = lfp\, \psi$.*

This result is of utmost importance in practice since it guarantees that, under reasonable conditions, the computation of fixpoints can be performed asynchronously without the need of synchronising the agents, which may proceed at different relative speeds.

The most significant restriction is the one that requires finite chains and stabilisation, namely the recognition of a finite prefix σ' of σ which is enough to reach the fixpoint, i.e. such that $\psi\psi'_\sigma = \psi'_\sigma$. However, we envisage in practice the use of libraries of function and formula patterns that already ensure those properties, so that the final user can just combine them at will. A typical example could be, for instance, to consider semirings as we did in [16] with finite discrete domains. Indeed, the semiring additive and multiplicative operations can both be used as aggregation functions (since both work on multisets), both are monotone (which ensures well-definedness of fixpoints), and the restriction to finite discrete domains ensures finite chains.

3.4. **Robustness against failures.** We now consider the possibility of agent failure. We restrict ourselves to the common case where not only agents can stay inactive for a (finite) period, but when they resume they enter a backup state they had in a previous iteration, possibly the initial one ($\bot$). Thus we exclude the erroneous behavior caused by an agent entering a completely unknown state, or occurring when the structure of the network is in any form damaged or modified. We prove that the stable, fixpoint state does not change, provided the system at some point enters a condition where no more failures occur.

Definition 3.18 (failure sequence)**.** Let σ be a finite strategy and $\psi : (N \to A) \to N \to A$ be an update function. A σ *failure sequence of* ψ, denoted ς_σ, is defined as:

$$\varsigma_\sigma n = \begin{cases} \bot_A & \text{if } \sigma = \epsilon \\ \psi\varsigma_{\sigma'} n & \text{if } \sigma = \sigma', \pi \text{ and } n \in \pi \\ \varsigma_{\sigma'} n & \text{if } \sigma = \sigma', \pi \text{ and } n \notin \pi \\ \varsigma_{\sigma'} n & \text{if } \sigma = \sigma', \sigma'', \pi \text{ with } \sigma'' \neq \epsilon \text{ and } n \notin \pi. \end{cases} \tag{3.1}$$

Now let us extend ς_σ and ψ_σ to infinite sequences $\tilde{\sigma}$:

$$\overline{\varsigma}_{\pi_1,\pi_2\ldots} = \varsigma_\epsilon, \varsigma_{\pi_1}, \varsigma_{\pi_1,\pi_2}, \ldots \qquad \overline{\psi}_{\pi_1,\pi_2\ldots} = \psi_\epsilon, \psi_{\pi_1}, \psi_{\pi_1,\pi_2}, \ldots$$

An infinite sequence $\overline{\varsigma}_{\tilde{\sigma}}$ is called a $\tilde{\sigma}$ *failure sequence of* ψ. We call it *safe* if, for all finite σ' larger than some finite $\hat{\sigma}$, the fourth option in equation 3.1 has not been used for computing $\varsigma_{\sigma'}$. The idea is that, from some $\hat{\sigma}$ on failures never occur again and all nodes can progress, possibly skipping some rounds (but never returning to a backup/initial state).

Notice that in the above definition ς_σ is not functional and models a non-deterministic presence of errors. Indeed if $n \notin \pi$ the value of node n can be left unchanged (third option), can be initialized to $\bot_A$ (fourth option with $\sigma' = \epsilon$), or it can be assigned any previous value (fourth option). If $n \in \pi$ then it is updated using ψ. Also, if the fourth option is never used in a sequence, then $\varsigma_\sigma = \psi_\sigma$, namely a failure sequence is just an ordinary chain. Conversely, observe that a generic failure sequence is not a chain, since it is not necessarily increasing.

We can now prove our main result in this setting: a safe σ failure sequence of ψ has a least upper bound which is the fixpoint of ψ.

Theorem 3.19 (least upper bound of a safe failure sequence)**.** *Let F be a field, with N_F finite, $\psi : (N \to A) \to N \to A$ be a monotone update function, $\tilde{\sigma}$ be a fair strategy and $\overline{\varsigma}_{\tilde{\sigma}}$ be some infinite safe $\tilde{\sigma}$ failure sequence of ψ. Then $\overline{\varsigma}_{\tilde{\sigma}}$ has a least upper bound which is the least fix point of ψ.*

Similar results can be provided for additional failure situations. For example, the above results could be adapted to the situation in which errors can occur infinitely often, but sufficiently long progress is guaranteed between errors. This would be provided by a middleware that enforces a phase restore-progress-backup after each failure recovery.

More general kinds of failure, concerning unrecoverable failure of some node, or possible changes in the structure of the network, cannot be recovered significantly. Specific recovery actions must be foreseen for maintaining networks with failures, which apply to our approach just as they concern similar coordination styles.

3.5. **SMuC Programs.** The atomic computations specified by SMuC formulas can be embedded in any language. To ease the presentation we present a global calculus where atomic computations are embedded in a simple imperative language similar to the While [23] language, a core imperative language for imperative programming which has formal semantics.

Definition 3.20 (SMuC syntax)**.** The syntax of SMuC is given by the following grammar

$$P, Q \quad ::= \quad \mathsf{skip} \mid i \leftarrow \Psi \mid P \; ; \; Q \mid \mathsf{if} \; \Psi \; \mathsf{then} \; P \; \mathsf{else} \; Q \mid \mathsf{until} \; \Psi \; \mathsf{do} \; P$$

where $i \in \mathcal{L}$, Ψ is a SMuC formula (cf. Def 3.2).

The simple imperative language we used in [16] featured $\mathsf{agree} \cdot \mathsf{on}$ variants of the traditional control flow constructs in order to remark the characteristics of the case study used there, where the global control flow depended on the existence of agreements among all agents in the field. This can be of course an expensive operation, which depends on the diameter of the graph.

The use of traditional control flow constructs does not restrict the possibility to deal with agreements. Indeed, the existence of an agreement of all agents on an expression Ψ can be easily verified by using the expression $eq(\Psi, eq \textcircled{id} \Psi, eq \blacktextcircled{id} \Psi) \neq none$, where id is the identity function and eq is a function equationally defined as follows:

$$eq(\emptyset) = any \qquad eq(\{a\}) = a \qquad eq(\{a, any\} \cup B) = eq(\{a\} \cup B)$$
$$eq(\{a, b\} \cup B) = none \qquad eq(\{a, a\} \cup B) = eq(\{a\} \cup B)$$

In words, the expression $eq(\Psi, eq \textcircled{id} \Psi, eq \blacktextcircled{id} \Psi) \neq none$ is true on all agents whenever Ψ is evaluated to the same value on each node and its neighbours (both through in- and out-going edges). Note that similar expressions can be used if one is interested in agreements that exclude certain values, say in a set B. The corresponding condition expression would be $eq(\Psi, eq \textcircled{id} \Psi, eq \blacktextcircled{id} \Psi) \not\subseteq B \cup \{none\}$.

The semantics of the calculus is straightforward, along the lines of While [23] with fields (and their interpretation functions) playing the role of memory stores.

The semantics of our calculus is a transition system whose states are pairs of calculus terms and fields and whose transitions $\rightarrow \subseteq (P \times \mathcal{F})^2$ are defined by the rules of Table 1. Most rules are standard. Rule IfT and IfF are similar to the usual rules for conditional branching. It is worth to remark that the condition Ψ must evaluate to $true$ in each agent n in the field F for the then branch to be taken, otherwise the else branch is followed. Similarly for the until operator (cf. rules UntilF and until). In particular, the until finishes when all agents agree on $true$, namely when the formula Ψ is evaluated to $\lambda n.true$. States of the form $\langle \mathsf{skip}, I \rangle$ represent termination.

(μStep)	$\dfrac{[\![\Psi]\!]_{\emptyset}^{I_F} = f \quad I'_F = I_F[^f/_i]}{\langle i \leftarrow \Psi, F\rangle \rightarrow \langle \mathsf{skip}, F[^{I'_F}/_{I_F}]\rangle}$
(Seq1)	$\dfrac{\langle P, F\rangle \rightarrow \langle P', F'\rangle}{\langle P;Q, F\rangle \rightarrow \langle P';Q, F'\rangle}$
(Seq2)	$\dfrac{\langle P, F\rangle \rightarrow \langle P', F'\rangle}{\langle \mathsf{skip};P, F\rangle \rightarrow \langle P', F'\rangle}$
(IfT)	$\dfrac{[\![\Psi]\!]_{\emptyset}^{F} \ = \ \lambda n.true}{\langle \mathsf{if}\ \Psi\ \mathsf{then}\ P\ \mathsf{else}\ Q, F\rangle \rightarrow \langle P, F\rangle}$
(IfF)	$\dfrac{[\![\Psi]\!]_{\emptyset}^{F} \ \neq \ \lambda n.true}{\langle \mathsf{if}\ \Psi\ \mathsf{then}\ P\ \mathsf{else}\ Q, F\rangle \rightarrow \langle Q, F\rangle}$
(UntilF)	$\dfrac{[\![\Psi]\!]_{\emptyset}^{F} \ \neq \ \lambda n.true}{\langle \mathsf{until}\ \Psi\ \mathsf{do}\ P, F\rangle \rightarrow \langle (P\ ;\ \mathsf{until}\ \Psi\ \mathsf{do}\ P), F\rangle}$
(UntilT)	$\dfrac{[\![\Psi]\!]_{\emptyset}^{F} \ = \ \lambda n.true}{\langle \mathsf{until}\ \Psi\ \mathsf{do}\ P, F\rangle \rightarrow \langle \mathsf{skip}\ , F\rangle}$

Table 1: Rules of the operational semantics

4. SMuC at Work: Rescuing Victims

The left side of Fig. 6 depicts a simple instance of the considered scenario. There, victims are rendered as black circles while landmarks and rescuers are depicted via grey and black rectangles respectively. The length of an edge in the graph is proportional to the distance between the two connected nodes. The main goal is to assign rescuers to victims, where each victim may need more than one rescuer and we want to minimise the distance that rescuers need to cover to reach their assigned victims. We assume that all relevant information of the victim rescue scenario is suitably represented in field F. More details on this will follow, but for now it suffices to assume that nodes represent rescuers, victims or landmarks and edges represent some sort of direct proximity (e.g. based on visibility w.r.t. to some sensor).

We will use semirings as suitable structures for our operations. It is worth to remark that in practice it is convenient to define A as a Cartesian product of semirings, e.g. for differently-valued node and edge labels. This is indeed the case of our case study. However, in order to avoid explicitly dealing with these situations (e.g. by resorting to projection functions, etc.) which would introduce a cumbersome notation, we assume that the corresponding semiring is implicit (e.g. by type/semiring inference) and that the interpretation of functions and labels are suitably specialised. For this purpose we decorate the specification in Fig. 5 with the types of all labels.

```
/* Initialisations */
finish ← false;
until finish do
    /* 1st Stage: Establishing the distance to victims */
    source ← victim ? (0, self) : (+∞, self);
    D ← μZ.min₁(source, min₁(dst)Z);

    /* 2nd Stage: Computing the rescuers paths */
    rescuers ← μZ.init ∪ ⋃(grd)Z;

    /* 3rd Stage: Engaging rescuers */
    /* engaging the rescuers */
    engaged ← μZ.choose ∪ ⋃(cgr)Z;
    /* updating victims and available rescuers */
    victim′ ← victim;
    victim ← victim ∧ ¬saved;
    rescuer ← rescuer ∧ (engaged = ∅);
    /* determining termination */
    finish ← (victim′ == victim);

/* 4th Stage: Checking success */
if ¬victim
    /* ended with success */
else
    /* ended with failure */
```

```
/* Semiring types of node labels */
        source, D  :  N → T ×₁ N_⊑
    init, rescuers :  N → 2^{T×N*}
 choose, engaged   :  N → 2^{N*}
   victim, victim′ :  N → 𝔹
   rescuer, finish

/* Semiring types of edge labels */
               dst :  E → T ×₁ N_⊑ → T ×₁ N_⊑
               grd :  E → 2^{T×N*} → 2^{T×N*}
               cgr :  E → 2^{N*} → 2^{N*}
```

Figure 5: Robot Rescue SMuC Program

We now describe the coordination strategy specified in the algorithm of Fig. 5. Many of the formulas that the algorithm uses are based on the formula patterns described in Examples 3.3–3.9. The algorithm consists of a loop that is repeated until an iteration does not produce any additional matching of rescuers to victims. The body of the loop consist of different stages, each characterised by a fixpoint computation.

1st Stage: Establishing the distance to victims. In the first stage of the algorithm the robots try to establish their closest victim. Such information is saved in to D, which is valued over the lexicographical Cartesian product of domains $T = \langle \mathbb{N} \cup \{+\infty\}, \leq, +\infty, 0\rangle$ and $N_{\sqsubseteq} = \langle N, \sqsubseteq_N, n_{|N|}, n_1\rangle$ given by some total ordering $n_1 \sqsubseteq_N n_2 \sqsubseteq_N .. \sqsubseteq_N n_{|N|}$ on nodes. In order to compute D, some information is needed on nodes and arrows of the field. For example, we assume that the boolean attribute victim initially records whether the node is a victim or not, while source is used to store that victims initially point to themselves with no cost, while the rest of the nodes point to themselves with infinite cost. The edge label dst is defined as $I(\mathsf{dst})(n, n') = \lambda(v, m).(distance(n, n') + v, n')$ where $distance(n, n')$ is the weight of (n, n'). Intuitively, dst provides a function to add the cost associated to the transition. The second component of the value encodes the direction to go for the shortest path, while the total ordering on nodes is used for solving ties.

The desired information is then computed as $\mathsf{D} \leftarrow \mu Z.\mathsf{min}_1(\mathsf{source}, \mathsf{min}_1 \textcircled{\scriptsize dst} Z)$. This formula is similar to the formulas presented in Examples 3.6–3.9. Here min_1 is the join of domain $T \times_1 N_{\leq_N}$, specifically for a set $B \subseteq (\mathbb{R} \cup \{+\infty\}) \times N$ the function min_1 is defined as $\mathsf{min}_1(B) = (a, n) \in B$ such that $\forall (a', n') \in B : a \leq a'$ and if $a = a'$ then $n \leq n'$.

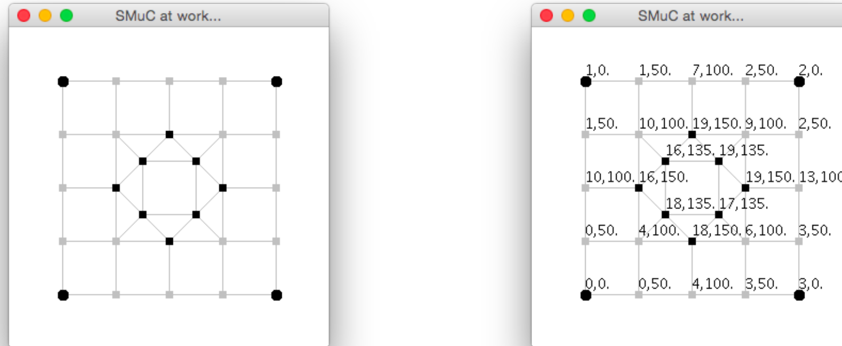


Figure 6: Execution of Robot Rescue SMuC Program (part 1)

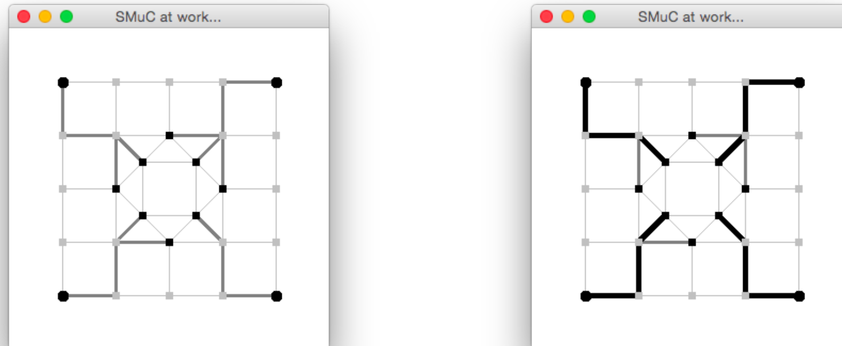


Figure 7: Execution of Robot Rescue SMuC Program (part 2)

At the end of this stage, D associates each element with the distance to its closest victim and the identity of the neighbour on the next edge in the shortest path. In the right side of Fig. 6 each node of our example is labeled with the computed distance. We do not include the second component of D (i.e. the identity of the closest neighbour) to provide a readable figure. In any case, the closest victim is easy to infer from the depicted graph: the closest victim of the rescuer in the top-left corner of the inner box formed by the rescuers is the victim at the top-left corner of the figure, and respectively for the top-right, bottom-left and bottom-right corners.

2nd Stage: Computing the rescuers paths to the victims. In this second stage of the algorithm, the robots try to compute, for every victim v, which are the paths from every rescuer u to v — but only for those u for which v is the closest victim — and the corresponding costs, as established by D in the previous stage. Here we use the semiring $2^{T \times N^*}$ with union as

additive operator, i.e. $\langle 2^{T \times N^*}, \cup, \cap, T \times N^*, \emptyset \rangle$. We use here decorations init and grd whose interpretation is defined as

- $I(\mathsf{init})n = \text{if } n \in \text{rescuer then } \{(n', \epsilon) \mid \mathsf{D}(n) = (u, n')\} \text{ else } \emptyset$;
- $I(\mathsf{grd})(n, n') = \lambda C. \text{ if } \mathsf{D}(n) = (u, n') \wedge \mathsf{D}(n') = (u', n'') \text{ then } n; C \text{ else } \emptyset$, where operation ; is defined as $n; C = \{(cost, n; path) \mid (cost, path) \in C\}$.

The idea of label *rescuers* is to compute, for every node n, the set of rescuers whose path to their closest victim passes through n (typically a landmark). However, the name of a rescuer is meaningless outside its neighbourhood, thus a path leading to it is constructed instead. In addition, each rescuer is decorated with its distance to its closest victim. Function init associates to a rescuer its name and its distance, the empty set to all the other nodes. Function grd checks if an arc (n, n') is on the optimal path to the same victim n'' both in n and n'. In the positive case, the rescuers in n are considered as rescuers also for n', but with an updated path; in the negative case they are discarded.

On the left side of Fig. 7 the result of this stage is presented. Since all edges of the graph have a corresponding edge in the opposite direction we have depicted the graph as it would be undirected. The edges that are part of a path from one rescuer to a victim are now marked (where the actual direction is left implicit for simplicity). We can notice that some victims can be reached by more than one rescuer.

3rd Stage: Engaging the rescuers. The idea of the third stage of the algorithm is that each victim n, which needs k rescuers, will choose the k closest rescuers, if there are enough, among those that have selected n as target victim. For this computation we use the decorations choose and cgr.

- $I(\mathsf{choose})(n) = \text{if } n \in \mathit{victim} \text{ and } \mathsf{saved}(n) \text{ then } \mathit{opt}(\mathsf{rescuers}(n), \mathit{howMany}(n)) \text{ else } \emptyset$, where:
  - $\mathsf{saved}(n) = \left|\mathsf{rescuers}(n)\right| \leq \mathit{howMany}(n)$ and $\mathit{howMany}(n)$ returns the number of rescuers n needs;
  - $\mathit{opt}(C, k) = \{\mathit{path} \mid (\mathit{cost}, \mathit{path}) \in C \text{ and } \left|\{(\mathit{cost}', \mathit{path}') \mid (\mathit{cost}', \mathit{path}') < (\mathit{cost}, \mathit{path})\}\right| < k\}$
    where $(\mathit{cost}, \mathit{path}) < (\mathit{cost}', \mathit{path}')$ if $\mathit{cost} < \mathit{cost}'$ or $\mathit{cost} = \mathit{cost}'$ and $\mathit{path} < \mathit{path}'$, and paths are totally ordered lexicographically;
- $I(\mathsf{cgr}(n, n') = \lambda C.\{\mathit{path} \mid n; \mathit{path} \in C\}$.

Intuitively, choose allows a victim n that has enough rescuers to choose and to record the paths leading to them. The annotation cgr associates to each edge (n, n') a function to select in a set C of paths those of the form $n; \mathit{path}$.

The computation in this step is $\mathsf{engaged} \leftarrow \mu Z.\mathsf{choose} \cup \bigcup \textcircled{\scriptsize cgr} Z$, which computes the desired information: in each node n we will have the set of rescuer-to-victim paths that pass through n and that have been chosen by a victim.

The result of this stage is presented in the right side of Fig. 7. Each rescuer has a route, that is presented in the figure with black edges, that can be followed to reach the assigned victim. Again, for simplicity we just depict some relevant information to provide an appealing and intuitive representation.

Notice that this phase, and the algorithm, may fail even if there are enough rescuers to save some additional victims. For instance if there are two victims, each requiring two rescuers, and two rescuers, the algorithm fails if each rescuer is closer to a different victim.

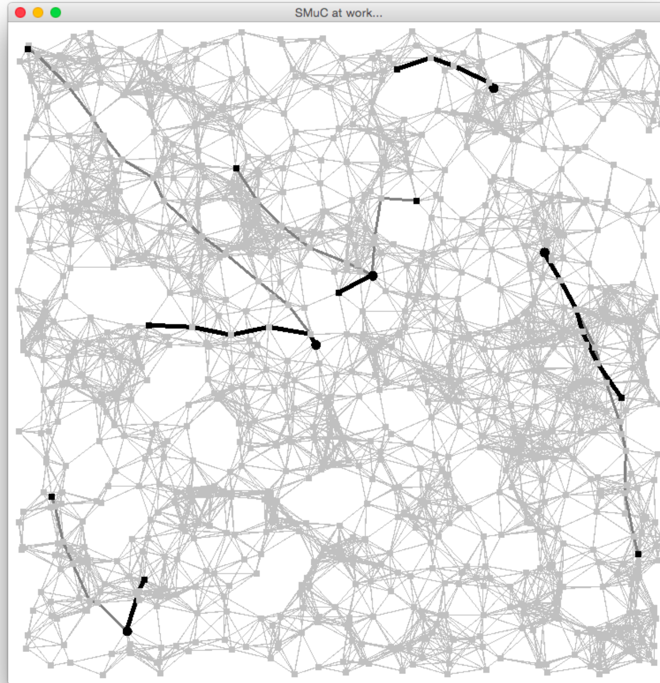


Figure 8: Execution of Robot Rescue SMuC Program on a random graph

These three stages are repeated until there is agreement on whether to finish. The termination criteria is that an iteration did not update the set of victims. In that case the loop terminates and the algorithm proceeds to the last stage.

4th Stage: Checking succes. The algorithm terminates with success when $victim' = \emptyset$ and with failure when $victim'$ is not empty. In Fig. 8 we present the result of the computation of program of Fig. 5 on a randomly generated graph composed by $1,000$ landmarks, 5 victims and 10 rescuers, which actually need just one rescuer. We can notice that, each victim can be reached by more than one rescuer and that the closer one is selected since.

5. On Distributing SMuC Computations

We discuss in this section the distributed implementation of SMuC computations. Needless to say, an obvious implementation would be based on a *centralised* algorithm. In particular, the nodes could initially send all their information to a centralised coordinator that would construct the field, carry on the SMuC computations, and distribute the results back to the nodes. This solution is easy to realise and could be based on our prototype which indeed performs a centralised, global computation, as a sequential program acting on the field. However, such a solution has several obvious drawbacks: first, it creates a bottleneck in the coordinator. Second, there are many applications in which the idea of constructing the whole field is not feasible and each agent needs to evolve independently.

To provide a general framework for the distributed evaluation of SMuC computations we introduce some specific forms of programs that simplify their evaluation in a fully distributed environment. We first define the class of *elementary formulas* which are formulas that do not contain neither fixpoints nor formula variables, and that consist of at most one operator.

Definition 5.1 (elementary formulas)**.** A SMuC formula Ψ is *elementary* if it has the following form:

$$\Psi_e ::= j \mid f(j,\dots,j) \mid g \textcircled{\alpha} j \mid g \text{\ding{192}} j$$

with $j \in \mathcal{L}$, $a \in \mathcal{L}'$, $f \in \mathcal{M}$.

Elementary formulas are used as expressions in *simple assignment* SMuC programs. These are programs that only use *elementary formulas.* Moreover, in this class of programs, boolean guards in until and if − then − else statements are always *node labels.*

Definition 5.2 (simple assignment programs)**.** A SMuC program P is in *simple assignment form* (SAF) if has the following syntax:

$$S, R \quad ::= \quad \mathsf{skip} \mid i \leftarrow \Psi_e \mid S \;;\; R \mid \mathsf{if}\ i\ \mathsf{then}\ S\ \mathsf{else}\ R \mid \mathsf{until}\ i\ \mathsf{do}\ S \mid \mathsf{free}(x,\dots,x)$$

where $i, j, x \in \mathcal{L}$, and Ψ_e is a SMuC elementary formula (cf. Def 5.1).

Above, we introduced a new costruct (free) that is used to *deallocate* labels during a program evaluation. The role of this construct will be clear later when a distributed evaluation of SMuC programs is introduced. The operational semantics of Table 1 is extended to consider the following rule:

$$(\textsc{Free}) \qquad \frac{I'_F = I_F[^{\mathsf{undef}}/_{x_1},\dots,{}^{\mathsf{undef}}/_{x_n}]}{\langle \mathsf{free}(x_1,\dots,x_n), F\rangle \to \langle \mathsf{skip}, F[^{I'_F}/_{I_F}]\rangle}$$

when $\mathsf{free}(x_1,\dots,x_n)$ is executed all the labels $x_1,\dots, x_n$, are removed from the interpretation in F. This is denoted with the value undef.

One can also observe that a SMuC program S in SAF does not contain any fixpoint formulas. However, if we consider only fields with a domain field with finite partially ordered chains only, like in Theorem 3.17, this does not limit the expressive power of our language. Indeed, fixpoints can be explicitly computed by using the other constructs of SMuC. In Table 2 function $\mathcal{P}$ (and the auxiliary function $\mathcal{A}$) is defined to transform a SMuC program P into an equivalent program S in SAF. This transformation introduces a set of *auxiliary* node labels denoted by x_k that we assume to be distinct from all the other symbols and not occurring in P. From now on we will use $\mathcal{X} \subseteq \mathcal{L}$ to denote the set of this auxiliary *node labels*. To guarantee the appropriate allocation of these symbols, function $\mathcal{P}$ (and the auxiliary function $\mathcal{A}$) is parametrised with a counter c that indicates the number of *auxiliary* symbols already introduced in the transformation. The result of $\mathcal{P}(P)_c$ is a pair $[S, c']$: S is a SAF SMuC program, while c' indicates the number of symbols allocated in the transformation. In the following we will use S_P to denote that, for some c and c', $\mathcal{P}(P)_c = [S_P, c']$. Similary, we will use S^i_Ψ to denote that, for some c and c', $\mathcal{A}(\Psi)^i_c = [S^i_\Psi, c']$.

Function $\mathcal{P}$ is inductively defined on the syntax of SMuC programs. The translation of skip and P;Q are straightforward. In the first case $\mathcal{P}$ does not change skip without allocating any auxiliary label while in the second case the translation of P;Q is obtained as the sequentialisation of $\mathcal{P}(P)_c$ and $\mathcal{P}(Q)_{c'}$, where c' indicates the amount of symbols allocated in the translation of P. The macro wait is used between the two processes. This is defined as follows:

$$\mathsf{wait}(x) \equiv x \leftarrow true; \mathsf{until}\ x\ \mathsf{do}\ \mathsf{skip}$$

$$\mathcal{P}(\mathsf{skip})_c = [\mathsf{skip}, c] \qquad \frac{\mathcal{A}(\Psi)^i_c = [S, c']}{\mathcal{P}(i \leftarrow \Psi)_c = S; \mathsf{free}(x_c, \dots, x_{c'-1})}$$

$$\frac{\mathcal{P}(P)_c = [S, c'] \qquad \mathcal{P}(Q)_{c'} = [R, c'']}{\mathcal{P}(P;Q)_c = [S;\mathsf{wait}(x_{c''});R, c''+1]}$$

$$\frac{\mathcal{A}(\Psi)^{x_c}_{(c+1)} = [R, c'] \quad \mathcal{P}(P)_{c'} = [S_1, c''] \qquad \mathcal{P}(Q)_{c'''} = [S_2, c''']}{\mathcal{P}(\mathsf{if}\ \Psi\ \mathsf{then}\ P\ \mathsf{else}\ Q)_c = [R;\mathsf{if}\ x_c\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, c''']}$$

$$\frac{\mathcal{A}(\Psi)^{x_c}_{(c+1)} = [R, c'] \quad \mathcal{P}(P)_{c'} = [S, c'']}{\mathcal{P}(\mathsf{until}\ \Psi\ \mathsf{do}\ P)_c = [R;\mathsf{until}\ x_c\ \mathsf{do}\ S;\mathsf{wait}(x_{c''});R, c''+1]}$$

$$\mathcal{A}(i)^j_c = [j \leftarrow i, c] \qquad \frac{\mathcal{A}(\Psi_1)^{x_c}_{c+1} = [S_1, c_1] \quad \cdots \quad \mathcal{A}(\Psi_n)^{x_{c_{n-1}}}_{c_{n-1}+1} = [S_n, c_n]}{\mathcal{A}(f(\Psi_1, \dots, \Psi_n))^j_c = [S_1\ ; \cdots ;\ S_n\ ;\ j \leftarrow f(x_c, \dots, x_{c_{n-1}}), c_n]}$$

$$\frac{\mathcal{A}(\Psi)^{x_c}_{c+1} = [S, c']}{\mathcal{A}(g \textcircled{\alpha} \Psi)^j_c = [S\ ;\ i \leftarrow g \textcircled{\alpha} x_c, c']} \qquad \frac{\mathcal{A}(\Psi)^{x_c}_{c+1,\rho} = [S, c']}{\mathcal{A}(g \text{●}\!\alpha\, \Psi)^j_c = [S\ ;\ i \leftarrow g \text{●}\!\alpha\, x_c, c']}$$

$$\frac{\mathcal{A}(\Psi[^{x_c}/_z])^{x_{c+1}}_{c+2} = [S, c']}{\mathcal{A}(\mu z.\Psi)^j_c = \left[\begin{array}{l} x_c \leftarrow \bot\ ; \\ x_{c+1} \leftarrow \bot\ ; \\ x_{c'} \leftarrow \mathit{false}; \\ \mathsf{until}\ x_{c'}\ \mathsf{do} \\ \qquad x_c \leftarrow x_{c+1}; \\ \qquad S\ ; \\ \qquad x_{c'} \leftarrow x_c = x_{c+1} \\ j \leftarrow x_{c+1} \end{array}, c'+1\right]}$$

$$\frac{\mathcal{A}(\Psi[^{x_c}/_z])^{x_{c+1}}_{c+2} = [S, c']}{\mathcal{A}(\nu z.\Psi)^j_c = \left[\begin{array}{l} x_c \leftarrow \top\ ; \\ x_{c+1} \leftarrow \top\ ; \\ x_{c'} \leftarrow \mathit{false}; \\ \mathsf{until}\ x_{c'}\ \mathsf{do} \\ \qquad x_c \leftarrow x_{c+1}; \\ \qquad S\ ; \\ \qquad x_{c'} \leftarrow x_c = x_{c+1} \\ j \leftarrow x_{c+1} \end{array}, c'+1\right]}$$

Table 2: Function $\mathcal{P}$ that transforms a SMuC program P in SAF.

The use of this statement has no impact in the global evaluation of a SMuC program. However, when distributed executions will be considered, a wait statement can be used as a barrier for a global synchronisation in the field.

Each assignment $i \leftarrow \Psi$ is translated into a program that first evaluates formula Ψ and then assigns the result to i. After that all the auxiliary labels used in the computation of Ψ are deallocated. The program computing Ψ is obtained via function $\mathcal{A}(\Psi)^i_c$, that is also

defined in Table 2 and that is described below. Function $\mathcal{P}$ translates a statement of the form if Ψ then P else Q in a program that first evaluates formula Ψ storing the result in the auxiliary label x_c which is then used to select either the translation of P or the translation of Q. Translation of until Ψ do P is similar to the previous one. At the end of the same body the construct wait(x) is used. Again, the role of this statement will be clear later when a distributed execution of SMuC programs is considered.

Function $\mathcal{A}$ is defined inductively on the syntax of formulas Ψ and takes as parameter a counter c, that is used to allocate auxiliary labels. Similarly to function $\mathcal{P}$, function $\mathcal{A}$ returns a pair consisting of a SMuC program and of a counter of allocated auxiliary node labels. When Ψ is a label i, $\mathcal{A}(\Psi)_c^j$ (which arises from the translation of $j \leftarrow i$ in $\mathcal{P}(i \leftarrow \Psi)_c$) is just $j \leftarrow i$ and no auxiliary variable is created. When Ψ is $f(\Psi_1, \dots, \Psi_n)$ (resp. $g \textcircled{\alpha} \Psi_1$, $g \text{\ding{108}}_{\alpha} \Psi_1$), $\mathcal{A}(\Psi)_c^j$ consists of the SMuC program that uses auxiliary variables $x_c, \dots, x_{c_{n-1}}$ (resp. x_c) to store the evaluation to Ψ_i and then assings to j the evaluation of the *simple* formula $f(x_c, \dots, x_{c_{n-1}})$ (resp. $g \textcircled{\alpha} x_c$, $g \text{\ding{108}}_{\alpha} x_c$). The program that evaluates $\mu z.\Psi$ (resp. $\nu z.\Psi$) uses two auxiliary variables, namely x_c and x_{c+1}. The former is initialised to $\bot$ (resp. $\top$), the latter will contain the evaluation of Ψ performed by $\mathcal{A}(\Psi[^{x_c}/_z])_{c+2}^{x_{c+1}}$.This evaluation continues until x_c will be equal to x_{c+1} (If not, x_c is assigned to x_{c+1}). This means that after $i > 0$ iterations x_c and x_{c+1} contain the evaluation of approximants $(i-1)$ and i of $\mu z.\Psi$ (resp. $\nu z.\Psi$).

The following Lemma guarantees that any formula Ψ, when interpreted over a field F satisfying conditions of Theorem 3.17, can be evaluated by the SAF SMuC program obtained from the application of function $\mathcal{A}$.

Lemma 5.1. Let F be a field with field domain A with finite chains only, Ψ a formula, $c \in \mathbb{N}$, and label i. Let $\mathcal{A}(\Psi)_c^i = [S, c']$, then:

$$[\![\Psi]\!]_\emptyset^F = f \Leftrightarrow \langle S, F \rangle \to^* \langle \mathsf{skip}, F' \rangle \text{ and } I_{F'}(i) = f$$

Lemma 5.2. Let F be a field with field domain A with finite chains only, for any P the following holds:

- if $\langle P, F \rangle \to \langle P', F' \rangle$ then $\langle S_P, F \rangle \to^* \langle S_{P'}, F'' \rangle$ and $F' = F'' \backslash \mathcal{X}$;
- if $\langle S_P, F \rangle \to \langle S', F' \rangle$ then there exist P' and F'' such that $\langle S', F' \rangle \to^* \langle S_{P'}, F'' \rangle$ and $\langle P, F \rangle \to \langle P', F'' \backslash \mathcal{X} \rangle$.

Above, we use $F' \backslash \mathcal{X}$ to refer to the field obtained from F' by erasing all the labels in $\mathcal{X}$.

5.1. **Asynchronous agreement.** We describe now a technique that, by relying on a specific structure, can be used to perform SMuC computations in an fully distributed way. Here we assume that each node n in the field has its own computational power and that it can interact with its neighbours to locally evaluate its part of the field. However, to guarantee a correct execution of the program, a global coordination mechanism is needed. The corner stone of the proposed algorithm is a *tree-based* infrastructure that spans the complete field. In this infrastructure each node/agent, that is referenced by a unique identifier, is responsible for the coordination of the computations occurring in its sub-tree. In the rest of this section we assume that this *spanning tree* is computed in a *set-up* phase executed when the system is deployed.

Definition 5.3 (distributed field infrastructure)**.** A *distributed field infrastructure* is a pair $\langle F, T\rangle$ where F is a field while $T \subseteq N_F \times N_F$ is a *spanning tree* of the underlying undirected graph $(N, E \cup E^{-1})$ of F, where $(x, y) \in T$ if and only if x is the parent of y in the spanning tree. Given a tree $T \subseteq N \times N$, let $children(T, n) = \{n' | (n, n') \in T\}$, $rel(T, n) = \{n' | (n, n') \in T \vee (n', n) \in T\}$, and $parent(T, n) = n'$ if and only if $(n', n) \in T$.

Given a *distributed field infrastructure* $\langle F, T\rangle$, we will use a variant of the *Dijkstra-Scholten* algorithm [13] for termination detection to check if a global agreement on the evaluation of a given node label x has been reached or not, where x takes value on the standard boolean lattice $\{true, false\}$. To check if an agreement has been reached or not each node uses two elements: an *agreement store* $\chi : N \to L_N \to \mathbb{N} \to \{\mathsf{undef}, ?true, true, false\}$ and a *counter* $\kappa : L_N \to \mathbb{N}$. Via the *agreement store* χ each node stores the status of the agreement of labels collected from its relatives in the spanning tree T. Since an agreement on the same label can be iterated in a SMuC program, and to avoid confusions among different iterations, a different value is stored for each iteration. Counter κ is then used to count the iterations associated with a label x. If χ_n is the *agreement store* used by node n, $\chi_n(n', k, x)$ is evaluated to:

- undef, when n does not know the state of evaluation of label x at n' after k iterations;
- $?true$ when n' and all the nodes in its subtree have evaluated x to *true* at iteration k;
- *false* when at least one node in the field has evaluated x to *false* at iteration k;
- *true* when at iteration k an agreement has been reached on $x = true$.

If a node n at iteration k evaluates x to *false*, a message is sent to the relatives of n in T. If at the same iteration k, the evaluation of x at n is *true*, for each child n', $\chi_n(n', k, x) = ?true$, χ_n is updated to let $\chi_n(n, k, x) = ?true$ and a message is sent to its parent. When this information is propagated in the spanning tree from leaves to the root, the latter is able to identify if at iteration k an agreement on $x = true$ has been reached. After that, a notification message flows from the root to the leaves of T and each node n will set $\chi_n(n, k, x) = true$.

5.2. **Distributed execution of SMuC programs.** A distributed execution of a SMuC program over a *distributed field infrastructure* consists of a set of *fragments* executed over each node in the field. In a *fragment* each node computes its part of the field and interacts with its neighbours to exchange the computed values.

Definition 5.4 (distributed execution)**.** Let F be a *field*, we let $\mathcal{D}$ be the set of *distributed fragments* d of the form:

$$d_i = n[\ S \mid \iota : \chi : \kappa]$$

where $n \in N_F$, S is SAF SMuC program, $\iota : N \to L_N \to N \to A$ is a *partial interpretation* of *node labels* at n, χ is an *agreement store* and κ is an *agreement counter*. A *distributed execution* D for F is a subset of $\mathcal{D}$ such that D consists of one fragment $d = n[\ S \mid \iota : \chi : \kappa]$ (for some S, ι, χ and κ) for each node $n \in N_F$.

In a *fragment* $d = n[\ S \mid \iota : \chi : \kappa]$, S represents the portion of the program currently executed at n, ι is the portion of the field computed at n together with the part of the field collected from the neighbour of n, χ and κ are the structures described in the previous subsection to manage the agreement in SMuC computations.

The semantics of distributed SMuC programs is defined via the labelled transition relations defined in Tab. 3, Tab. 5 and Tab. 6. The behaviour of the single fragment d is

$$\text{(D-Seq1)}\ \frac{n[\ S \mid \iota : \chi : \kappa] \xrightarrow{\lambda}_{\langle F,T\rangle} n[\ S' \mid \iota' : \chi' : \kappa']}{n[\ \mathsf{skip}; S \mid \iota : \chi : \kappa] \xrightarrow{\lambda}_{\langle F,T\rangle} n[\ S' \mid \iota' : \chi' : \kappa']}$$

$$\text{(D-Seq2)}\ \frac{n[\ S \mid \iota : \chi : \kappa] \xrightarrow{\lambda}_{\langle F,T\rangle} n[\ S' \mid \iota' : \chi' : \kappa']}{n[\ S; R \mid \iota : \chi : \kappa] \xrightarrow{\lambda}_{\langle F,T\rangle} n[\ S'; R \mid \iota' : \chi' : \kappa']}$$

$$\text{(D-Step)}\ \frac{\Psi \mapsto_F^{n,\iota} v \qquad X = \{n' | (n,n') \in E_F \vee (n',n) \in E_F\}}{n[\ i \leftarrow \Psi \mid \iota : \chi : \kappa] \xrightarrow{\overline{\langle n,i,v\rangle @X}}_{\langle F,T\rangle} n[\ \mathsf{skip} \mid \iota[^v/_{(i)(n)}] : \chi : \kappa]}$$

$$\text{(D-Free)}\ \frac{\iota' = \iota[^{\lambda n.\mathsf{undef}}/_{x_1}, \cdots, ^{\lambda n.\mathsf{undef}}/_{x_n}]}{n[\ \mathsf{free}(x_1,\ldots,x_n) \mid \iota : \chi : \kappa] \xrightarrow{\tau}_{\langle F,T\rangle} n[\ \mathsf{skip} \mid \iota' : \chi : \kappa]}$$

$$\text{(D-IfT)}\ \frac{\kappa(i) = c \qquad \chi(n)(i)(k) = \mathit{true}}{n[\ \mathsf{if}\ i\ \mathsf{then}\ S\ \mathsf{else}\ R \mid \iota : \chi : \kappa] \xrightarrow{\tau}_{\langle F,T\rangle} n[\ S \mid \iota : \chi : \kappa[^i/_{c+1}]]}$$

$$\text{(D-IfF)}\ \frac{\kappa(i) = c \qquad \chi(n)(i)(k) = \mathit{false}}{n[\ \mathsf{if}\ i\ \mathsf{then}\ S\ \mathsf{else}\ R \mid \iota : \chi : \kappa] \xrightarrow{\tau}_{\langle F,T\rangle} n[\ R \mid \iota : \chi : \kappa[^i/_{c+1}]]}$$

$$\text{(D-UntilF)}\ \frac{\kappa(i) = c \qquad \chi(n)(i)(k) = \mathit{false}}{n[\ \mathsf{until}\ i\ \mathsf{do}\ S \mid \iota : \chi : \kappa] \xrightarrow{\tau}_{\langle F,T\rangle} n[\ S; \mathsf{until}\ i\ \mathsf{do}\ S \mid \iota : \chi : \kappa[^i/_{c+1}]]}$$

$$\text{(D-UntilT)}\ \frac{\kappa(i) = c \qquad \chi(n)(i)(k) = \mathit{true}}{n[\ \mathsf{until}\ i\ \mathsf{do}\ S \mid \iota : \chi : \kappa] \xrightarrow{\tau}_{\langle F,T\rangle} n[\ \mathsf{skip} \mid \iota : \chi : \kappa[^i/_{c+1}]]}$$

Table 3: Distributed Semantics of SMuC programs (*fragments*).

described by the transition relation $\xrightarrow{\cdot}_{\langle F,T\rangle} \subseteq \mathcal{D} \times \Lambda \times \mathcal{D}$ defined in Tab. 3 and Tab. 5 where Λ denotes the set of transition labels λ having the following syntax:

$$\lambda ::= \tau \mid m \mid \overline{m} \qquad m ::= \langle n,i,v\rangle @X \mid \langle n,i,c,v\rangle @X$$

where $n \in N_F$, $X \subseteq N_F$, $i \in \mathcal{L}_N$ and $v \in A_F$. Following a standard notation in process algebras, transition label τ identifies internal operations. A transition is labelled with $\overline{m}$ when a message m is sent. Finally, transitions labelled with m show how a fragment reacts when the message m is received.

Rules in Tab. 3 are similar to the corresponding ones in Tab. 1. However, while in the global semantics all elements of the field are synchronously evaluated, via the rules in Tab. 3 each *fragment* proceeds independently. Rules (D-Seq1) and (D-Seq2) are standard, while rule (D-Step) deserves more attention. It relies on relation $\mapsto_F^{n,\iota}$, defined in Tab. 4 that is used to evaluate an *elementary formula* Ψ_e in a given node n under a specific *partial interpretation* ι and *context* ρ. A formula Ψ_e can be directly evaluated when it is either a label i or a function $f(i_1,\ldots,i_n)$. In this case, the evaluation simply relies on the local evaluation ι. However, when Ψ_e is either $g \bigcirc\!\!\!\!\alpha\ i$ or $g \bullet\!\!\alpha\ i$, the evaluation is possible only when for each node n' in the poset (resp. preset) of n, $\iota(n')(i)$ is defined. When all these values are available, the evaluation of $g \bigcirc\!\!\!\!\alpha\ i$ (resp. $g \bullet\!\!\alpha\ i$) consists in the appropriate aggregation of values obtained from neighbours following outgoing ($\bigcirc$) or incoming ($\bullet$) edges and using the edge capability a with function g. When Ψ_e can be evaluated to value v, $n[\ i \leftarrow \Psi \mid \iota : \chi : \kappa]$ can perform a step and ι is updated to consider the new value for label i ($\iota[^v/_{(i)(n)}]$) while the

$$\frac{\iota(n)(i) = v}{i \mapsto_F^{n,\iota} v} \qquad \frac{\iota(n)(i_1) = v_1 \cdots \iota(n)(i_n) = v_n}{f(i_1, \ldots, i_n) \mapsto_F^{n,\iota} f(v_1, \ldots, v_n)}$$

$$\frac{\forall n' : (n, n') \in E_F : \iota(n')(i) \neq \mathsf{undef}}{g \textcircled{\alpha} i \mapsto_F^{n,\iota} [\![g]\!]_{A_F}(\{I_F(\alpha)(n, n')(\iota(n')(i)) \mid (n, n') \in E_F\})}$$

$$\frac{\forall n' : (n', n) \in E_F : \iota(n')(i) \neq \mathsf{undef}}{g \text{\ding{\alpha}} i \mapsto_F^{n,\iota} [\![g]\!]_{A_F}(\{I_F(\alpha)(n', n)(\iota(n')(i)) \mid (n, n') \in E_F\})}$$

Table 4: Distributed evaluation of formula.

message $\langle n, i, v\rangle @X$ is sent to all the neighbours of n ($X = \{n' | (n, n') \in E_F \vee (n', n) \in E_F\}$) to notify them that the value of i is changed at n.

We want to remark that rule (D-Step) can be applied only when all the values needed to evaluate formula Ψ_e are locally available. This means that an *assignment* can be a barrier in a distributed computation. To guarantee that in the execution only updated values are used, command free can be used. By applying rule (D-Free) all the labels passed as arguments are deallocated in ι.

Finally, rules (D-IfT), (D-IfF), (D-UntilF) and (D-UntilT) are as expected. We can notice that these rules are applied only when the label used as condition in the statement is evaluated in the agreement structure ($\chi(n)(c)(i) = v \in \{true, false\}$). Moreover, when one of these rule is applied, the *agreement counter* is updated ($\kappa[^i/_{c+1}]$) to avoid interferences with subsequent evaluations of the same statement.

Rules in Tab. 5 show how a node interacts with the other nodes in the field to check if an agreement has been reached or not in the field and implement the coordination mechanism discussed in the previous subsection. All these rules can be applied only when $isGuard(P, i)$ is true, namely when P is either if i then P' else Q' or until i do P'.

Rule (D-AgreeF1) is applied when in a node n the label i is evaluated to *false* and no information about the agreement at the current iteration $\kappa(i) = c$ is available ($\chi(i)(n)(c) = \mathsf{undef}$). In this case we can soon establish that the agreement is not reached at this iteration. Hence, this information is locally stored in the *agreement structure* ($\chi[^{false}/_{(n)(i)(c)}]$) while all the relatives in the spanning tree are notified with the message $\langle n, i, c, false\rangle$. Note that, after the application of rule (D-AgreeF1), either rule (D-IfF) or rule (D-UntilF) will be enabled.

When label i is evaluated to *true* ($\iota(i)(n) = true$) at n, data from the children of n are needed to establish whether an agreement is possible. If one of the children of n has notified n that the agreement on i has not been reached at iteration c (i.e. $\exists n' : (n, n') \in T : \chi(i)(n')(c) = false$) rule (D-AgreeF2) is applied. Like for rule (D-AgreeF1), the *agreement structure* is updated and all the relatives in the spanning tree are informed that an agreement has not been reached yet. Otherwise, when n has received information about a *local agreement* from all its children, i.e. $\forall n' : (n, n') \in T : \chi(i)(n')(c) = ?true$, the local agreement structure is updated accordingly. If n is not the root of T, rule (D-AgreeT) is applied and the parent of n is notified about the possible agreement. While, if n is the root of T an agreement is reached: rule (D-AgreeN1) is applied and all the children of n are then notified. At this point, rule (D-AgreeP) is used to propagate the status of the agreement from the root of the tree to its leaves.

$$\text{(D-AgreeF1)}\ \frac{\begin{array}{c} isGuard(S,i) \quad \iota(i)(n) = false \quad \kappa(i) = c \\ \chi(n)(i)(c) = \mathsf{undef} \quad X = \{n'|(n',n) \in T \vee (n,n') \in T\}\end{array}}{n[\ S \mid \iota : \chi : \kappa] \xrightarrow{\overline{\langle n,i,c,false\rangle @X}}_{\langle F,T\rangle} n[\ S \mid \iota : \chi[^{false}/_{(n)(i)(c)}] : \kappa]}$$

$$\text{(D-AgreeT)}\ \frac{\begin{array}{c} isGuard(S,i) \quad \iota(i)(n) = true \quad \kappa(i) = c \quad \chi(n)(i)(c) = \mathsf{undef} \\ \forall n' : (n,n') \in T : \chi(i)(n')(c) = ?true \quad (n',n) \in T\end{array}}{n[\ S \mid \iota : \chi : \kappa] \xrightarrow{\overline{\langle n,i,c,?true\rangle @\{n'\}}}_{\langle F,T\rangle} n[\ S \mid \iota : \chi[^{?true}/_{(n)(i)(c)}] : \kappa]}$$

$$\text{(D-AgreeF2)}\ \frac{\begin{array}{c} isGuard(S,i) \quad \iota(i)(n) = true \quad \kappa(i) = c \quad \chi(n)(i)(c) = \mathsf{undef} \\ \exists n' : (n,n') \in T : \chi(i)(n')(c) = false \quad X = \{n'|(n',n) \in T \vee (n,n') \in T\}\end{array}}{n[\ S \mid \iota : \chi : \kappa] \xrightarrow{\overline{\langle n,i,c,false\rangle @X}}_{\langle F,T\rangle} n[\ S \mid \iota : \chi[^{false}/_{(n)(i)(c)}] : \kappa]}$$

$$\text{(D-AgreeN1)}\ \frac{\begin{array}{c} isGuard(S,i) \quad \iota(i)(n) = true \quad \kappa(i) = c \quad \chi(n)(i)(c) = \mathsf{undef} \\ \forall n' : (n,n') \in T : \chi(i)(n')(c) = ?true \quad isRoot(n,T) \quad X = \{n'|(n,n') \in T\}\end{array}}{n[\ S \mid \iota : \chi : \kappa] \xrightarrow{\overline{\langle n,i,c,true\rangle @X}}_{\langle F,T\rangle} n[\ S \mid \iota : \chi[^{false}/_{(n)(i)(c)}] : \kappa]}$$

$$\text{(D-AgreeP)}\ \frac{\begin{array}{c} isGuard(S,i) \quad \kappa(i) = c \quad \chi(n)(i)(c) = ?v \\ (n',n) \in T \quad \chi(n')(i)(c) = v' \in \{true, false\} \quad X = \{n'|(n,n') \in T\}\end{array}}{n[\ S \mid \iota : \chi : \kappa] \xrightarrow{\overline{\langle n,i,c,v\rangle @X}}_{\langle F,T\rangle} n[\ S \mid \iota : \chi[^{v}/_{(n)(i)(c)}] : \kappa]}$$

Table 5: Distributed Semantics of SMuC programs (*agreement*).

The behaviour of a distribute execution D is described via the transition relation $\dot{\Rightarrow}_{\langle F,T\rangle} \subseteq 2^{\mathcal{D}} \times \Lambda \times 2^{\mathcal{D}}$ defined in Tab. 6. These rules are almost standard and describe the interaction among the fragments of a distributed execution D. In Tab. 6 we use $D = D_1 \oplus D_2$ to denote that $D = D_1 \cup D_2$ and $D_1 \cap D_2 = \emptyset$. In the following we will write $D_1 \Rightarrow_{\langle F,T\rangle} D_2$ to denote that $D_1 \overset{\lambda}{\Rightarrow}_{\langle F,T\rangle} D_2$, with $\lambda = \tau$ of $\lambda = \overline{m}$; $D_1 \Rightarrow^*_{\langle F,T\rangle} D_2$ is reflexive and transitive closure of $D_1 \Rightarrow_{\langle F,T\rangle} D_2$.

Rule (D-Comp) lifts transitions from the level of fragments to the level of distributed executions. Rules (R-Field) and (R-Agree) show how a fragment reacts when a new message is received, that is updating the partial field evaluation ι when a message of the form $\langle n', i, v\rangle$ is received, and updating the agreement structure χ when a message of the form $\langle n', i, c, v\rangle$ is received. Rules (I-Field) and (I-Agree) are used when a node is not the recipient of a message, while (D-Int), (D-Sync) and (D-Recv) describe possible interactions at the level of systems.

We are now ready to introduce the key result of this section. Namely, that one is always able to switch from a centralised evaluation to a distributed execution. Indeed, we can define a *projection operator* that given a SMuC program S and a field F, distributes the execution of S over the nodes in N_F. To define this operator, we need first to introduce the projection of a an interpretation I_F with respect to a set of nodes $X \subseteq N_F$.

Definition 5.5 (interpretation projection)**.** Let F a field, and $X \subseteq N_F$, the projection of I_F to X ($I_F\lfloor_X$) is the function ι such that:

$$\iota(i)(n) = \begin{cases} I_F(i)(n) & n \in X \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

$$\text{(D-Comp)}\ \frac{n[\ S \mid \iota : \chi : \kappa] \xrightarrow{\lambda} n[\ S' \mid \iota' : \chi' : \kappa']}{\{n[\ S \mid \iota : \chi : \kappa]\} \stackrel{\lambda}{\Rightarrow} \{n[\ S' \mid \iota' : \chi' : \kappa']\}}$$

$$\text{(R-Field)}\ \frac{n \in X \quad \iota(i) = \rho}{\{n[\ S \mid \iota : \chi : \kappa]\} \xRightarrow{\langle n', i, v\rangle @ X} \{n[\ S \mid \iota[^{\rho[^{n'}/_{v}]}/_{i} : \chi : \kappa]\}}$$

$$\text{(R-Agree)}\ \frac{n \in X}{\{n[\ S \mid \iota : \chi : \kappa]\} \xRightarrow{\langle n', i, c, v\rangle @ X} \{n[\ S \mid \iota : \chi[^{v}/_{(n)(i)(c)}] : \kappa]\}}$$

$$\text{(I-Field)}\ \frac{n \notin X}{\{n[\ S \mid \iota : \chi : \kappa]\} \xRightarrow{\langle n', i, v\rangle @ X} \{n[\ S \mid \iota : \chi : \kappa]\}}$$

$$\text{(I-Agree)}\ \frac{n \notin X}{\{n[\ S \mid \iota : \chi : \kappa]\} \xRightarrow{\langle n', i, c, v\rangle @ X} \{n[\ S \mid \iota : \chi : \kappa]\}}$$

$$\text{(D-Int)}\ \frac{D_1 \stackrel{\tau}{\Rightarrow} D_1'}{D_1 \oplus D_2 \stackrel{\tau}{\Rightarrow} D_1' \oplus D_2}$$

$$\text{(D-Sync)}\ \frac{D_1 \stackrel{\overline{m}}{\Rightarrow} D_1' \quad D_2 \stackrel{m}{\Rightarrow} D_2'}{D_1 \oplus D_2 \stackrel{\overline{m}}{\Rightarrow} D_1' \oplus D_2'}$$

$$\text{(D-Recv)}\ \frac{D_1 \stackrel{m}{\Rightarrow} D_1' \quad D_2 \stackrel{m}{\Rightarrow} D_2'}{D_1 \oplus D_2 \stackrel{m}{\Rightarrow} D_1' \oplus D_2'}$$

Table 6: Distributed Semantics of SMuC programs (*interactions*).

Definition 5.6 (program projection)**.** Let F a field, and S a program the projection of S to F ($S\lfloor_F$) is the function distributed execution D such that:

$$D = \{n[\ S \mid I_F\lfloor_{N(n)}\colon \lambda n.\lambda i.\lambda c.\mathsf{undef} : \lambda n.\lambda i.0] \mid n \in N_F\}$$

where for each $n \in N_F$, $N(n) = \{n' | (n, n') \in E_F \vee (n', n) \in E_F\}$ is the set of neighbour of n in F.

Given a distributed execution D we can reconstruct a global interpretation I_F for a given field F.

Definition 5.7 (lifted interpretation)**.** Let be F a field, and D be a distributed execution, $D\lceil_F$ denotes the interpretation I such that:

$$I(i)(n) = v \Leftrightarrow n[\ S \mid \iota : \chi : \kappa] \in D \wedge \iota(i)(n) = v$$

We say that a distributed execution D *agrees* with F if and only if $I_F = D\lceil_F$.

Given a distributed execution D it is sometime useful to check if all the nodes in D are executing exactly the same piece of code.

Definition 5.8 (aligned execution)**.** A distributed execution D is *aligned* at S if and only if: $\forall d \in D.d = n[\ S \mid \iota : \chi : \kappa]$, for some n, ι, χ and κ. We will write D_S to denote that the distributed execution D is aligned at S.

We can notice that while the global operational semantics defined in Table 1 is deterministic, the distributed version considered in this section is not. This is due to the fact

that each node can progress independently. However, we will see below that in the case of our interest, we can guarantee the existence of a common *flow*. We say that D_1 flows into D_2 if and only if any computation starting from D_1 eventually reaches D_2.

Definition 5.9 (execution flows)**.** A distributed execution D_1 *flows* into D_2 if and only if $D_1 \Rightarrow^*_{\langle F,T \rangle} D_2$ and

- either $D_1 = D_2$
- or, for any D' such that $D_1 \Rightarrow_{\langle F,T \rangle} D'$, D' flows into D_2;

To reason about *distributed computations* it is useful to introduce the appropriate notation that represents the execution of sequentially composed programs.

Definition 5.10 (concatenation)**.** Let D be a distributed execution and S a SAF SMuC program, we let $D; S$ denote:

$$\{n[\ S'; S \mid \iota : \chi : \kappa] \mid n[\ S' \mid \iota : \chi : \kappa] \in D\}$$

The following Lemma guarantees that sequential programs can be computed asynchronously, while preserving the final result, while each $\mathsf{wait}(x)$ represents a synchronization point.

Lemma 5.3. For any S_1 and S_2, and for any D_{S_1} that flows into D_{skip}, the following hold:

- if $D_{\mathsf{skip}}; S_2$ flows into D' then also $D_{S_1}; S_2$ flows into D';
- $D_{S_1}; \mathsf{wait}(x); S_2$ flows into $D_{\mathsf{skip}}; \mathsf{wait}(x); S_2$.

The following theorem guarantees that, when we consider a field F with field domain A with finite chains only, any global computation of a SMuC program P can be realised in terms of a distributed execution of its equivalent SAF program S_P. Moreover, any distributed execution will always converge to the same field computed by P.

Lemma 5.4. Let F be a field with field domain A with finite chains only, Ψ a formula, and S^x_Ψ the SAF SMuC program that evaluates Ψ, then any distributed execution $D_{S^x_\Psi}$ that agrees with F, $D_{S^x_\Psi}$ flows in a distributed execution D_{skip} such that $[\![\Psi]\!]^F_\emptyset = D_{\mathsf{skip}} \restriction_F (x)$.

Theorem 5.11. *Let F be a field with field domain A with finite chains only and T be a spanning tree of F, for any* SMuC *program P, for any D_{S_P} that agrees with F:*

(i) *if $\langle P, F \rangle \to \langle P', F' \rangle$ then $D_{S_P} \Rightarrow_{\langle F,T \rangle} D_{S_{P'}}$ and $D_{S_{P'}}$ agrees with F'.*
(ii) *For any D' such that $D_{S_P} \Rightarrow^*_{\langle F,T \rangle} D'$ there exists P' such that $D' \Rightarrow^*_{\langle F,T \rangle} D_{S_{P'}}$, $\langle P, F \rangle \to \langle P', F' \rangle$ and $D_{S_{P'}}$ agrees with F'.*

6. Related Works

In recent years, spatial computing has emerged as a promising approach to model and control systems consisting of a large number of cooperating agents that are distributed over a physical or logical space [3]. This computational model starts from the assumption that, when the density of involved computational agents increases, the underlying network topology is strongly related to the geometry of the space through which computational agents are distributed. Goals are generally defined in terms of the system's spatial structure. A main advantage of these approaches is that their computations can be seen both as working

on a single node, and as computations on the distributed data structures emerging in the network (the so-called "computational fields").

The first examples in this area is Proto [1, 2]. This language aims at providing links between local and global computations and permits the specification of the individual behaviour of a node, typically in a sensor-like network, via specific space-time operators to situate computation in the physical world. In [33, 10] a minimal core calculus, named *field calculus*, has been introduced to capture the key ingredients of languages that make use of computational fields.

The calculus proposed in this paper starts from a different perspective with respect to the ones mentioned above. In these calculi, computational fields result from (recursive) functional composition. These functions are used to compute a field, which may consists of a tuple of different values. Each function in the *field calculus* or in Proto plays a role similar to a SMuC formula. In our approach, each step of a SMuC program computes a different field, which is then used in the rest of the computation. This step is *completed* only when a *fixpoint* is reached. This is possible because in SMuC only specific functions over the appropriate domains are considered. This guarantees the existence of fixpoints and the possibility to identify a global stability in the field computation.

In SMuC a formula is evaluated when a *fixpoint* is reached. The resulting value is then used in the continuation of the program. The key feature of our approach is that our formulas have a declarative, global meaning, which restates in this setting the well understood interpretation of temporal logic and μ-calculus formulas, originally defined already on graphical structures, namely on labelled transition systems or on Kripke frames. In addition, the chains approximating the fixpoints can be understood as propagation processes, thus giving also a pertinent operational interpretation to the formulas.

Our approach is also reminiscent of the bulk synchronous model of computation, as adopted for instace by Pregel [19] and Giraph [8]. In this model computations consist of a sequence of iterations, called *supersteps*. During a *superstep* the framework evaluates a user-defined function at each vertex. Such evaluations are conceptually executed in parallel. After each superstep the computed value is propagated to the neighbours via the outgoing edges and used in the next supersteps. Approaches similar to ours can be found in the field of distributed and parallel programming, in particular in early works on distributed fixpoint computations. For example [5] presents a general distributed algorithm schema for the computation of fixpoints where iterations are not synchronised. The spirit of the work is similar to our results on robustness, but is focused on functions on domains of real numbers, while we consider the more general case of field domains. Another related example is [26], which presents solutions for distributed termination in asynchronous iterative algorithms for fixpoint computations.

In the *field calculus* the evaluation of a program yields a continuous stream of data (the field) that does not stop even if a fixpoint is reached. Under this perspective, one interesting property is the *self-stabilisation*, i.e. the ability of a *field* to reach a *stable* value after a perturbation or starting from some initial conditions. In [32, 9] *self-stabilisation* for the *field calculus* is studied. In these papers sufficient conditions for self-stabilisation are presented as the ability to react to changes in the environment finding a new stable state in finite time. A type-based approach is used to provide a correct checking procedure for self-stabilisation.

A deep comparison of the field calculus and the SMuC calculus would require an extensive discussion. We focus here on the two main aspects in which these calculi differ. First, even if the two calculi share the starting motivations, the *field-calculus* and SMuC operate at

two different levels of abstraction. Indeed, while the former aims at defining a general and universal framework for *computational fields*, SMuC operates at a higher level where devices (i.e. the nodes in the graph) achieve results (the fixpoints) definable in expressive declarative ways, but rely on an underlying framework that can be used to support communications and to check termination of formula evaluations (correct computation of fixpoints). Moreover, the *field-calculus* is mainly functional while SMuC, as already mentioned, is based on a declarative definition of *fields* computed by two kinds of recursions (least and greatest fixpoint). Second, the underlying graph in SMuC is explicitly considered in the operational semantics while it is abstract in the *field-calculus*. Moreover, links are also equipped with network capabilities that can be used to transform communicated values.

Different middleware/platforms have been proposed to support coordination of distributed agents via computational fields [20, 31, 22, 25]. Protelis [25][2] is a language that, inspired by Proto and integrating the Field Calculus features, aims at providing a Java framework for simplifying development of networked systems. In [20] the framework TOTA (*Tuples On The Air*), is introduced to provide spatial abstractions for a novel approach to distributed systems development and management, and is suitable to tackle the complexity of modern distributed computing scenarios, and promotes self-organisation and self-adaptation. In [31] a similar approach has been extended to obtain a chemical-inspired model. This extends tuple spaces with the ability of evolving tuples mimicking chemical systems and provides the machinery enabling agents coordination via spatial computing patterns of competition and gradient-based interaction. In [27] computational fields and ant colony optimisation techniques are combined in a cloud computing scenario. The idea in that approach is to populate the network with mobile agents that explore and build a computational field of pheromones to be exploited when looking for computational resources in the cloud system. The approach is validated using a simulator of cloud systems [28]. In [22] a framework for distributed agent coordination via *eco-laws* has been proposed. This kind of laws generalise the chemical-inspired ones [31] in a framework where self-organisation can be injected in pervasive service ecosystems in terms of spatial structures and algorithms for supporting the design of context-aware applications. The proposed calculus considers computational fields at a more higher level of abstraction with respect to the above mentioned frameworks. However, these frameworks could provide the means for developing a distributed implementation of SMuC.

Finally, we can observe that SMuC, like many of the languages and frameworks referenced above, is remindful of *gossip protocols* [14, 6]. These are a class of communication protocol that, inspired by the form of gossip experienced in social networks, try to solve coordination/computational problems in distributed systems: each node in the network spreads/collets relevant information to/from its neighbours until a global equilibrium is reached. SMuC somehow generalizes some classes of gossip protocols. Functions associated with edges and nodes via labels can be used to control and aggregate the data exchanged among nodes while providing a general framework that can be used to model/program many of the existing protocols. However, many gossip protocols are probabilistic in nature, while SMuC computations are deterministic. Further investigations are definitively needed to assess the exact relation between gossip protocols and SMuC.

[2]http://protelis.github.io/

7. Conclusion

We have presented a simple calculus, named SMuC, that can be used to program and coordinate the activities of distributed agents via computational fields. In SMuC a computation consists of a sequence of fixpoints computed in a fixed graph-shaped field that represents the space topology modelling the underlying network. Our graph-based fields have attributes on both nodes and arcs, where the latters represent interaction capabilities between nodes. Under reasonable conditions, fixpoints can be computed via asynchronous iterations. At each iteration the attributes of some nodes are updated according to the values of neighbors in the previous iteration. The fixpoint computation is robust against certain forms of unavailability and failure situations. SMuC is also equipped with a set of control-flow constructs which allow one to conveniently structure the fixpoint computations. We have also developed a prototype tool for our language, equipped with a graphical interface that provides useful visual feedback. Indeed we employ those visual features to illustrate the application of our approach to a robot rescue case study, for which we provide a novel rescue coordination strategy, programmed in SMuC.

Finally, we have presented a distributed implementation of our calculus. The translation is done in two phases, from SMuC programs into normal form SMuC programs and then into distributed programs. The correctness of translations exploits the above mentioned results on asynchronous computations.

As future work we plan to deploy the implementation specified in the paper on a suitable distributed architecture, and to carry out experiments about case studies of aggregate programming and gossip protocols. Specific domains of applications are the Internet-of-Things and Big (Graph) Data analytics. The former has been subject of focus by seminal works on aggregate programming [4], while the latter seems particularly attractive given the similarities between the model of computation of SMuC and the BSP model of computation [30] on which parallel graph analysis frameworks like Google's Pregel [19] and Apache's Giraph [8] are based on. We will also consider gossip based protocols for aggregate computations in large dynamic and p2p networks (see for instance [15, 18]). Another possible field of application could be distributed and parallel model checking, given that SMuC formulas generalise some well-known temporal logics used in the field of model checking.

Furthermore, we plan to compare the *expressivity aspects* of SMuC with respect to the languages and calculi previously proposed in literature, and to the *field calculus* [10] in particular. This comparison is not only interesting from a theoretical point of view, but could also provide a deeper understanding of possible alternative *paradigms* for *aggregate programming.*

We also plan to study mechanisms that allow dynamic deployment of new SMuC code fragments. From this point of view a source of inspiration could be the works presented in [11] where a higher-order version of the *field calculus* is presented and [24] where overlapping fields are used to adapt to network changes.

Acknowledgement

The authors wish to thank Carlo Pinciroli for interesting discussions in preliminary stages of the work and the anonymous referees from the conference version of this work for their insightful and encouraging comments. This work has been supported by the European

projects IP 257414 ASCENS and STReP 600708 QUANTICOL, and the Italian PRIN 2010LHT4KM CINA.

Appendix A. Proofs

Lemma 3.1 (semiring monotony)**.** Let F be a field, where F_A is a semiring, I_F is such that $I_F(\alpha)(e)$ is monotone for all $\alpha \in L_A, e \in E_A$, $\mathcal{M}$ contains only function symbols that are obtained by composing additive and multiplicative operations of the semiring, ρ be an environment and Ψ be a ρ-closed formula. Then, every function $\lambda \mathsf{f}.[\![\Psi]\!]^F_{\rho[^{\mathsf{f}}/_z]}$ is monotone and continuous.

Proof. The proof is easily obtained from the monotony requirements and the properties of the semiring operators. □

Lemma 3.2 (monotony of pattern-restricted application)**.** Let F be a field, $\psi : (N \to A) \to N \to A$ be a monotone update function, f, $\mathsf{f1}$, $\mathsf{f2} : N \to A$ be node valuations, π, π_1, π_2 be patterns, and $n \in N_F$ be a node. Then, the following holds:

(i) Function ψ_π is monotone;
(ii) $n \in \pi_1 \Leftrightarrow n \in \pi_2$ implies $\psi_{\pi_1} \mathsf{f}\, n = \psi_{\pi_2} \mathsf{f}\, n$;
(iii) $n \in \pi_1 \Leftrightarrow n \in \pi_2$ and $\mathsf{f}_1 \sqsubseteq \mathsf{f}_2$ implies $\psi_{\pi_1}\, \mathsf{f}_1\, n \sqsubseteq \psi_{\pi_2}\, \mathsf{f}_2\, n$.

Proof. We prove the above statements (i-iii) separately.

(i) We have to show that for all node valuations $\mathsf{f}_1, \mathsf{f}_2$ such that $\mathsf{f}_1 \sqsubseteq \mathsf{f}_2$ we have that $\psi_\pi\, \mathsf{f}_1 \sqsubseteq \psi_\pi\, \mathsf{f}_2$, i.e. that for all nodes $n \in N_F$ we have that $\psi_\pi\, \mathsf{f}_1\, n \sqsubseteq \psi_\pi\, \mathsf{f}_2\, n$. We assume $\mathsf{f}_1 \sqsubseteq \mathsf{f}_2$ and we consider two cases for n depending on whether it belongs to π or not. If $n \in \pi$ then, according to Definition 3.12, $\psi_\pi \mathsf{f}_1\, n = \psi\, \mathsf{f}_1\, n$ and $\psi_\pi \mathsf{f}_2\, n = \psi\, \mathsf{f}_2\, n$. Since ψ is monotone it follows that $\psi\, \mathsf{f}_1\, n \sqsubseteq \psi\, \mathsf{f}_2\, n$ and hence $\psi_\pi\, \mathsf{f}_1\, n \sqsubseteq \psi_\pi\, \mathsf{f}_2\, n$. Otherwise, if $n \notin \pi$ then, according to Definition 3.12, $\psi_\pi \mathsf{f}_1\, n = \mathsf{f}_1\, n$ and $\psi_\pi \mathsf{f}_2\, n = \mathsf{f}_2\, n$. Since $\mathsf{f}_1 \sqsubseteq \mathsf{f}_2$ it clearly follows that $\mathsf{f}_1\, n \sqsubseteq \mathsf{f}_2\, n$ and hence $\psi_\pi\, \mathsf{f}_1\, n \sqsubseteq \psi_\pi\, \mathsf{f}_2\, n$.
(ii) This can be easily derived from Definition 3.12. Indeed, if both $n \in \pi_1$ and $n \in \pi_2$ then $\psi_{\pi_1} \mathsf{f}\, n = \psi\, \mathsf{f}\, n$ and $\psi_{\pi_2} \mathsf{f}\, n = \psi\, \mathsf{f}\, n$. Otherwise, if $n \notin \pi_1$ and $n \notin \pi_2$ then $\psi_{\pi_1} \mathsf{f}\, n = \mathsf{f}\, n$ and $\psi_{\pi_2} \mathsf{f}\, n = \mathsf{f}\, n$.
(iii) This follows immediately from (ii) and (iii). Indeed, from (i) we have that $\psi_{\pi_1}\, \mathsf{f}_1\, n \sqsubseteq \psi_{\pi_1}\, \mathsf{f}_2\, n$, and from (ii) we have that $\psi_{\pi_1}\, \mathsf{f}_2\, n = \psi_{\pi_2}\, \mathsf{f}_2\, n$. □

Lemma 3.3 (pattern-restricted application bounds)**.** Let F be a field, $\psi : (N \to A) \to N \to A$ be a monotone update function, σ be a finite strategy and π be a pattern. Then it holds $\psi_\sigma \sqsubseteq \psi_\pi\, \psi_\sigma \sqsubseteq \psi\, \psi_\sigma$

Proof. The proof is by induction on the length of σ.

Case $|\sigma| = 0$**,** i.e. $\sigma = \epsilon$**:** We have to prove that $\psi_\epsilon \sqsubseteq \psi_\pi\, \psi_\epsilon \sqsubseteq \psi\, \psi_\epsilon$. Since $\psi_\epsilon = \bot$ this amounts to proving $\bot \sqsubseteq \psi_\pi \bot \sqsubseteq \psi \bot$. First, we have $\bot \sqsubseteq \psi_\pi \bot$ by definition. And, second, $\psi_\pi \bot \sqsubseteq \psi \bot$ can be shown pointwise on nodes n distiguishing whether n belongs to π or not, as in the above proofs. Indeed, for $n \in \pi$ we have $\psi_\pi \bot n = \psi \bot n$ by definition. If instead, $n \notin \pi$ we have $\psi_\pi \bot n = \bot n = \bot_A$ by definition and, clearly, $\bot_A \sqsubseteq \psi \bot n$.

Case $|\sigma| > 0$**:** Assume as induction hypothesis that for all prefixes $\sigma[1..i]$ of $\sigma[1..|\sigma| - 1]$ the lemma holds, i.e. for all patterns π' we have $\psi_{\sigma[1..i]} \sqsubseteq \psi_{\pi'}\, \psi_{\sigma[1..i]} \sqsubseteq \psi\, \psi_{\sigma[1..i]}$. We prove

that the theorem follows pointwise for every node $n \in N_F$, i.e. that $\forall n \in N_F : \psi_\sigma\, n \sqsubseteq \psi_\pi\, \psi_\sigma\, n \sqsubseteq \psi\, \psi_\sigma\, n$.

We start proving the first part of the inequality in the theorem, i.e. $\psi_\sigma \sqsubseteq \psi_\pi\, \psi_\sigma$. We consider two cases for n depending on whether it belongs to π or not. The easiest case is when $n \notin \pi$. Indeed, in this case we obtain $\psi_\pi \psi_\sigma\, n = \psi_\sigma\, n$ from Definition 3.12. If instead $n \in \pi$ the proof is more elaborated. Let $\sigma[1..j]$ be the longest prefix of σ, if any, where n has been updated. We have $\sigma = \sigma[1..j], \sigma_{j+1}, \sigma[j+1..|\sigma|]$ and $n \notin (\sigma_{j+1} \cup .. \cup \sigma_{|\sigma|})$. We have $\psi_{\sigma[1..j]} \sqsubseteq \psi_\sigma$ by the inductive hypothesis, thus $\psi_{\sigma_{j+1}}\, \psi_{\sigma[1..j]}\, n \sqsubseteq \psi_\pi\, \psi_\sigma\, n$ by Lemma 3.2. But $\psi_{\sigma_{j+1}}\, \psi_{\sigma[1..j]}\, n = \psi_{\sigma[1..j+1]}\, n = .. = \psi_\sigma\, n$ by Definition 3.12 since n does not belong to any pattern in $\sigma[j+1..|\sigma|]$ (otherwise $\sigma[1..j]$ would not be the longest prefix of σ where n has been updated) Therefore, $\psi_\sigma\, n \sqsubseteq \psi_\pi\, \psi_\sigma\, n$.

We now prove the second part of the inequality in the theorem, i.e. $\psi_\pi\, \psi_\sigma\, n \sqsubseteq \psi\, \psi_\sigma\, n$. Again, we consider two cases for n depending on whether it belongs to π or not. If $n \in \pi$ then, from Definition 3.12, we have that $\psi_\pi\, \psi_\sigma\, n = \psi\, \psi_\sigma\, n$. If instead $n \notin \pi$, we have that, according to Definition 3.12, $\psi_\pi\, \psi_\sigma\, n = \psi_\sigma\, n$ and $\psi_\sigma\, n \sqsubseteq \psi\, \psi_\sigma\, n$ by letting $\pi = N_F$ in the previous result. □

Corollary 3.1 (strategy-restricted applications yield chains)**.** *Let F be a field, $\psi : (N \to A) \to N \to A$ be a monotone update function and σ be an infinite strategy. Then, the sequence $\bot \sqsubseteq \psi_{\sigma[1..1]} \sqsubseteq \psi_{\sigma[1..2]} \sqsubseteq \ldots$ is actually a partially ordered chain.*

Proof. The corollary follows immediately from Lemma 3.3. □

Theorem 3.16 (bounds under fair strategies)**.** *Let F be a field, with N_F finite, $\psi : (N \to A) \to N \to A$ be a monotone update function and σ be a fair strategy. Then all elements of the partially ordered chains $\bot \sqsubseteq \psi_{\sigma[1..1]} \sqsubseteq \psi_{\sigma[1..2]} \sqsubseteq \ldots$ and $\bot \sqsubseteq \psi\,\bot \sqsubseteq \psi^2\,\bot \sqsubseteq \ldots$ have the same set of upper bounds and hence the same least upper bound, namely the least fixpoint of ψ.*

Proof. We prove that for any element of a chain there is an element in the other chain which is larger or equal.

We start first proving that this holds for chain $\bot \sqsubseteq \psi_{\sigma[1..1]} \sqsubseteq \psi_{\sigma[1..2]} \sqsubseteq \ldots$ with respect to $\bot \sqsubseteq \psi\,\bot \sqsubseteq \psi^2\,\bot \sqsubseteq \ldots$. In particular, for every chain element $\psi_{\sigma[1..k]}$ there is a chain element $\psi^l\,\bot$ such that $\psi_{\sigma[1..k]} \sqsubseteq \psi^l\,\bot$. Indeed, this holds for $k = l$. We hence prove that $\psi_{\sigma[1..k]} \sqsubseteq \psi^k\,\bot$ for every $k \in \mathbb{N}$. The proof is by induction on k. For $k = 0$ we have, $\psi_\epsilon = \bot = \bot = \psi^k\,\bot$. Assuming $\psi_{\sigma[1..k]} \sqsubseteq \psi^k\,\bot$ as induction hypothesis we can easily prove that $\psi_{\pi_{k+1}}\, \psi_{\sigma[1..k]} = \psi_{\sigma[1..k+1]} \sqsubseteq \psi^{k+1}\,\bot = \psi\,\psi^k\,\bot$. In fact, $\psi_{\pi_{k+1}}\, \psi_{\sigma[1..k]} \sqsubseteq \psi\,\psi_{\sigma[1..k]}$ holds by Lemma 3.3 and $\psi\,\psi_{\sigma[1..k]} \sqsubseteq \psi\,\psi^k\,\bot$ follows from the inductive hypothesis ($\psi_{\sigma[1..k]} \sqsubseteq \psi^k\,\bot$) and the monotony of ψ.

We now prove the other direction, i.e. that for every element $\psi^k\,\bot$ in $\bot \sqsubseteq \psi\,\bot \sqsubseteq \psi^2\,\bot \sqsubseteq \ldots$ there an element $\psi_{\sigma[1..h]}$ in $\bot \sqsubseteq \psi_{\sigma[1..1]} \sqsubseteq \psi_{\sigma[1..2]} \sqsubseteq \ldots$ that is larger or equal (i.e. $\psi^k\,\bot \sqsubseteq \psi_{\sigma[1..h]}$). We prove this by induction on k. For $k = 0$ we can choose h to be 0 as well, so that we trivially have $\psi^0\,\bot = \bot = \psi_\epsilon$. Assume by induction that there is an $h \in \mathbb{N}$, such that $\psi^k\,\bot \sqsubseteq \psi_{\sigma[1..h]}$. We will show that there is an element $\psi_{\sigma[1..h+i]}$ that can bound $\psi^{k+1}\,\bot$. In particular, we choose such an element (determined by i) that satisfies $\forall n.n \in \sigma_{h+1} \cup .. \cup \sigma_{h+i}$. In words, every node will be updated within the next i steps. Such an i does necessarily exist since the strategy σ is fair. We prove $\psi^{k+1}\,\bot \sqsubseteq \psi_{\sigma[1..h+i]}$ pointwise. Let $h + j$ be the last index such that $n \in \sigma_{h+j}$. Thus we have $\psi\,\psi_{\sigma[1..h+j]}\, n = \psi_{\sigma[1..h+j+1]}\, n = .. = \psi_{\sigma[1..h+i]}\, n$.

Finally we conclude:

$$\begin{array}{rcll}
\psi^k \perp & \sqsubseteq & \psi_{\sigma[1..h]} & \text{by the inductive hypothesis} \\
\psi_{\sigma[1..h]} & \sqsubseteq & \psi_{\sigma[1..h+j]} & \text{by Corollary 3.1} \\
\psi^k \perp & \sqsubseteq & \psi_{\sigma[1..h+j]} & \text{by transitivity} \\
\psi^{k+1} \perp n & \sqsubseteq & \psi\, \psi_{\sigma[1..h+j]}\, n & \text{by monotonicity and pointwise ordering} \\
\psi^{k+1} \perp n & \sqsubseteq & \psi_{\sigma[1..h+i]}\, n & \text{by the above equality}
\end{array}$$

□

Theorem 3.17 (robustness against unavailability)**.** *Let F be a field with finite set of nodes N_F and field domain A with finite partially ordered chains only, σ be a fair strategy and $\psi : (N \to A) \to N \to A$ be a monotone update function. Then the partially ordered chain $\perp \sqsubseteq \psi_{\sigma[1..1]} \sqsubseteq \psi_{\sigma[1..2]} \sqsubseteq \dots$*

(i) *stabilizes to its least upperbound* f*;*

(ii) *its least upper bound* f *does not depend on the fair strategy: we always have $f = \mathit{lfp}\, \psi$.*

Proof. It is immediate to see that domain $N \to A$ has only finite chains. Thus given any fair strategy σ, the chain $\perp \sqsubseteq \psi_{\sigma[1..1]} \sqsubseteq \psi_{\sigma[1..2]} \sqsubseteq \dots$ reaches in, say, k' steps its least upper bound (which proves (i)). Property (ii) immediately follows from Theorem 3.16. □

Theorem 3.19 (least upper bound of a safe failure sequence)**.** *Let F be a field, with N_F finite, $\psi : (N \to A) \to N \to A$ be a monotone update function, $\tilde{\sigma}$ be a fair strategy and $\overline{\varsigma}_{\tilde{\sigma}}$ be some infinite safe $\tilde{\sigma}$ failure sequence of ψ. Then $\overline{\varsigma}_{\tilde{\sigma}}$ has a least upper bound which is the least fix point of ψ.*

Proof. We prove that: (i) for any element of the failure sequence $\overline{\varsigma}_{\tilde{\sigma}}$ there is an element in the chain $\overline{\psi}_{\tilde{\sigma}}$ which is larger or equal of it; and (ii) viceversa. Thus both the failure sequence and the chain have the same set of upper bounds: but the chain is guaranteed to have a least upper bound, which, by Theorem 3.17, is the minimal fix point of ψ.

For (i), given ς_σ, with σ finite, we choose just ψ_σ. We have to prove $\varsigma_\sigma \sqsubseteq \psi_\sigma$ for all finite σ. First, we have $\varsigma_\epsilon = \perp_{N \to A} = \psi_\epsilon$. Also, assuming $\varsigma_\sigma \sqsubseteq \psi_\sigma$ we have $\psi_\pi \varsigma_\sigma \sqsubseteq \psi_\pi \psi_\sigma$. But $\forall n. \varsigma_{\sigma,\pi} n \sqsubseteq \psi_\pi \varsigma_\sigma n$, and thus $\varsigma_{\sigma,\pi} \sqsubseteq \psi_{\sigma,\pi}$, as required. In fact, if $n \in \pi$, or $n \notin \pi$ and in equation 3.1 the third option is taken, then ψ_π has the same effect as the recursive call in equation 3.1, while if the fourth option is taken we have $\varsigma_{\sigma,\pi} n = \varsigma_{\sigma'} n \sqsubseteq \psi_{\sigma'} n \sqsubseteq \psi_\sigma n$, where the former inclusion holds for the inductive hypothesis, while the latter inclusion holds since $\overline{\psi}_{\tilde{\sigma}}$ is a chain.

For (ii), given a $\tilde{\sigma}$ safe failure sequence $\overline{\varsigma}_{\tilde{\sigma}}$, let $\tilde{\sigma} = \hat{\sigma}, \tilde{\sigma}'$, where $\hat{\sigma}$, finite, corresponds to the only part of $\overline{\varsigma}_{\tilde{\sigma}}$ where the fourth option of equation 3.1 has been employed. Such $\hat{\sigma}$ do exists since $\overline{\sigma}$ is safe. Thus $\overline{\varsigma}_{\tilde{\sigma}'}$ and $\overline{\psi}_{\tilde{\sigma}'}$ coincide, while for every $\psi_{\sigma''}$ in $\overline{\psi}_{\tilde{\sigma}'}$ we have $\psi_{\sigma''} \sqsubseteq \varsigma_{\hat{\sigma},\sigma''}$ for all finite σ''. In fact, $\psi_\epsilon = \perp_{N \to A} \sqsubseteq \varsigma_{\hat{\sigma}}$ and then the same update functions have been applied on both sides according to σ''. Furthermore, chains $\overline{\psi}_{\tilde{\sigma}'}$ and $\overline{\psi}_{\tilde{\sigma}}$, according to Theorem 3.16 have the same least upper bound, the fix point of ψ. Thus given any element ψ_σ in $\overline{\psi}_{\tilde{\sigma}}$ there exists a $\psi_{\sigma'}$ in $\overline{\psi}_{\tilde{\sigma}'}$ with $\psi_\sigma \sqsubseteq \psi_{\sigma'}$. But in $\overline{\varsigma}_{\tilde{\sigma}}$ we have an even larger element: $\psi_{\sigma'} \sqsubseteq \varsigma_{\hat{\sigma},\sigma'}$. Thus we conclude $\psi_\sigma \sqsubseteq \varsigma_{\hat{\sigma},\sigma'}$. □

Lemma 5.1. Let F be a field with field domain A with finite chains only, Ψ a formula, $c \in \mathbb{N}$, and label i. Let $\mathcal{A}(\Psi)^i_c = [S, c']$, then:

$$[\![\Psi]\!]^F_\emptyset = f \Leftrightarrow \langle S, F \rangle \to^* \langle \mathsf{skip}, F' \rangle \text{ and } I_{F'}(i) = f$$

Proof. By induction on the syntax of Ψ.

Base of Induction: If $\Psi = j$ then statement follows directly from the fact that $[\![j]\!]_\rho^F = I_F(j)$, $\mathcal{A}_c^i = i \leftarrow j$ and from rule (μSTEP) in Table 1.

Inductive Hypothesis: Let $\Psi_1, \ldots, \Psi_k$ be such that, for any j:

$$[\![\Psi_j]\!]_\emptyset^F = f \Leftrightarrow \langle S_{\Psi_j}^{i_j}, F\rangle \rightarrow^* \langle \mathsf{skip}, F'\rangle \text{ and } I_{F'}(i_j) = f$$

Inductive Hypothesis: Many cases are standard and we provide the details here only for the cases $\Psi = f(\Psi_1, \ldots, \Psi_k)$ and $\Psi = \mu z.\Psi_i$ while we omit $\Psi = g \textcircled{\alpha} \Psi_i$, $\Psi = g \textcircled{\alpha} \Psi_i$ and $\Psi = \nu z.\Psi_i$ that are similar to the ones considered in the proof. In this case we have that $[\![f(\Psi_1, \ldots, \Psi_j)]\!]_\rho^F = f([\![\Psi_1]\!]_\rho^F, \ldots [\![\Psi_n]\!]_\rho^F)$. Moreover:

$$S_\Psi^i = S_{\Psi_1}^{x_{c_1}} \;; \cdots ; \; S_{\Psi_k}^{x_{c_k}} \;; \; i \leftarrow f(x_{c_1}, \ldots, x_{c_k})$$

for the appropriate $x_{c_1}, \ldots, x_{c_n}$. By *Inductive Hypothesis* we have that for each k:

$$[\![\Psi_k]\!]_\emptyset^{F_k} = f \Leftrightarrow \langle S_{\Psi_k}^{x_k}, F_k\rangle \rightarrow^* \langle \mathsf{skip}, F_{k+1}\rangle \text{ and } I_{F_{k+1}}(x_k) = f$$

where $F_1 = F$. Moreover, since each $S_{\Psi_k}^{x_{c_k}}$ uses different auxiliary labels, we also have that:

- $\langle S_{\Psi_1}^{x_{c_1}} \;; \cdots ; \; S_{\Psi_n}^{x_{c_n}} \;; \; i \leftarrow f(x_{c_1}, \ldots, x_{c_n}), F\rangle \rightarrow^* \langle j \leftarrow f(x_{c_1}, \ldots, x_{c_n}), F_{n+1}\rangle$;
- for any k, $I_{F_{n+1}}(x_k) = I_{F_{k+1}}(x_k)$ and $[\![\Psi_k]\!]_\rho^F = [\![\Psi_k]\!]_\rho^{F_k} = [\![\Psi_k]\!]_\rho^{F_{n+1}})$.

Hence, we have that $\langle i \leftarrow f(x_{c_1}, \ldots, x_{c_n}), F_{n+1}\rangle \rightarrow \langle \mathsf{skip}, F'\rangle$, where:

$$\begin{array}{rcl} I_{F'}(i) & = & \lambda n.[\![f]\!]_{A_F}(I_{F_k+1}(x_{c_1})n, \ldots, I_{F_k+1}(x_{c_k})n) \\ & = & \lambda n.[\![f]\!]_{A_F}([\![\Psi_1]\!]_\rho^F\, n, .., [\![\Psi_k]\!]_\rho^F\, n) \\ & = & [\![\Psi]\!]_\rho^F \end{array}$$

Notice that similar considerations apply when $\Psi = g \textcircled{\alpha} \Psi_i$, $\Psi = g \textcircled{\alpha} \Psi_i$.

Let us now consider the case $\Psi = \mu\kappa.\Psi_i$ (Similarly considerations can be used when $\Psi = \nu\kappa.\Psi_i$). We can first of all notice that since A_F has only only finite partially ordered chains, we have that there exists a value k such that $[\![\mu z.\Psi]\!]_\emptyset^F = [\![\tilde{\Psi}_k]\!]_\emptyset^F$ where $\tilde{\Psi}_0 = \bot$ while $\tilde{\Psi}_{k+1} = \Psi[{}^{\tilde{\Psi}_k}/_z]$. Moreover, for any $k' \geq k$ we also have that: $[\![\tilde{\Psi}_{k'}]\!]_\emptyset^F = [\![\tilde{\Psi}_{k'+1}]\!]_\emptyset^F$. We can also notice that S_Ψ has the form:

```
x_c ← ⊥ ;
x_{c+1} ← ⊥ ;
x_{c'} ← ⊥;
while x_{c'} do
    x_c ← x_{c+1};
    S_{Ψ[x_c/z]} ;
    x_{c'} ← x_c = x_{c+1}
i ← x_{c+1}
```

Let F_k be the field obtained after k iterations, of the program above. By using inductive hypothesis we have that $I_{F_k}(x_{c+1}) = [\![\tilde{\Psi}_k]\!]_\emptyset^F$ while, if $k > 1$, $I_{F_k}(x_c) = [\![\tilde{\Psi}_{k-1}]\!]_\emptyset^F$. The program above terminates only when x_c and x_{c+1} are equal, i.e. when $[\![\tilde{\Psi}_k]\!]_\emptyset^F = [\![\tilde{\Psi}_{k+1}]\!]_\emptyset^F = [\![\mu z.\Psi]\!]_\emptyset^F$, and i is assigned to x_{c+1}. □

Lemma 5.2. Let F be a field with field domain A with finite chains only, for any P the following holds:

- if $\langle P, F\rangle \to \langle P', F'\rangle$ then $\langle S_P, F\rangle \to^* \langle S_{P'}, F''\rangle$ and $F' = F''\backslash\mathcal{X}$;
- if $\langle S_P, F\rangle \to \langle S', F'\rangle$ then there exist P' and F'' such that $\langle S', F'\rangle \to^* \langle S_{P'}, F''\rangle$ and $\langle P, F\rangle \to \langle P', F''\backslash\mathcal{X}\rangle$.

Above, we use $F'\backslash\mathcal{X}$ to refer to the field obtained from F' by erasing all the labels in $\mathcal{X}$.

Proof. We prove this lemma by induction on the syntax of P.

Base of Induction: When $P = \mathsf{skip}$ the thesis follows directly from the fact that $S_P = P$, while when $P = i \leftarrow \Psi$ we can directly use Lemma 5.1.

Inductive Hypothesis: For any P_1 and P_2:

- if $\langle P_i, F\rangle \to \langle P', F'\rangle$ then $\langle S_{P_1}, F\rangle \to^* \langle S_{P'}, F''\rangle$ and $F' = F''\backslash\mathcal{X}$;
- if $\langle S_{P_1}, F\rangle \to \langle S', F'\rangle$ then exist P' and F'' such that $\langle S', F'\rangle \to^* \langle S_{P'}, F''\rangle$ and $\langle P, F\rangle \to \langle P', F''\backslash\mathcal{X}\rangle$.

Inductive Step: We have to consider the following cases:

- $P = P_1; P_2$. If $P_1 = \mathsf{skip}$ we have that $\mathsf{skip}; P_2$ has exactly the same computations of P_2. Moreover, $S_P = \mathsf{skip}; \mathsf{wait}(x_c); S_{P_2}$ and we can directly derive our thesis from the inductive hypothesis and from the fact that: $\langle \mathsf{skip}; \mathsf{wait}(x_c); S_{P_2}, F\rangle \to \langle \mathsf{skip}; S_{P_2}, F\rangle$.

  Let $P_1 \neq \mathsf{skip}$. If $\langle P_1; P_2, F\rangle \to \langle P', F'\rangle$ then $P' = P_1'; P_2$ (for some P_1') and $\langle P_1, F\rangle \to \langle P_1', F'\rangle$. By inductive hypothesis we have that $\langle S_{P_1}, F\rangle \to^* \langle S_{P_1'}, F''\rangle$ and $F' = F''\backslash\mathcal{X}$. However, $S_P = S_{P_1}; \mathsf{wait}(x_c); S_{P_2}$, $\langle S_{P_1}; \mathsf{wait}(x_c); S_{P_2}, F\rangle \to^* \langle S_{P_1'}; \mathsf{wait}(x_c); S_{P_2}, F''\rangle$ and $S_{P'} = S_{P_1'}; \mathsf{wait}(x_c); S_{P_2}$.

  Similarly, if $\langle S_{P_1}; \mathsf{wait}(x_c); S_{P_2}, F\rangle \to \langle S', F'\rangle$ then there exists S_1' such that $\langle S_{P_1}, F\rangle \to \langle S_1', F'\rangle$. By inductive hypothesis, there exist P_1' and F'' such that $\langle S_1', F'\rangle \to^* \langle S_{P_1'}, F''\rangle$ and $\langle P_1, F\rangle \to \langle P_1', F''\backslash\mathcal{X}\rangle$. However, by using rule (Seq1) in Table 1, we have that $\langle S_1'; \mathsf{wait}(x_c); S_{P_2}, F'\rangle \to^* \langle S_{P_1'}; \mathsf{wait}(x_c); S_{P_2}, F''\rangle$ and $\langle P_1; P_2, F\rangle \to \langle P_1'; P_2, F''\backslash\mathcal{X}\rangle$.
- $P = \mathsf{if}\ \Psi\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$. We have that $S_P = S_\Psi^{x_c}; \mathsf{if}\ x_c\ \mathsf{then}\ S_{P_1}\ \mathsf{else}\ S_{P_2}$. If $\langle P, F\rangle \to \langle P', F'\rangle$ then $F = F'$ and either $[\![\Psi]\!]_\emptyset^F = \lambda n.true$ and $P' = P_2$ or $[\![\Psi]\!]_\emptyset^F \neq \lambda n.true$ and $P' = P_2$. We can consider only the first case, since the other one follows with similar considerations. By using Lemma 5.1, we have that if $[\![\Psi]\!]_\emptyset^F = \lambda n.true$ then $\langle S_\Psi^{x_c}, F\rangle \to^* \langle \mathsf{skip}, F'\rangle$ and $I_{F'}(x_c) = \lambda n.true$. Hence:

  $$\langle S_P, F\rangle \to^* \langle \mathsf{skip}; \mathsf{if}\ x_c\ \mathsf{then}\ S_{P_1}\ \mathsf{else}\ S_{P_2}, F''\rangle \to \langle S_{P_1}, F''\rangle$$

  where $F = F''\backslash\mathcal{X}$ since in $S_\Psi^{x_c}$ only auxiliary labels can be assigned.

  If $\langle S_P, F\rangle \to \langle S', F'\rangle$ then $S' = S''; \mathsf{if}\ x_c\ \mathsf{then}\ S_{P_1}\ \mathsf{else}\ S_{P_2}$ and $\langle S_\Psi^{x_c}, F\rangle \to \langle S'', F'\rangle$. Moreover, thanks to Lemma 5.1, we have that $\langle S'', F'\rangle \to^* \langle \mathsf{skip}, F''\rangle$ and $I_{F''}(x_c) = [\![\Psi]\!]_\emptyset^F$. This implies that either $\langle S', F'\rangle \to \langle S_{P_1}, F''\rangle$ (when $I_{F''}(x_c) = [\![\Psi]\!]_\emptyset^F = \lambda n.true$) or $\langle S', F'\rangle \to \langle S_{P_2}, F''\rangle$ (when $I_{F''}(x_c) = [\![\Psi]\!]_\emptyset^F \neq \lambda n.true$). In the first case $\langle P, F\rangle \to \langle P_1, F\rangle$ while in the second case $\langle P, F\rangle \to \langle P_2, F\rangle$. In both the cases $F = F''\backslash\mathcal{X}$.
- $P = \mathsf{while}\ \Psi\ \mathsf{do}\ P_1$. We have that

  $$S_P = R_\Psi^{x_c}; \mathsf{while}\ x_c\ \mathsf{do}\ S_{P_1}; \mathsf{wait}(x_{c''}); R_\Psi^{x_c}$$

  If $\langle P, F\rangle \to \langle P', F'\rangle$ then $F = F'$ and either $[\![\Psi]\!]_\emptyset^F = \lambda n.true$ and $P' = \mathsf{skip}$, or $[\![\Psi]\!]_\emptyset^F \neq \lambda n.true$ and $P' = P_1; \mathsf{while}\ \Psi\ \mathsf{do}\ P_1$. In both the cases the statement follows like in the previous item by using Lemma 5.1. □

Lemma 5.3. For any S_1 and S_2, and for any D_{S_1} that flows into D_{skip}, the following hold:

- if $D_{\mathsf{skip}}; S_2$ flows into D' then also $D_{S_1}; S_2$ flows into D';
- $D_{S_1}; \mathsf{wait}(x); S_2$ flows into $D_{\mathsf{skip}}; \mathsf{wait}(x); S_2$.

Proof. Both the cases follow directly from Def. 5.9 and from the rules in Tab. 3,Tab. 5 and Tab. 6 □

Lemma 5.4. Let F be a field with field domain A with finite chains only, Ψ a formula, and S^x_Ψ the SAF SMuC program that evaluates Ψ, then any distributed execution $D_{S^x_\Psi}$ that agrees with F, $D_{S^x_\Psi}$ flows in a distributed execution D_{skip} such that $[\![\Psi]\!]^F_\emptyset = D_{\mathsf{skip}} \upharpoonright_F (x)$.

Proof. We proceed by induction on the syntax of Ψ.

Base of Induction: If $\Psi = j$ then statement follows directly from the fact that $[\![j]\!]^F_\rho = I_F(j)$ and from the fact that $S^x_\Psi = x \leftarrow j$ via the appropriate application of rule (D-Step) in Table 3 and of rules in Table 6.

Inductive Hypothesis: Let $\Psi_1, \ldots, \Psi_k$ be such that for any distributed execution $D_{S^{x_i}_{\Psi_i}}$ that agrees with F, it flows in a distributed execution D_{skip} such that $[\![\Psi_k]\!]^F_\emptyset = D_{\mathsf{skip}} \upharpoonright_F (x)$.

Inductive Step:

- $\Psi = f(\Psi_1, \ldots, \Psi_k)$. In this case we have that $[\![f(\Psi_1, \ldots, \Psi_k)]\!]^F_\rho = f([\![\Psi_1]\!]^F_\rho, \ldots [\![\Psi_k]\!]^F_\rho)$. Moreover:
$$S^i_\Psi = S^{x_{c_1}}_{\Psi_1} \ ; \cdots ; \ S^{x_{c_k}}_{\Psi_k} \ ; i \leftarrow f(x_{c_1}, \ldots, x_{c_k})$$
for the appropriate $x_{c_1}, \ldots, x_{c_k}$. By *Inductive Hypothesis* we have there exists $F_1, \ldots, F_{k+1}$, such that for each j, $D_{S^{x_j}_{\Psi_j}}$ agrees with F_j and it flows in a distributed execution D^j_{skip} such that
$$[\![\Psi_j]\!]^{F_j}_\emptyset = D^j_{\mathsf{skip}} \upharpoonright_{F_j} (x_j)$$
where $F_1 = F$ while $F_{j+1} = F_j[^{D^j_{\mathsf{skip}}\upharpoonright_{F_j}}/_{I_{F_j}}]$. Let $S = S^{x_{c_1}}_{\Psi_1} \ ; \cdots ; \ S^{x_{c_k}}_{\Psi_k}$, by Lemma 5.3, we have that if D_S agrees with F then D_S flows in D_{skip} and for any j, $D_{\mathsf{skip}} \upharpoonright_{F_k+1} (x_j) = [\![\Psi_j]\!]^{F_j}_\emptyset$. However, Ψ_j refers only to labels in F. Hence, $[\![\Psi_j]\!]^{F_j}_\emptyset = [\![\Psi_j]\!]^F_\emptyset$. Moreover, it is easy to see that via the appropriate application of rule of rule (D-Step) in Table 3 and of rules in Table 6, $D_{\mathsf{skip}}; i \leftarrow f(x_{c_1}, \ldots, x_{c_k})$ flows in D'_{skip} and for any $n \in N_F$, $D'_{\mathsf{skip}} \upharpoonright_F (x) = [\![\Psi]\!]^F_\emptyset$.
- $\Psi = g \textcircled{\alpha} \Psi_i$ or $\Psi = g \textcircled{\alpha} \Psi_i$, in both the cases we can proceed as in the previous case. Indeed, the SAF program that evaluates Ψ is $S^x_\Psi = S^{x_i}_{\Psi_i}; x \leftarrow g \textcircled{\alpha} x_i$ or $S^x_\Psi = S^{x_i}_{\Psi_i}; x \leftarrow g \textcircled{\alpha} x_i$. In both the cases we have that and via notice that $D_{S^x_\Psi}$ flows to D_{skip} while $D_{\mathsf{skip}}; x \leftarrow g \textcircled{\alpha} x_i$ (resp. $D_{\mathsf{skip}}; x \leftarrow g \textcircled{\alpha} x_i$) flows to the appropriate D'_{skip} such that $D'_{\mathsf{skip}} \upharpoonright_F (x) = [\![\Psi]\!]^F_\emptyset$.

- $\Psi = \mu\kappa.\Psi_i$ or $\Psi = \mu\kappa.\Psi_i$. We consider here $\Psi = \mu\kappa.\Psi$ since the proof for the other case is similar. First of all we can observe that S_Ψ has the form:

$$\begin{array}{l} x_c \leftarrow \bot \ ; \\ x_{c+1} \leftarrow \bot \ ; \\ x_{c'} \leftarrow \bot; \\ \mathsf{while}\ x_{c'}\ \mathsf{do} \\ \quad x_c \leftarrow x_{c+1}; \\ \quad S_{\Psi[^{x_c}/_z]} \ ; \\ \quad x_{c'} \leftarrow x_c = x_{c+1} \\ x \leftarrow x_{c+1} \end{array}$$

Let us consider the sequence $D_0,\ldots,D_i,\ldots$ of distributed programs starting from D_{S_Ψ}, where $D_0 = D_{S_\Psi}$ and for any i, $D_i \Rightarrow_{\langle F,T\rangle} D_{i+1}$. Let Ψ_i be the interpretation x_{c+1} at D_i ($\Psi_i = D_i \upharpoonright_F (x_{c+1})$), we can consider the sequence of elements Ψ_j such that $\Psi_j \neq \Psi_{j-1}$. We can note that this sequence $\Psi_0, \Psi_{j_1}, \ldots$ defines a monotonic sequence that is the result of the application of a *fair strategy* (the one induced by the application of the distributed operational semantics). Since A_F has only only finite partially ordered chains we are under the hypotheses of Theorem 3.17. Hence a fixpoint is eventually reached. Hence, there exists an index k such that: $D_k \upharpoonright_F (x_c) = D_i \upharpoonright_F (x_{c+1}) = [\![\mu\kappa.\Psi_i]\!]^F_\emptyset$. Moreover, $D_k \upharpoonright_F (x_{c'}) = \lambda n.true$. This implies that, after a number of reductions, distributed execution D_{skip} is reached where $D_{\mathsf{skip}} \upharpoonright_F (x) = [\![\mu\kappa.\Psi_i]\!]^F_\emptyset$. □

Theorem 5.11. *Let F be a field with field domain A with finite chains only and T be a spanning tree of F, for any* SMuC *program P, for any D_{S_P} that agrees with F:*

(i) *if $\langle P, F\rangle \to \langle P', F'\rangle$ then $D_{S_P} \Rightarrow_{\langle F,T\rangle} D_{S_{P'}}$ and $D_{S_{P'}}$ agrees with F'.*
(ii) *For any D' such that $D_{S_P} \Rightarrow^*_{\langle F,T\rangle} D'$ there exists P' such that $D' \Rightarrow^*_{\langle F,T\rangle} D_{S_{P'}}$, $\langle P, F\rangle \to \langle P', F'\rangle$ and $D_{S_{P'}}$ agrees with F'.*

Proof.

(i) We proceed by induction on the syntax of P.

Base of Induction: Let $P = \mathsf{skip}$ or $P = i \leftarrow \Psi$;. In the first case the thesis is trivially verified while in the second case our statement follows directly from Lemma 5.4 and from Lemma 5.3.

Inductive Hypothesis: Let P_1, P_2, D_{P_1} and D_{P_2} be such that $D_{S_{P_i}}$ that agrees with F, if $\langle P_i, F\rangle \to \langle P_i', F'\rangle$ then $D_{S_{P_i}} \Rightarrow_{\langle F,T\rangle} D_{S_{P_i'}}$ and $D_{S_{P_i'}}$ agrees with F'.

Inductive Step: The following cases can be considered:

- $P = P_1; P_2$. We have that $S_P = S_{P_1}; \mathsf{wait}(x); S_{P_2}$. We have that $\langle P, F\rangle \to \langle P', F'\rangle$ if and only if either $P_1 = \mathsf{skip}$ and $\langle P_2, F\rangle \to \langle P', F'\rangle$ or $\langle P_1, F\rangle \to \langle P_1', F'\rangle$ and $P' = P_1'; P_2$. In both the cases the statement follows by inductive hypothesis and by Lemma 5.3.
- $P = \mathsf{if}\ \Psi\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$. We have that $S_P = R^x_\Psi; \mathsf{if}\ x\ \mathsf{then}\ S_{P_1}\ \mathsf{else}\ S_{P_2}$. Moreover, if $\langle P, F\rangle \to \langle P', F'\rangle$ then $F = F'$ and either $[\![\Psi]\!]^F_\emptyset = \lambda n.true$, $P' = P_1$ or $[\![\Psi]\!]^F_\emptyset = \lambda n.true$, $P' = P_2$. Let us assume that $[\![\Psi]\!]^F_\emptyset = \lambda n.true$. From Lemma 5.4 we have that $D_{R^x_\Psi} \Rightarrow^*_{\langle F,T\rangle} D_{\mathsf{skip}}$ and by applying rules in Table 3, Table 5 and Table 6 we have that

$$D_S \Rightarrow^*_{\langle F,T\rangle} D_{\mathsf{skip}}; \mathsf{if}\ x\ \mathsf{then}\ S_{P_1}\ \mathsf{else}\ S_{P_2}$$

Moreover, $D_{\mathsf{skip}} \restriction_F (x) = \lambda n.true$ (resp.$D_{\mathsf{skip}} \restriction_F (x) \neq \lambda n.true$), hence

$$D_{\mathsf{skip}}; \mathsf{if}\ x\ \mathsf{then}\ S_{P_1}\ \mathsf{else}\ S_{P_2} \Rightarrow^*_{\langle F,T \rangle} D_{S_{P_1}}$$

and $D_{S_{P_1}}$ agrees with F'.

- $P = \mathsf{while}\ \Psi\ \mathsf{do}\ P_1$. This case follows similarly to previous one.

(ii) We proceed by induction on the syntax of P.

Base of Induction: If $P = \mathsf{skip}$ or $P = i \leftarrow \Psi$, like for the previous point, the statement follows directly from Lemma 5.4 and from Lemma 5.3.

Inductive Hypothesis: Let P_1, P_2, D_{P_1} and D_{P_2} be such that if $D_{S_{P_i}} \Rightarrow^*_{\langle F,T \rangle} D'$ there exists P' such that $D' \Rightarrow^*_{\langle F,T \rangle} D_{S_{P'}}$, $\langle P_i, F \rangle \rightarrow \langle P', F' \rangle$ and $D_{S_{P'}}$ agrees with F'.

Inductive Step: The following cases can be considered:

- $P = P_1; P_2$. We have that $S_P = S_{P_1}; \mathsf{wait}(x); S_{P_2}$. Let $D_{S_P} \Rightarrow^*_{\langle F,T \rangle} D'$. By Lemma 5.3 we have that D_S flows into $D_{\mathsf{wait}(x);S_{P_2}}$. For this reason we can distinguish three cases: (1) there exists D'' such that $D_{S_{P_1}} \Rightarrow^*_{\langle F,T \rangle} D''$ and $D' = D''; \mathsf{wait}(x); S_{P_2}$; (2) $D' \Rightarrow^*_{\langle F,T \rangle} D_{\mathsf{wait}(x);S_{P_2}}$; (3) $D' \Rightarrow^*_{\langle F,T \rangle} D_{\mathsf{wait}(x);S_{P_2}}$. In all the cases the statement follows directly from the inductive hypothesis.
- $P = \mathsf{if}\ \Psi\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$. We have that $S_P = R^x_\Psi; \mathsf{if}\ x\ \mathsf{then}\ S_{P_1}\ \mathsf{else}\ S_{P_2}$. By Lemma 5.3 we have that we can distinguish three cases: (1) there exists D'' such that $D_{R^x_\Psi} \Rightarrow^*_{\langle F,T \rangle} D''$ and $D' = D''; \mathsf{if}\ x\ \mathsf{then}\ S_{P_1}\ \mathsf{else}\ S_{P_2}$; (2) $[\![\Psi]\!]^F_\emptyset = \lambda n.true$ and $D_{S_{P_1}} \Rightarrow^*_{\langle F,T \rangle} D'$; (3) $[\![\Psi]\!]^F_\emptyset \neq \lambda n.true$ and $D_{S_{P_2}} \Rightarrow^*_{\langle F,T \rangle} D'$. In all the cases the statement follows directly from the inductive hypothesis and by simple applications of rules in Table 3, Table 5 and Table 6.
- $P = \mathsf{while}\ \Psi\ \mathsf{do}\ P_1$. This case follows similarly to previous one by noticing that $S_P = R^{x_c}_\Psi;\mathsf{while}\ x_c\ \mathsf{do}\ S;\mathsf{wait}(x_{c''});R^{x_c}_\Psi$ and that after k iterations D_{S_P} flows into $D^k_{\mathsf{wait}(x_{c''});S_P}$. □

Appendix B. Symbols

Symbol	Meaning
A	field domain carrier
a	element of A
B	subset of A
$\sqsubseteq$	field domain ordering relation
$\top$	field domain top element
$\bot$	field domain bottom element
N	set of field nodes
$\mathcal{N}$	universe of nodes
$n, n', \ldots$	node
E	set of field edges
$\mathcal{E}$	universe of edges
$e, e', \ldots$	edge
L	set of field labels
$\mathcal{X}$	set of auxiliary field labels
$\mathcal{L}$	set of all labels
$i, j, x, \ldots$	node labels
α	edge label
I	interpretation of field labels
F	field
$\mathcal{F}$	set of all fields
f	combination operations
g	aggregation operations
$\mathcal{M}$	set of all function symbols
f	node valuation
z	recursion variable
$\mathcal{Z}$	set of all recursion variables
ρ	recursion variable envionment
Ψ	SMuC formula
ψ	update function
μ	least fixpoint operator
ν	greatest fixpoint operator
$\bigcirc$	modal operator (out)
$\bullet$	modal operator (in)
π	execution pattern
σ	execution strategy
ς	failure sequence
P	SMuC program
S, R	SMuC program in SAF
c	counter
χ	agreement store
κ	agreement counter
ι	interpretation of node labels
d	fragment
λ	transition label
τ	silent action
m	message
D	distributed execution
X	set of nodes

Table 7: Symbol notation

References

[1] Jacob Beal and Jonathan Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21:10–19, 2006.

[2] Jacob Beal, Oliver Michel, and Ulrik Pagh Schultz. Spatial computing: Distributed systems that take advantage of our geometric world. *ACM Transactions on Autonomous and Adaptive Systems*, 6:11:1–11:3, 2011.

[3] Jacob Beal, Stefan Dulman, Kyle Usbeck, Mirko Viroli, and Nikolaus Correll. Organizing the aggregate: Languages for spatial computing. *CoRR*, abs/1202.5509, 2012.

[4] Jacob Beal, Danilo Pianini, and Mirko Viroli. Aggregate programming for the internet of things. *IEEE Computer*, 48(9):22–30, 2015.

[5] Dimitri P. Bertsekas. Distributed asynchronous computation of fixed points. *Math. Program.*, 27(1):107–120, 1983.

[6] Ken Birman. The promise, and limitations, of gossip protocols. *SIGOPS Oper. Syst. Rev.*, 41(5):8–13, October 2007.

[7] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Semiring-based constraint satisfaction and optimization. *J. ACM*, 44(2):201–236, 1997.

[8] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *PVLDB*, 8(12):1804–1815, 2015.

[9] Ferruccio Damiani and Mirko Viroli. Type-based self-stabilisation for computational fields. *Logical Methods in Computer Science*, 11(4), 2015.

[10] Ferruccio Damiani, Mirko Viroli, and Jacob Beal. A type-sound calculus of computational fields. *Science of Computer Programming*, 117:17 – 44, 2016.

[11] Ferruccio Damiani, Mirko Viroli, Danilo Pianini, and Jacob Beal. Code mobility meets self-organisation: A higher-order calculus of computational fields. In Susanne Graf and Mahesh Viswanathan, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 35th IFIP WG 6.1 International Conference, FORTE 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, volume 9039 of *Lecture Notes in Computer Science*, pages 113–128. Springer, 2015.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: a flexible data processing tool. *Commun. ACM*, 53(1):72–77, 2010.

[13] Edsger W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[14] Márk Jelasity. *Gossip*, pages 139–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[15] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, August 2005.

[16] Alberto Lluch-Lafuente, Michele Loreti, and Ugo Montanari. A fixpoint-based calculus for graph-shaped computational fields. In Tom Holvoet and Mirko Viroli, editors, *Coordination Models and Languages - 17th IFIP WG 6.1 International Conference, COORDINATION 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings*, volume 9037 of *Lecture Notes in Computer Science*, pages 101–116. Springer, 2015.

[17] Alberto Lluch-Lafuente and Ugo Montanari. Quantitative mu-calculus and CTL defined over constraint semirings. *Theor. Comput. Sci.*, 346(1):135–160, 2005.

[18] Rafik Makhloufi, Grégory Bonnet, Guillaume Doyen, and Dominique Gaïti. *Decentralized Aggregation Protocols in Peer-to-Peer Networks: A Survey*, pages 111–116. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[19] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 135–146. ACM, 2010.

[20] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The TOTA approach. *ACM Transactions on Software Engingeering and Methodology*, 18:15:1–15:56, 2009.

[21] Marco Mamei and Franco Zambonelli. Field-based coordination for pervasive computing applications. In Pietro Liò, Eiko Yoneki, Jon Crowcroft, and Dinesh C. Verma, editors, *BIOWIRE*, volume 5151 of *Lecture Notes in Computer Science*, pages 376–386. Springer, 2007.

[22] Sara Montagna, Mirko Viroli, Jose Luis Fernandez-Marquez, Giovanna Di Marzo Serugendo, and Franco Zambonelli. Injecting self-organisation into pervasive service ecosystems. *MONET*, 18(3):398–412, 2013.

[23] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Undergraduate Topics in Computer Science. Springer, 2007.

[24] Danilo Pianini, Jacob Beal, and Mirko Viroli. Improving gossip dynamics through overlapping replicates. In Alberto Lluch-Lafuente and José Proença, editors, *Coordination Models and Languages - 18th IFIP WG 6.1 International Conference, COORDINATION 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9686 of *Lecture Notes in Computer Science*, pages 192–207. Springer, 2016.

[25] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In Roger L. Wainwright, Juan Manuel Corchado, Alessio Bechini, and Jiman Hong, editors, *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1846–1853. ACM, 2015.

[26] S. A. Savari and Dimitri P. Bertsekas. Finite termination of asynchronous iterative algorithms. *Parallel Computing*, 22(1):39–56, 1996.

[27] Stefano Sebastio, Michele Amoretti, and Alberto Lluch-Lafuente. A computational field framework for collaborative task execution in volunteer clouds. In Gregor Engels and Nelly Bencomo, editors, *9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2014, Proceedings, Hyderabad, India, June 2-3, 2014*, pages 105–114. ACM, 2014.

[28] Stefano Sebastio, Michele Amoretti, and Alberto Lluch-Lafuente. AVOCLOUDY: a simulator of volunteer clouds. *Softw., Pract. Exper.*, 46(1):3–30, 2016.

[29] Michael B. Smyth. Power domains. *J. Comput. Syst. Sci.*, 16(1):23–36, 1978.

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

[31] Mirko Viroli, Matteo Casadei, Sara Montagna, and Franco Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *TAAS*, 6(2):14, 2011.

[32] Mirko Viroli and Ferruccio Damiani. A calculus of self-stabilising computational fields. In eva Kühn and Rosario Pugliese, editors, *Coordination Models and Languages - 16th IFIP WG 6.1 International Conference, COORDINATION 2014, Held as Part of the 9th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, volume 8459 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2014.

[33] Mirko Viroli, Ferruccio Damiani, and Jacob Beal. A calculus of computational fields. In Carlos Canal and Massimo Villari, editors, *Advances in Service-Oriented and Cloud Computing - Workshops of ESOCC 2013, Málaga, Spain, September 11-13, 2013, Revised Selected Papers*, volume 393 of *Communications in Computer and Information Science*, pages 114–128. Springer, 2013.

[34] Shanhe Yi, Cheng Li, and Qun Li. A survey of fog computing: Concepts, applications and issues. In Qun Li and Dong Xuan, editors, *Proceedings of the 2015 Workshop on Mobile Big Data, Mobidata@MobiHoc 2015, Hangzhou, China, June 21, 2015*, pages 37–42. ACM, 2015.

This work is licensed under the Creative Commons Attribution-NoDerivs License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/2.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany