A Framework to Specify and Verify Computational Fields for Pervasive Computing Systems

Matteo Casadei and Mirko Viroli
Alma Mater Studiorum – Università di Bologna, Italy
{m.casadei,mirko.viroli}@unibo.it

***Abstract*—Pervasive context-aware computing networks call for designing algorithms for information propagation and reconfiguration that promote self-adaptation, namely, which can guarantee – at least to a probabilistic extent – certain reliability and robustness properties in spite of unpredicted changes and conditions. The possibility of formally analyzing their properties is obviously an essential engineering requirement, calling for general-purpose models and tools.**

As proposed in recent works, several such algorithms can be modeled by the notion of *computational field*: a dynamically evolving spatial data structure mapping every node of the network to a data value. Based on this idea, as a contribution toward formally verifying properties of pervasive computing systems, in this article we propose a specification language to model computational fields, and a framework based on PRISM stochastic model checker explicitly targeted at supporting temporal property verification. By a number of pervasive computing examples, we show that the proposed approach can be effectively used for quantitative analysis of systems running on networks composed of hundreds of nodes.

I. Introduction

The study and adoption of novel distributed approaches to computing is getting more and more importance as a result of the growing trend toward pervasive and mobile computing scenarios [22]. This is leading to developing a new class of applications, intrinsically distributed and mobile, ranging from providing a better interaction with the physical environment to managing and improving cooperation among human activities. This picture poses new challenges in the development of pervasive systems, that cannot be dealt with by traditional centralized approaches. In fact, these emerging domains require for applications to support: *(i) self-adaptivity*, i.e. the ability to cope with unpredicated changes like the mobility of system actors, locally managing the resulting changes in the computing environment in an unsupervised way; *(ii) context-awareness*, i.e. handling spatially situated activities depending on the surrounding physical and virtual context.

The *computational field* (or simply *field*) notion is recognized as a fundamental abstraction to design pervasive systems [11], [2], [4]. A field is a function mapping each node of a spatially distributed network of computing devices to a (possibly structured) value, representing some aspect of the local system state. It hence provides a viewpoint of distributed computing algorithms as spatial computations, namely, as the dynamic evolution of spatial data structures. As a reference example, a mobile agent can retrieve a device in a mobile and faulty network by "descending" the self-healing *gradient* (a special case of computational field) pumped by the device [2]. Computational fields have been adopted in several pervasive computing domains, surveyed in [6], to devise efficient and robust algorithms for routing and communication in sensor networks, provide support for context awareness in mobile pervasive computing applications, and coordinate task assignment in situated settings. Programming frameworks for computational fields include the TOTA middleware [12], the Proto language [2], [1], the chemical-oriented models proposed in [17], [19].

Despite the dynamic and unpredictable nature intrinsic to pervasive computing domains, very little effort has been devoted so far to define verification techniques able to formally analyze probabilistic and temporal properties of their algorithms, e.g. providing probability of convergence or average convergence time. In this article we tackle this issue by a framework for probabilistic model checking of computational fields. We propose a specification language to formally define and model such algorithms, and support their stochastic verification via translation into a CTMC (Continuous Time Markov Chain) model, most specifically, into a specification for the well-known PRISM stochastic model checker [8]. In fact, stochastic model checking allows to cope with the probabilistic aspects proper of computational fields and their distributed nature: probability plays a key role in both analyzing the unpredictable behaviors typical of open domains – such as pervasive systems – and deriving robust and self-adaptive algorithms for pervasive scenarios. State-space explosion is addressed by relying on approximate results according to the approach described in [7]. We discuss our approach by reference algorithms useful in pervasive domains, such as gradient diffusion, gradient descent, and information segregation.

The main contribution resulting from this article is a framework explicitly targeted at modeling and analyzing computational fields through stochastic model checking, demonstrating its applicability for networks composed of up to hundreds of nodes, as it happens with pervasive computing domains [22].

The remainder of the article is organized as follows. Section II details background on stochastic model checking; Section III introduces the details of our modeling language and its translation into PRISM; Section IV discusses the application of our framework to some example algorithms; and finally Section V concludes providing final remarks.

II. Approximate Stochastic Model Checking

In recent years, there has been a growing interest on formal models and verification of distributed, large-scale networks like those regarding pervasive computing. In this section, we present approximate stochastic model-checking as a general tool to verify quantitative properties on distributed systems, and in particular on computational field.

Stochastic Model Checking: Stochastic model checking [9] is basically the probabilistic extension of traditional model checking [5]. It is based on probabilistic models such as DTMC (Discrete-Time Markov Chain) and CTMC (Continuous-Time Markov Chain), which allow one to express the likelihood of the occurrence of certain transitions an algorithm (or system behavior) is composed of—in this article we deal with CTMC, for we shall consider (continuous) time an important aspect of system modeling. Verification over such systems basically amounts to computing the overall probability of the set of paths that satisfy the logic formula of interest, expressed in a suitable probabilistic temporal logic, e.g. CSL (Continuous Stochastic Logic) [10]: for instance, we might ask questions like *"will the system eventually reach state S with a probability greater than 80%?"*, or *"which is the probability for the system to stay in state S for at least* 1 *hour?"*.

Approximate Verification: A major drawback of model checking is *state-space explosion*: as a system grows in size, the number of states quickly diverges, easily making model checking unfeasible in practice. Given the size of systems modeling large networks (as typical in pervasive computing), exact model checking would be impractical: hence, it is necessary to consider approximate techniques.

Approximate model checking is basically about running a large set N of simulations per test case, with specific techniques to obtain statistic information out of them that can be interpreted as a model-checking (search in the whole state space) done with a given approximation ϵ and confidence δ—computed as $N \geq 4log(\frac{2}{\delta})/\epsilon^2$ [7]. As a realistic example, $\epsilon = 10^{-2}$ and $\delta = 10^{-3}$ require $N \approx 130'000$ number of independent simulations. By this technique one can in principle answer the same questions as in standard stochastic model checking, yet trading-off simulation time with quality of the result (approximation/confidence),namely, obtaining results with high confidence and good approximation may require time-consuming simulations performed by generating a large number N of simulation runs. On the other hand, the problem is state-space explosion is avoided, and hence bigger networks can typically be tackled. In the following we hence mostly deal with approximate stochastic model checking—which is simply referred to as stochastic model checking.

PRISM: One of the most well-known probabilistic model checker – which we will rely upon in this article – is PRISM [8]. PRISM models are specified in terms of modules whose behavior is expressed by transition rules and whose state is encoded in a set of discrete variables. In CTMC, transitions are associated with a rate and can be labelled—transitions with same label are synchronized. A transition follows the syntax "`[label] guard -> r:(update)`", where `guard` is a sequence of conditions that need to be satisfied to enable the transition, `r` represents transition rate, and `update` is a list of operations to update (part of) the state of the system by modifying variables.

Properties to verify are CSL formulas: for instance, property *"Which is the probability for variable* `state` *to reach value* `true` *within time* `T`*?"* is expressed by formula "`P=?[F<=T (state=true)]`", where `F` is the standard *future* operator of temporal logic.

Languages and Frameworks for Computational Fields: Computational fields promote the viewpoint of conceiving algorithms for computer networks in terms of evolution of distributed data structures.

Proto [1] is a reference language for modeling and implementing spatial computing systems, featuring the notion of computational field as a primary abstraction. Proto allows to define the behavior of single devices within a network, located at specific points in a continuous space. For instance, the specification:

```
(rep d (inf) (mux (sense 1) 0
        (min-hood (+ (nbr-range) (nbr d))))
)
```

associates each node of the network with its overall distance from a source node where `sense` operator is activated. Variable `d` is initialized to infinite, and updated to 0 where `sense` is activated (source node). In any other node n, the variable is assigned with the minimum value of `d` (as computed in any neighboring node n'), plus the distance between n and n'.

The applicability of computational fields for pervasive computing has already been studied and detailed in the series of works concerning the TOTA (Tuples On The Air) infrastructure [12], and recently in the context of chemical tuple spaces [17], [18] and pervasive service ecosystems [19], [21].

III. Framework

The framework presented here allows to model computational fields in pervasive and spatial systems via a specification language based on the concept of transition. This is basically a variant of the PRISM language, capturing the concept of "networked set of nodes". A generation process translates the model into a PRISM CTMC module, and then builds a complete PRISM specification representing the whole network, composed of as many PRISM modules as the number of nodes (devices) in the network: to this end, *module renaming* – a mechanism proper of PRISM – is exploited whenever possible to reduce code expansion. In other words, the framework essentially provides the necessary support for modeling computational field algorithms via a transition-based specification language, along with properties, rewards, and formulas. In particular, the specification language allows one to model each device in a pervasive system as a set of transition rules whereby the device – through observation of its neighbors – can change its state and/or influence the state of the neighbors. This mechanism is key to both model self-adaptativity (changing system behavior only through local interactions) and context-awareness (system behavior intrinsically depends on the local context) [20]. Finally, also properties, rewards and CSL formulas are translated into the corresponding PRISM components, so as to obtain a comprehensive PRISM specification ready to be model checked.

A. Architecture

The components of the framework are essentially a language translator and a PRISM generator. The language translator takes a specification written in our language – defining the behavior of a generic node in the pervasive network – and produces a corresponding PRISM template specification, which cannot yet be correctly parsed by PRISM, since some parts of the generated code are left unspecified: those are the parts involving actions related to neighbors. To this end, the PRISM generator takes a description of the network topology, along with a file containing the properties, rewards, and formulas to be verified, as input arguments . It then creates a complete PRISM code to be verified, as shown in Section IV.

As regards network generation, two basic modalities are possible. On the one hand, a set of *fixed* topologies are available: *(i)* square grid, *(ii)* square grid with diagonal links, and *(iii)* hexagonal grid. Additional parameters to be set in this case include: number of nodes, torus structure (nodes at the boundary of the grid are connected to those on the opposite side), and probability for each potential link to be actually present—the latter is especially useful when modeling real domains, as it allows to tackle the heterogeneity typical of real environments. Though we found the above cases covering a wide set of scenarios for experiments, we also allow *ad-hoc* topologies: in this case, the user has to provide an external file indicating the actual network structure to adopt, which is defined as the set of node pairs modeling links. Finally, it is possible to compactly define the initial states of nodes, i.e. of their variables as described below.

B. Language Syntax

We now introduce our specification language for computational fields. This language is state- and transition-based like that adopted in PRISM: however, despite PRISM, there is no concept of module, as our language simply aims at abstractly describing the behavior of a generic node in a pervasive network. Furthermore, in order to support computational field modeling, the language is provided with constructs that make it possible to describe the behavior of a node by referring to nodes in its neighborhood.

The syntax of the language is reported in Figure 1, where `r` is a meta-variable over real numbers, `L` over label names, `X` over variable names and `M` over node names (conceived as alphanumeric sequences). Notation $\overline{x}$ is generally used to denote a possibly empty list of elements of type x: in particular, $\overline{\mathtt{D}}$ stands for $\mathtt{D}_1 \ldots \mathtt{D}_n$, $\overline{\mathtt{T}}$ stands for $\mathtt{T}_1 \ldots \mathtt{T}_n$, $\overline{\mathtt{P}}$ stands for $\mathtt{P}_1$ `& ... &` $\mathtt{P}_n$, and $\overline{\mathtt{A}}$ for $\mathtt{A}_1$ `& ... &` $\mathtt{A}_n$. Additionally, as concerns the latter two cases, the empty list is represented by term `true`.

```
S ::= D̅ T̅
D ::= X : [n_l..n_u];
T ::= [L] P̅ --e--> A̅;
A ::= V'=e
P ::= b | N:=&f[e] | N:=&f[b]
f ::= any | min | max
e ::= r | V | (e) | e+e | e-e | e*e | e-e | -e | f[e]
b ::= e<=e | e<e | e>=e | e>e | e=e | e!=e
V ::= X | M.X | @.X
```

Fig. 1. Syntax of the specification language for computational fields.

In our language, a model specification `S` is essentially a list of variable definitions $\overline{\mathtt{D}}$ followed by behavior definition, i.e. a list of transitions $\overline{\mathtt{T}}$. New variables are introduced declaring their name `X`, a lower bound `n_l` and an upper bound `n_r`. Transitions provide a list of preconditions $\overline{\mathtt{P}}$ to be checked, a numerical markovian rate specified via a numeric expression `e`, and a list of updates $\overline{\mathtt{A}}$, formed by assignments.

An assignment `V'=e` is used to update the state of a variable with an expression `e`—see below for the actual shape of variables and expressions. Notation `V'` is retained as a PRISM legacy to mean the new value that `V` will take.

A precondition `P` is either a boolean expression `b`, or an assignment of kind `N:=&f[e]` or `N:=&f[b]`: this assigns `N` with the identifier of the node selected by function `f` based on the result of expression evaluation. Cases currently supported include: `N:=&any[b]`, meaning that any node where `b` is positively evaluated is non-deterministically selected; `N:=&max[e]`, meaning that the node having the greatest value resulting from evaluation of `e` is selected; and similarly for `N:=&min[e]`.

Numerical expressions include real numbers, variables, application of function `f` as above (yielding any, maximum or minimum value of an expression in the neighborhood), and any combination of them by standard mathematical operators and parenthesis. Similarly, boolean expressions are obtained by combining numerical expressions with comparison operators. Finally, and most importantly, variables can be qualified in three ways: `X` refers to a local variable, `M.X` to variable `X` in node `M`, and finally `@.X` refers to variable `X` in the currently considered neighboring node. Informally, when used outside the scope of a function `f`, `@.X` basically spawns one copy of the transition per neighboring node, otherwise it denotes the neighbor considered by function `f`. For instance, `N:=&any[X>@.X]` selects one neighboring node in which variable `X` is smaller than the local one, assigning its identifier to `N` for future use. Finally, as additional details not explicitly reported in the syntax, it is worth remarking that `N:=&f[e]` can appear just once in list $\overline{\mathtt{P}}$, and function `f[e]` cannot contain another nested `f[e]`.

For the sake of space, the details related to translating our specification language into PRISM are introduced via simple examples, which are sufficient enough to grasp the necessary details.

C. Example translations

Diffusion: As a first paradigmatic example to show the details of translation into PRISM code, we adopt the case of a computational field defining the diffusion of a sample value (1) so as to cover the whole network: initially, such a value is present only in one node, referred to as source. The complete specification modeling diffusion is simply:

```
value : [0..1];
[diff] value=0 -- 1.0 --> value'= max[@.value];
```

where `value` represents the value to be spread across the network: it is set to 1 in the source and 0 elsewhere. Diffusion occurs via rule `diff`: each node with `value` = 0 keeps updating `value` at rate 1.0 by choosing the greatest `value` among its neighbors. As a result, value 1 spreads across the whole network, until completely covering it. The corresponding PRISM code for one node of the network, produced by the generator, is

```
module node_1
  value_1 : [0..1] init 1;
  [diff_1] value_1=0 -> 1.0 :
          value_1 = max(value_3,value_2,value_4);
endmodule
```

where `value_1` represents the value to be spread on the network in `node_1`: it is worth noting that this node is the source since `value_1` is initialized to 1. The neighborhood of each node is entirely created by the generator in an automatic fashion, in accordance with the chosen topology. In rule `diff_1`, `value'=max[@.value]` is translated into `value_1'=max(value_3,value_2,value_4)`: `value_2`, `value_3`, and `value_4` are the values in the neighbors of `node_1`. Function `max` is a built-in PRISM function returning the greatest of its arguments: the returned value is then assigned to `value_1`.

Random Walk: This example models the random movement of a reference value (again, 1) across the network, starting from a source node toward a node designated as the target of the "walk". The corresponding specification is:

```
value : [0..1]; target : [0..1];
[move] value=1 & N:=&any[@.value=0]
      -- 1.0 -->
  value'=0 & N.value'=1;
```

where `value` represents the value to be moved across the network, and `target` = 1 is adopted to denote the target node of the walk. After choosing non-deterministically a neighbor where `value` = 0, rule `move` updates its local state by setting local `value` to 0, and `value` on the neighbor to 1. The translation into PRISM results in this code:

```
module node_1
  value_1 : [0..1] init 1;
  target_1 : [0..1] init 0;

  [move_1_2] value_2 = 0 -> 1.0 : (value_1'=0);
  [move_1_3] value_3 = 0 -> 1.0 : (value_1'=0);

  [move_2_1] true -> 1.0 : (value_1'=0);
  [move_3_1] true -> 1.0 : (value_1'=0);
endmodule
```

that represents the specification for a node of the network identified as `node_1`, which is the source of the random walk (`value_1` set to 1). The neighbors of this node are `node_2` and `node_3`. In particular, rules `move_1_2` and `move_1_3` model the nondeterministic choice of one of the neighbors of `node_1` satisfying condition `@.value=0`: this condition is translated into guards `value_2` = 0 and `value_3` = 0 appearing in rules `move_1_2` and `move_1_3`, respectively. Put simply, the nondeterministic choice specified in our language via `any[@.value=0]` is modeled in PRISM as a race between the `move_i_j` transitions satisfying condition `value_j`= 0, where `i` denotes current node and `j` ranges among neighbors of i.

Once a transition is chosen, the corresponding update is executed, so as to set `value_1` to 0. The value in the chosen node has still to be set to 1: this cannot be done directly in the module of `node_1`, as PRISM does not allow to change the value of variables defined in modules external to the current one. It is then necessary to exploit PRISM *synchronization*, that makes it is possible to update variables in external modules by synchronizing transitions with the same name in different modules, that are then executed simultaneously. Accordingly, each neighbor `j` of node `i` is as well provided with a transition "`[move_i_j] true -> 1.0 : (value_j'=0)`", which is always enabled and executed together with the corresponding `move_i_j` rule defined in the module of `node_i`.

D. Property Specification and Rewards

Our specification language allows to specify *properties* and *rewards*. On the one hand, properties are a necessary component to perform verification, as they identify states of interest whose probability to occur can be analyzed via stochastic model checking. On the other hand, rewards make it possible to further extend computational field analysis, making it possible to not only reason about probabilities for a model to behave in a certain fashion, but also quantitatively measure cumulative performance parameters regarding system behavior—e.g., how many times a transition fired.

A property is expressed as "`property "name" = Q [ Bexp ];`", defining a new property `name`, where `Q` represents a quantifier on `Bexp`, which is a PRISM boolean expression over model variables. The quantifiers that can be adopted are `forall` and `exist`, meaning that `Bexp` must be satisfied for every node of the network and for at least one node of the network, respectively. Typical logical connectives among variables adopted in boolean expressions are: *and* (represented by symbol `&`), *or* (symbol `|`), and *not* (symbol `!`). An example of property specified on diffusion is:

```
property "diffusion_complete" = forall[value=1];
```

stating that diffusion is complete in every node of the network. In the random walk example the following property can be of interest:

```
property "target_reached" = exist[target=1 & value=1];
```

It states that the target of the random walk has been reached if there exists a node where $\texttt{target} = 1$ (denoting the target node) and $\texttt{value} = 1$. A property is then simply translated into a PRISM label, a construct denoting specific system states. For instance, `diffusion_complete` is translated into this PRISM label:

```
label "diffusion_complete" =
   (value_1=1) & (value_2=1) & ... & (value_n=1);
```

which is a simple expansion of the property specified in our specification language.

Rewards adopt a syntax similar to that used in PRISM rewards: from a practical viewpoint, a reward in our specification language is a compact form of a PRISM reward. As a reference example, consider the following reward, called `hops`, defined on diffusion:

```
rewards "hops" = [diff] true : 1;
```

which increments `hops` by 1 at each execution of `diff` in any node of the network. This is translated into the PRISM reward:

```
rewards "hops"
  [diff_1] true : 1; ... [diff_n] true : 1;
endrewards
```

IV. Model Checking Computational Fields

Based on the above discussed specification language and the associated properties/rewards, we here present some paradigmatic examples of computational fields, and accordingly show the resulting analysis performed via stochastic model checking. For each example, we show the corresponding model specified in our language, a description of the important properties to be analyzed and verified, and finally the results obtained via model checking of those properties in PRISM.

Reference Topology: For the sake of uniformity and simplicity in presenting the experiments, we find it useful to consider one reference network topology to use. Although the choice may evidently affect the results of model checking, and many topologies could be considered and compared, we here stick to a single case, which we consider to be of a general validity for the context of pervasive systems. On the one hand, locations are placed as nodes in a grid network, such that each location has in its proximity 8 nodes, 4 in the horizontal/vertical direction, and 4 in the diagonal direction—nodes at the boundary of the grid are connected to the nodes on the opposite side (both in the horizontal/vertical and diagonal directions) so as to form a torus. This choice is motivated by the fact that very often computing devices are placed more or less uniformly over the space formed by buildings, corridors, or rooms of the pervasive computing systems of interest. On the other hand, exceptions to this case are the norm, hence we introduce some randomness in the topology to tackle heterogeneity of the environment, as well as failures and so on. Accordingly, links between nodes are chosen in a random way: namely, the probability that a node is connected to one of the 8 in its proximity is 50%: in this way, the average

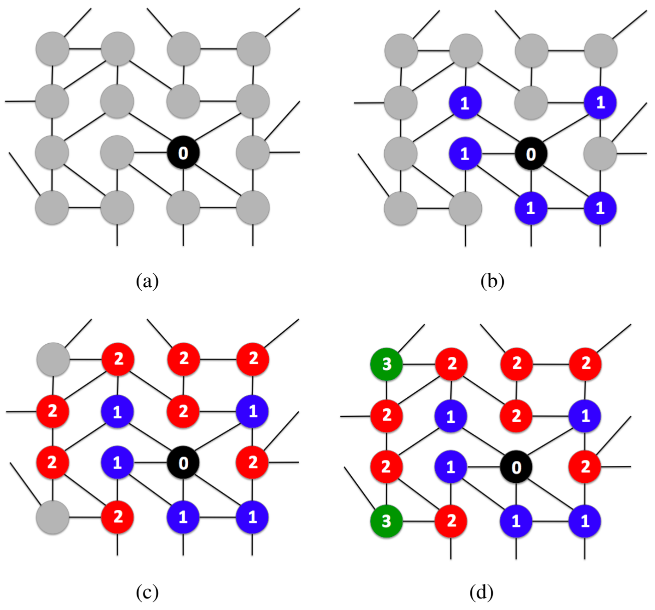


Fig. 2. Example of gradient diffusion on a 4x4 random torus topology.

```
pump  : [0..1];  field : [0..MAX];

[]     pump=1 & field>0 -- 1.0 --> field'= 0;
[diff] pump=0 -- 1.0 --> field'= min[@.field]+1;
```

Fig. 3. Specification of the gradient algorithm.

number of connections in the neighborhood of a node is 4. This topology is hereafter referred to as *random torus*.

A. Gradient Diffusion

Gradients are a particular kind of computational field where the value of the field in a given node depends solely on a notion of "distance" from the so-called gradient source node. Here, we consider a gradient where field is represented by an integer value, denoting the distance in terms of minimum number of hops from the source. The gradient is completely established when all nodes are "covered" with a proper field value. Figure 2 shows an example of gradient diffusion on a 4x4 random torus topology.

Gradients are actually useful in pervasive domains, in situations where it is necessary to find a (minimum) route to a specific device in the network, as well as a basic pattern to form virtual communication channels between nodes that need to interact with one another. In general, gradient is the basic brick to inject self-organization in a network, making a local situation affect a system in a more global way through (possibly selective) propagation [3]. Furthermore, gradient-based algorithms can be implemented in order to be flexible, i.e. able to adapt to changes in the network, such as node failures, connection changes (reflecting node mobility), and new nodes entering the network: the gradient algorithm modeled in the next is shown to be able to reconfigure in response to changes in the network.

Gradient Model: Figure 3 shows a simple model of gradient diffusion specified through our formal language: the model simply defines the behavior enforced in each node of the network—it is similar to the Proto example shown in Section II. Variable `pump` is an integer value used to denote the source of the gradient: it is initialized to 1 in the source node and set to 0 in every other node. The `field` variable represents the actual value of the gradient in a node: it is initially set to `MAX` in every node, where `MAX` is a value higher than the maximum value the gradient can reach once completely diffused in network—the so-called *diameter* of the network. Put simply, nodes not yet reached by gradient diffusion will have `field` = `MAX`. The behavior of each node is essentially specified by two rules: the first applies only on the source node, resulting in setting the value of the gradient to 0. The second rule (`diff`) is applied on other nodes of the network, and set `field` to 1 plus the minimum value of the gradient among the neighboring nodes. It is worth noting that these rules are continuosly executed at markovian rate 1.0 in each node—namely, the time between two executions is drawn by a negative exponential distribution of probability, following the well-known memoryless property. This means that each node updates in an independent and asynchronous way with respect to others, so that gradient diffusion and its consequent stabilization are achieved by emergence: this also allows gradient to self-adapt to changes occurring in the network. Indeed, if e.g. a node disconnects from the network, its neighboring nodes are able to automatically and independently update their gradient value.

Properties to Verify: Among the different properties we may be interested to analyze on a gradient, one of the most important is undoubtably how long it takes for a gradient to completely spread over all the network and get stable. For instance, if you think of a gradient exploited to diffuse information about a specific device or to establish a connection between two devices, it is important to know how long it is needed for the gradient to cover the network. A gradient is completely established and stabilized when for each non-source node i of the network (i.e. where `pump` = 0), variable `field` is precisely 1 plus the minimum value of `field` in the neighborhood. Accordingly, in our language we can define the following property:

```
property "established_gradient" =
      forall[ (pump=0 & field=min(@.field)+1) |
             (pump=1 & field=0)  ];
```

A further property that would be interesting to analyze on the network concerns the cost for the gradient to get established, expressed in terms of network hops necessary for the gradient to spread across the whole network. Often, in pervasive scenarios, communication among devices can be really expensive due to power/battery constraints, so that being aware of the number of hops necessary for completely establishing a gradient becomes important. This is perhaps crucial also in order to trade off performance (time necessary to establish the gradient) and cost—expressed here in terms of number of hops. In order to measure the number of hops needed to establish the gradient, it is possible to define a *reward*, which allows to reason on quantitative measures regarding model behavior. Expected number of hops can be considered by introducing the following reward:

```
rewards "hops" =  [diff] true : 1;
```

that assigns a reward of 1 to every execution of transition `diff` in any of the nodes composing the network.

Experimental Results: Based on the properties identified above, here we show some of the results obtained by applying approximate stochastic model checking on the gradient algorithm, for different network instances. Given the large size of the model instances verified, approximate model checking was adopted. To this end, we chose an approximation $\epsilon = 10^{-2}$ and a confidence $\delta = 10^{-3}$: this means that the probability for the obtained results to be affected by an error less than (or equal to) 10^{-2} is greater than 99.9%. According to the formula described in Section II, the corresponding number of runs required to meet these approximation and confidence values is about $130'000$.

The first property verified on the gradient algorithm concerns the time necessary for the gradient to be completely established in the network. This is specified by the following CSL-like property expressed by the PRISM syntax:

```
P=? [true U<=k  "established_gradient"]
```

which is a bounded-until path property, simply returning the probability to achieve the `established_gradient` state within `k` time units. This property was verified against different 5xN random torus instances, with N $\in \{5, 10, 15, 20, 25\}$ and `k` ranging from 0 to 30 time units: the corresponding results are shown in Figure 4 (a). For each instance, the probability of achieving the `established_gradient` state gets 1 after a certain time—still considering the approximation due to ϵ and δ. Before that stabilisation time, probability of convergence is of course smaller than 1. Figure 4 (b) shows the trend of the minimum time necessary to achieve an established gradient with probability 1 over the different investigated instances: it is easy to see that minimum time increases in a (quasi) linear way as the size of the network grows.

As regards verifying the expected number of hops needed to establish the gradient, the following reachability reward-based quantitative property was exploited:

```
R{"hops"}=? [ F "established_gradient" ]
```

that – according to the `"hops"` reward structure – returns the expected number of hops to achieve the `established_gradient` state. Figure 5 (up) reports the expected number of hops to establish the gradient, obtained from verifying the property on the same 5xN random torus instances adopted in previous experiments: it is easy to recognize a linear increase.

We now describe an important issue concerning the complexity of model checking from the standpoint of time: understanding how much time it takes to verify properties is fundamental to assess feasibility of model checking. Figure

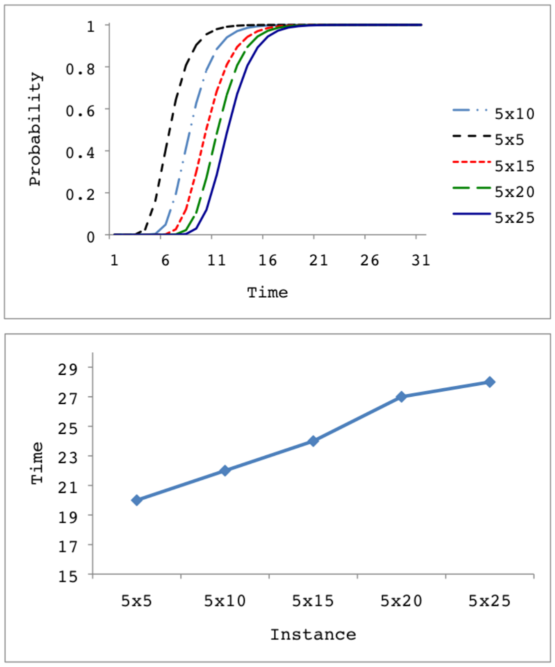


Fig. 4. Probability of achieving an established gradient over time on different 5xN random torus instances (up), and minimum time necessary to achieve the established gradient with probability 1 over the different instances (down).

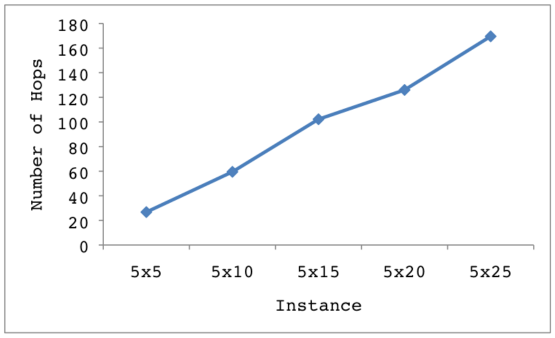


Fig. 5. Expected number of hops required for achieving an established gradient.

6 reports the time necessary to perform model checking on the investigated gradient instances, executed on a computer equipped with two 2.66 Ghz Dual-Core Xeon processors (4 MB L2 cache per processor), 2 GB 667 MHz DDR2 RAM, and 1.33 GHz bus. Note that, as expected, it can be clearly seen that time growth is polynomial. However, it is worth pointing out that the magnitude of these times clearly demonstrates the viability of (approximate) stochastic model checking even on networks composed of hundreds of nodes. The same analysis about time complexity of model checking was conducted also on the algorithms presented in the rest of the article, leading to similar conclusions. More generally, we believe that using state-of-the art hardware for simulation, and implementing easy techniques of distribution of those simulation runs (on different machines and cores) will soon allow us to simulate pervasive computing scenarios of very large scale in few hours and with good approximation.

B. Gradient Descent

We here consider another key algorithm for computational fields (see e.g. [12]) aimed at supporting data retrieval in pervasive computing scenarios, allowing data to follow downhill a gradient until reaching its source. This can be useful for retrieving specific information/services in spatially distributed networks of devices: e.g. a user starts diffusing a gradient containing a query for the specific information/service she/he is interested in: once the gradient reaches the location containing the sought information, such information follows downhill the gradient until coming to its source, i.e. the location of the user requiring the information/service.

In the example reported below, gradient descent occurs in a probabilistic way: the sought information follows the gradient by choosing probabilistically a direction where the gradient value is low. In other words, the lower the gradient value of a node in the neighborhood of the current one, the higher the probability for the sought information to move to that node.

Gradient Descent Model: The behavior to be enforced in each node of the network, modeling gradient descent, is defined according to the specification reported in Figure 7. The information descending the gradient is represented through variable `desc` set to 1: initially, `desc` is 1 in the target node, and 0 elsewhere. As soon as the gradient reaches the target node, information starts descending the gradient by rule `move`: this is modeled by making value 1 of `desc` move across the network, following downhill the gradient. This rule applies to nodes where `desc` = 1. It states that any neighbor with a lower gradient value (`&any[@.field<field]`) can be chosen and designated as `N`. Then, the corresponding transition occurs with a rate depending on the normalized difference between gradient value in the current node and that in neighbor `N`: the higher this difference, the higher the likelihood to move to `N`—overall transition rate is however 1.0 due to normalization. As a result of transition execution, `desc` is set to 0 in the current node and 1 in `N`.

Properties to Verify: We are first interested to know the total time necessary for `desc` to completely descend the gradient and reach its source. This time needs to be calculated from the start of gradient diffusion until `desc` reaches gradient source: this is due to the fact that, in a real scenario, a request for

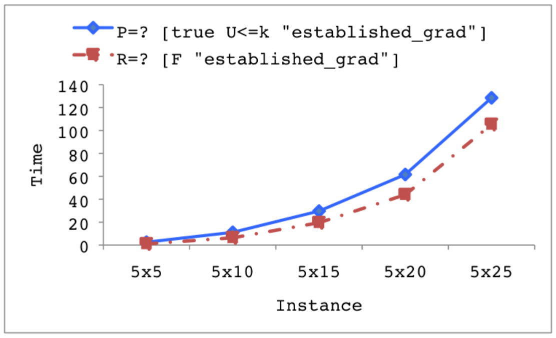


Fig. 6. Gradient diffusion: time required to perform approximate model checking (in minutes) over the investigated 5xN random torus instances.

```
pump  : [0..1];
field : [0..MAX];
desc  : [0..1];

[]     pump=1 & field>0 -- 1.0 --> field'= 0;
[diff] pump=0 -- 1.0 --> field'= min[@.field]+1;
[move] desc=1 & N:=&any[@.field<field]
          -- (field-@.field)/@.field/
             sum((field-@.field)/@.field) -->
       desc'=0 & N.desc'=1;
```

Fig. 7. Specification defining gradient descent.

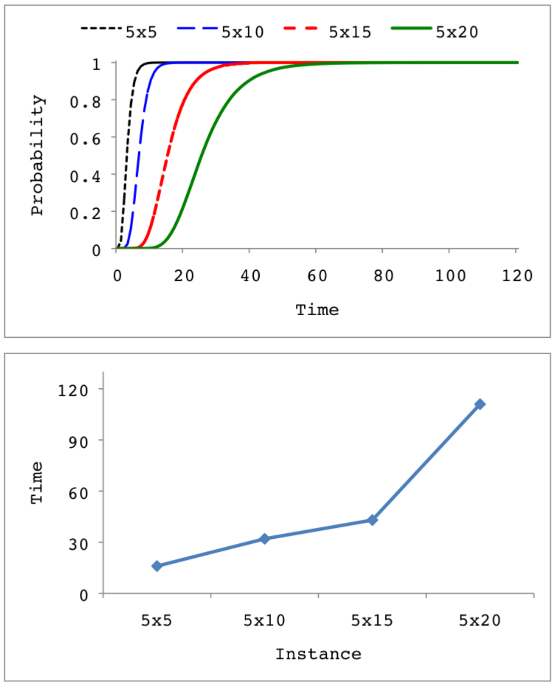


Fig. 8. Probability of descending a gradient and coming to its source over time on different 5xN random torus instances (up), and corresponding trend of the minimum time necessary to come to gradient source with probability 1 over the same instances (down).

some kind of information (represented by desc) starts with gradient diffusion and gets completed when the information arrives to gradient source (the requesting node). We can state that gradient descent is finished if the desc $= 1$ condition applies in the source node, namely:

```
property "descent_complete" = exist[pump=1 & desc=1];
```

Again, as a further property, it is also interesting to analyze the number of network hops needed for desc to reach the source. This is simply obtained by assigning a reward of 1 to transition move as in previous subsection—this reward is hereafter referred to as descending_hops.

Experimental Results: Approximate model checking on gradient descent was executed with the same confidence and approximation values chosen for gradient diffusion. The first verified property was the time needed for desc to completely descent the gradient, which can be expressed according to the CSL formula:

```
P=? [true U<=k "descent_complete"]
```

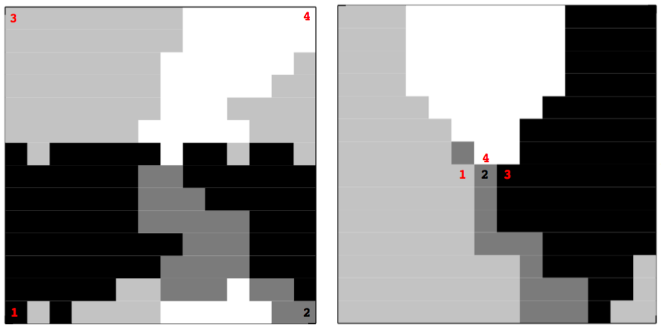


Fig. 9. Segregation: sample runs on a 15x15 random torus showing two possible spatial distributions of regions, starting from data sources placed at the corners of the network (left) and as an "inverted T" (right). Numbers represent the position of the sources of the different data kinds.

simply returning the probability of achieving the descent_complete state within k time units. This property was verified on different 5xN random torus instances, with N $\in \{5, 10, 15, 20\}$ and k ranging from 0 to 80 time units: in each instance, gradient source and target (represented by desc= 1) were placed at the maximum practicable distance, which is $\lceil N/2 \rceil$ on the adopted topology. The corresponding results are shown in Figure 8 (up). There, it can be seen that, for each instance, the probability of getting the descent_complete state becomes 1 after a given time, representing the minimum time needed for completing the descent. The corresponding trend is reported in Figure 8 (down): time increase is polynomial with instance size.

As in the case of gradient diffusion, we can also verify the expected number of network hops necessary to complete gradient descent, by using the descending_hops reward. The corresponding results on the 5xN random torus instances are: 3.021 hops necessary on the 5x5 instance, 5.025 on the 5x10, 7.119 on the 5x15, and 10.016 on the 5x20. As in the case of gradient diffusion, these results correspond to a quasi linear increase with instance size.

C. Segregation

Segregation is a further computational field algorithm based on gradients, conceived to diffuse data across pervasive networks: here, the objective is to keep data of a certain kind segregated from data of different kinds so as to form spatial regions characterized by data organized per kind. In other words, it is possible to conceive a network where several gradient sources are placed in different nodes, one per data kind. Such different kinds of data are then diffused on the network, stopping where another region is found so as to ultimately partition the whole network without overlaps. Examples of segregation on a 15x15 random torus are reported in Figures 9 (left) and (right), showing the final data organization starting from gradient sources placed at the corners of the network and as an "inverted T", respectively. This is for instance useful in pervasive service scenarios, to support spatial competition among services providing the same functionality: in fact, segregation can support competition, allowing services to organize and satisfy user requests by adaptively organizing their distribution on the network.

Segregation Model: A model for segregation is shown in Figure 10. The only additional variable with respect to gradient model is `segr`, whose value represents the data kind stored in a given node. In this model, `segr` ranges from 0 to 4, meaning that we are considering four different kinds of data (from 1 to 4) to be diffused in the network—0 denotes a node containing no data, i.e. a node not yet reached by diffusion of any kind of data. Segregation is actually modeled via rule `spread`, simply stating that the data kind hosted by current node will be that of any of its neighbors with a smaller gradient value: as usual, this neighbor is designated as `N`, and its value set to `&any[@.field<field]`. Then, the corresponding transition occurs with a rate depending on the normalized difference between gradient value in the current node and that in neighbor `N`: the higher this difference, the higher the likelihood for the current node to host the same data kind hosted by `N`. Hence, this basically models a probabilistic data diffusion toward nodes with higher gradient values—with a competition between different data kinds around the boundary of regions.

Properties to Verify: As a first property to be verified, we are interested in the time necessary for the different kinds of data to spread in the network and completely cover it. The corresponding state – featuring every node of the network hosting a data kind – can be expressed in our language via the following property:

```
property "segregation_complete" = forall[ segr > 0 ];
```

which states that all the nodes of the network need to have a value of `segr` > 0, meaning they all host a certain kind of data.

Additionally, it would be worth also to analyze the expected size achieved by the various regions formed as a result of data diffusion: the size of each region depends on the location of the corresponding data source. To measure this property for the region formed by diffusion of data kind `i`, we need to define a reward such as:

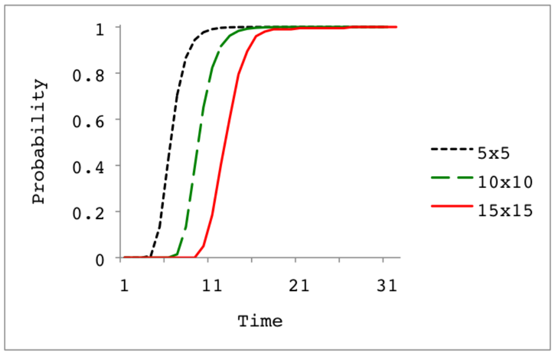
```
reward "region_i_size" = (segr = i) : 1;
```

where `i` represents the data kind to analyze. This reward assigns a reward of 1 to each node where `segr` $=$ `i`, i.e. hosting data kind `i`.

Experimental Results: Approximate model checking on segregation was again performed with the usual confidence and approximation values. We initially verified total time necessary for completing segregation, as previously described. To this end, the following CSL property was adopted:

```
P=? [true U<=k "segregation_complete"]
```

simply returning the probability of achieving the `segregation_complete` state within `k` time units. This property was verified against different NxN random torus instances, with $N \in \{5, 10, 15\}$ and $k \in [0, 40]$, starting



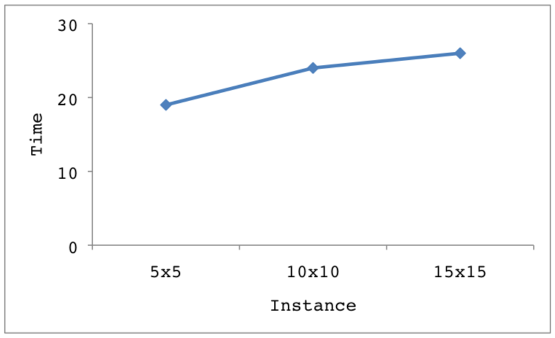


Fig. 11. Probability of achieving complete segregation over time on different NxN random torus instances (up), and corresponding trend of the minimum time necessary to achieve complete segregation with probability 1 over the same instances (down).

with the four data sources arbitrarily placed at the four corners of the network: since the topology is a torus, this configuration defines a sort of "square"—different initial configurations can anyway be exploited. The corresponding results are reported in Figure 11 (up): it is easy to recognize that, after a certain time, probability of completing segregation becomes 1 for all the instances. Complementarily, Figure 11 (down) shows the trend of the minimum time needed to complete segregation with probability 1 over the analyzed instances: the corresponding time increase is quasi linear.

As regards the expected size of each formed region `i` (where `i` denotes data kind), we relied on the following cumulative reward-based property:

```
R{"region_i_size"}=? [ I = k ]
```

that returns the sum of rewards cumulated concerning region `i` at time `k`. Constant `k` has to be chosen carefully: it has to be at least equal to the minimum time at which the probability of achieving complete segregation becomes 1. This value can be inferred by looking at the results shown in Figure 11. The results obtained by verifying this property on a 10x10 random torus (with data sources placed to form an "inverted T" like in Figure 9 (b)) are: 25.61 nodes for data kind 1, 15.63 for kind 2, 30.86 for kind 3, and 29.87 for kind 4. The expected region size varies a lot among the different regions: in particular, the region formed by data kind 2 has a noticeable lower expected number of nodes as the corresponding source position limits the space available for diffusion, which is constrained by the close proximity of data sources 1 and 3.

```
pump  : [0..1];   field : [0..MAX];   segr  : [0..4];

[]       pump=1 & field>0 -- 1.0 --> field'= 0 ;
[diff]   pump=0           -- 1.0 --> field'= min[@.field]+1;
[spread] N:=&any[@.field<field]
         -- (field-@.field)/@.field/sum((field-@.field)/@.field) -->
         segr'=N.segr;
```

Fig. 10. Specification of the segregation algorithm.

V. Conclusion and Future Work

To the best of our knowledge, this article represents the first explicit attempt to deal with modeling and verification of computational field algorithms, which are basic buildling blocks for pervasive computing domains as studied in [11], [2]. To this end, a specification language and a framework for stochastic model checking were provided, that are targeted at quantitative analysis of computational fields. As concerns stochastic model checking, we relied on the PRISM probabilistic model checker (which was also an obvious inspiration to our syntactic and semantic approach), showing how the proposed specification language can be translated into PRISM code, so as to provide an easy way to perform quantitative analysis.

Our verification framework can be upgraded with new ingredients. A notable example would be the introduction of mechanisms to model node failures and mobility, so as to verify aspects concerning with self-adaptation to probabilistic events. Then, the flexibility and expressiveness of our language should be extended to deal with a wider set of algorithms: as such, it would be interesting to evaluate the possibility of translating Proto [1] and the eco-law lanaguage of the SAPERE appraoch [19] into PRISM using our language as an intermediate step. Alternatively, it would be interesting to turn existing simulators like e.g. ALCHEMIST [16] into approximate stochastic model-checkers. Finally, it would be of general interest to devise a methodology for applying stochastic model checking to multiagent systems, which we believe would enjoy the use of meta-models based on the notion of artifact [14], [15], [13].

Acknowledgments

This work has been supported by the EU-FP7-FET Proactive project SAPERE Self-aware Pervasive Service Ecosystems, under contract no.256873.

References

[1] J. Beal. The MIT proto language. http://groups.csail.mit.edu/stpg/proto.html, 2008.

[2] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *IEEE Intelligent Systems*, 21(2):10–19, 2006.

[3] J. Beal, J. Bachrach, D. Vickery, and M. Tobenkin. Fast self-healing gradients. In R. L. Wainwright and H. Haddad, editors, *SAC*, pages 1969–1975. ACM, 2008.

[4] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. Organizing the aggregate: Languages for spatial computing. *CoRR*, abs/1202.5509, 2012.

[5] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, January 2000.

[6] J. L. Fernandez-Marquez, G. Di Marzo Serugendo, S. Montagna, M. Viroli, and J. L. Arcos. Description and composition of bio-inspired design patterns: a complete overview. *Natural Computing*, May 2012. Online First.

[7] T. Hérault, R. Lassaigne, F. Magniette, and S. Peyronnet. Approximate probabilistic model checking. In *Proc. 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04)*, volume 2937 of *LNCS*. Springer, 2004.

[8] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. Prism: A tool for automatic verification of probabilistic systems. In *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 441–444. Springer, 2006.

[9] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with prism: a hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):128–142, 2004.

[10] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer, 2007.

[11] M. Mamei and F. Zambonelli. *Field-based Coordination for Pervasive Multiagent Systems*. Springer Verlag, 2006.

[12] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. Softw. Eng. Methodol.*, 18(4), 2009.

[13] A. Omicini, A. Ricci, and M. Viroli. An algebraic approach for modelling organisation, roles and contexts in MAS. *Applicable Algebra in Engineering, Communication and Computing*, 16(2-3):151–178, Aug. 2005.

[14] A. Omicini, A. Ricci, and M. Viroli. Coordination artifacts as first-class abstractions for MAS engineering: State of the research. In *Software Engineering for Multi-Agent Systems IV: Research Issues and Practical Applications*, volume 3914 of *LNAI*, pages 71–90. Springer, Apr. 2006.

[15] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3), June 2008.

[16] D. Pianini, S. Montagna, and M. Viroli. A chemical inspired simulation framework for pervasive services ecosystems. In *Proceedings of the Federated Conference on Computer Science and Information Systems*, pages 675–682, Szczecin, Poland, 18-21 September 2011. IEEE Computer Society Press.

[17] M. Viroli and M. Casadei. Biochemical tuple spaces for self-organising coordination. In *Coordination Languages and Models*, volume 5521 of *LNCS*, pages 143–162. Springer-Verlag, June 2009.

[18] M. Viroli, M. Casadei, S. Montagna, and F. Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Transactions on Autonomous and Adaptive Systems*, 6(2):14:1 – 14:24, June 2011.

[19] M. Viroli, D. Pianini, S. Montagna, and G. Stevenson. Pervasive ecosystems: a coordination model based on semantic chemistry. In *27th Annual ACM Symposium on Applied Computing (SAC 2012)*, Riva del Garda, TN, Italy, 26-30 March 2012. ACM.

[20] M. Viroli and F. Zambonelli. A biochemical approach to adaptive service ecosystems. *Information Sciences*, 180(10):1876–1892, May 2010.

[21] F. Zambonelli, G. Castelli, L. Ferrari, M. Mamei, A. Rosi, G. D. M. Serugendo, M. Risoldi, A.-E. Tchao, S. Dobson, G. Stevenson, J. Ye, E. Nardini, A. Omicini, S. Montagna, M. Viroli, A. Ferscha, S. Maschek, and B. Wally. Self-aware pervasive service ecosystems. *Procedia CS*, 7:197–199, 2011.

[22] F. Zambonelli and M. Viroli. A survey on nature-inspired metaphors for pervasive service ecosystems. *International Journal of Pervasive Computing and Communications*, 7(3):186–204, 2011.