Towards a logic and chemical based coordination model

DE ANGELIS, Francesco, DI MARZO SERUGENDO, Giovanna

Abstract

In this paper we propose a coordination model that integrates logic programs into a chemical-based coordination framework. The goal is to engineer self-organising systems as well as assess their emergent global properties. Our model is generic and accommodates various logics and it is also powerful enough to supersede various chemical-based approaches. By tuning the internal logic language we can tackle and solve coordination problems in a rigorous way, without renouncing to important engineering properties such as compactness, modularity and reusability of code. We present our logic-based coordination model and we shows how to engineer well-know self-organising mechanisms and how to assess their corresponding emergent properties.

Reference

DE ANGELIS, Francesco, DI MARZO SERUGENDO, Giovanna. Towards a logic and chemical based coordination model. 2015

Available at:
http://archive-ouverte.unige.ch/unige:55396

Disclaimer: layout of this document may differ from the published version.
Towards a logic and chemical based coordination model

Francesco L. De Angelis
francesco.deangelis@unige.ch and
Giovanna Di Marzo Serugendo
giovanna.dimarzo@unige.ch

Institute of Information Services Science, University of Geneva, SWITZERLAND

Abstract In this paper we propose a coordination model that integrates logic programs into a chemical based coordination framework. The goal is to engineer self-organising systems as well as assess their emergent global properties. Our model is generic and accommodates various logics and it is also powerful enough to supersede various chemical based approaches. By tuning the internal logic language we can tackle and solve coordination problems in a rigorous way, without renouncing to important engineering properties such as compactness, modularity and reusability of code. We present our logic-based coordination model and we shows how to engineer well-know self-organising mechanisms and how to assess their corresponding emergent properties.

1 Introduction

Engineering self-organising and self-adaptive systems from scratch is a complex task; consequently nowadays, engineers and scientists build such a kind of systems by resorting to coordination models and middlewares. Coordination models have proven useful for designing and implementing distributed systems. They are particularly appealing for developing self-organising systems, since the shared tuple space on which they are based is a powerful paradigm to implement self-organising mechanisms, particularly those requiring indirect communication (e.g. stigmergy) [16]. Chemical based coordination models are a category of coordination models that use the chemical reaction metaphor and have proven useful to implement several types of self-organising mechanisms [?]. A well-known difficulty designing self-organising systems stem from the analysis, validation and verification (at design-time or run-time) of so-called emergent properties - i.e. properties that can be observed at a global level but that none of the interacting entities exhibit on its own. Few coordination models integrate features supporting the validation of emergent properties, none of them relying on the chemical metaphor.
In this paper, we propose to enrich a chemical based coordination model with the notion of Logic Fragments (i.e. a combination of logic programs). Our logic-based coordination model allows agents to inject Logic Fragments into the shared space. Those fragments actually define on-the-fly ad hoc chemical reactions that apply on matching data tuples present in the system, removing tuples and producing new tuples, possibly producing also new Logic Fragments. Our model is defined independently of the logic language used to define the syntax of the Logic Fragment, an actual instantiation and implementation of the model can use its own logic(s). The advent of new families of logic languages (e.g. [17]) has enriched the paradigm of logic programming, allowing, among other things, practical formalisation and manipulation of data inconsistency, knowledge representation of partial information and constraints satisfaction. By combining those logics with a chemical based coordination model, we argue that global properties can be verified at design time.

Section 2 discusses related works, section 3 presents our logic-based coordination model. Section 4 shows several case studies about pattern recognition scenarios and the gradient and chemotaxis pattern. Finally, section 5 concludes the technical report.

2 Related works

2.1 Chemical based coordination models

An important class of coordination models is represented by so-called chemical based coordination models, where “chemical” stands for the process of imitating the behaviours of chemical compounds in chemical systems. Such a kind of models turns to be powerful in decreasing and managing the complexity of a coordination problem by modeling the objects of the problem domain in terms of simple entities which react among each other according to predefined coordination rules, in the same way molecules react according to chemical laws.

Gamma (General Abstract Model for Multiset manipulation) [2] and its evolutions historically represents an important chemical-inspired coordination model. The core of the model is based on the concept of virtual chemical reactions expressed through condition-action rewriting pairs. Virtual chemical reactions are applied on input multisets which satisfy a condition statement and they produce as output multisets where elements are modified according to the corresponding action (like for chemical compounds); the execution of virtual chemical reactions satisfying a condition pair is nondeterministic. Gamma presents two remarkable properties: (i)
the constructs of the model implicitly support the definition of parallel programs; (ii) the language was proposed in the context of systematic program derivation and correctness and well as termination of programs is easy to prove ([7]). Its major drawback is represented by the complexity of modeling real large applications.

The rationale of virtual chemical reactions is reconsidered in the Fraglets model [15] to create new mechanisms for process coordinations exhibiting self-organising properties. The Fraglet model can be represented as a restricted and specialised protocol-oriented version of Gamma, where protocols are defined in terms of multiset entities (Fraglets) manipulated through a predefined set of actions. One of the major features of such a model is the capability of expressing an entire communication protocol in an self-contained way by combining several atomic sub-parts of it.

The SAPERE model [4] (Figure 1a) is a coordination model for multiagent pervasive systems inspired by chemical reactions. It is based on four main concepts: Live Semantic Annotations (LSAs), LSA Tuple Space, agents and eco-laws. LSA are tuples of types (name, value) used to store applications data. For example, a tuple of type (date, 04/04/1988) can be used to define a hypothetical date. LSAs belonging to a computing node are stored in a shared container named LSA Tuple Space. Each LSA is associated with an agent, an external entity that implements some domain-specific logic. For example, agents can represent sensors, services or general applications that want to interact with the LSA space - injecting or retrieving LSAs from the LSA space. Inside the shared container, tuples react in a virtual chemical way by using a predefined set of coordination rules named eco-laws, which can (i) instantiate relationships among LSAs (Bonding eco-law) (ii) aggregate them (Aggregate eco-law), (iii) delete them, (Decay eco-law) and (iv) spread them across remote LSA Tuples Spaces (Spreading eco-law). Spontaneous executions of eco-laws can be fired after setting specific commands (named operators) as tuple values. When a tuple is modified by an eco-law, its corresponding agent is notified: in this way, agents react to virtual chemical reactions according to the domain-specific logic they implement. The implementation of the SAPERE model, named SAPERE middleware, has been proven to be powerful enough and robust to permit the development of several kinds of real distributed self-adaptive and self-organising applications, as reported in [?]. In particular, when instantiated on the devices of a network, the SAPERE middleware can be used to realize context-aware data flows, secure channels for mobile ad-hoc networks and self-organising crowd-steering mechanisms. Nevertheless, the model does not aim at proving
correctness or emergence of global properties programs built on it: this means that proving correctness of applications may turn to be a complex task.

2.2 Formal approaches for tuple based coordination models

Coordination models based on tuple spaces are amenable to several kinds of analytical formalization.

PoliS [5] is a coordination model based on multiset rewriting in which coordination rules consume and produce multisets of tuples; rules are expressed in a Chemical Abstract Machine style [3]. In PoliS, properties can be proved by using the PoliS Temporal Logic and the PoliMC model checker.

Tuples centres [14] allow the use of a specification language (named RespecT) to define computations performed in the tuple space. Computations are associated with events triggered internally because of reactions previously fired or during the execution of traditional input/output operations by agents. RespecT is based on first-order logic and unification of unitary clauses (tuple templates) and ground atoms (tuples) represents the basic tuple matching mechanism.

In the ACLT model [6], the tuple space is treated as a container of logic theories, which can be accessed by logic agents to perform deduction processes. Again, the first-order logic and unification of unitary clauses and ground atoms is used as matching mechanism; the model offers specific input-output primitives tailored to provide different meaning for unification by allowing a certain control in selecting the set of unitary clauses to be treated as facts in case of backtracks or temporary missing information in the deduction process.

In our model we do not express coordination in terms of rewriting rules; moreover, the logic layer is enhanced by considering several types of logic languages.

3 Logic and chemical based coordination model

3.1 Definition of the model

The chemical based coordination model we present in this paper is designed to exploit several important features of the models cited above in the context of self-organising and self-adaptive applications; our goal is to define a coordination model with the following characteristics: (i) coordination algorithms can be described in an sufficiently abstract way
starting from high-level specification; (ii) the constructs used to express coordination algorithms are amenable to formal analysis of their correctness, they incentivize the decoupling of logic from implementation and they meet software engineering properties such as modularity, reusability and compactness. The rationale leading the definition of our coordination model can be synthesized as the adoption of Kowalski’s terminology [11]: algorithm = logic + control. This formulation promotes the dichotomy of algorithms in: (i) logic components (formulae), that determine the meaning of the algorithm, the knowledge used to solve a problem (i.e. what has to be done) and (ii) control components, which specify the manner the knowledge is used (i.e. how it has to be done). When the former is clearly specified it becomes more clear what the algorithm does and whether it is correct or not. The latter influences the performance of the algorithm. This means that when the same algorithm can be modeled as $L + C_1$ and $L + C_2$, the formulation with the outperforming control part can be chosen for the implementation.

Figure 1: The generalization of the SAPERE Model

The coordination model we define (Figure 1b) is a generalization of the SAPERE model with two additional features: (i) LSAs can store not only data tuples but actual logic programs (Section 3.2); (ii) the bonding eco-law is replaced by a new one named Logic eco-law, which is in charge of executing logic programs and performing the bonding actions. The remaining components of the model are exactly the same ones of the SAPERE model. The virtual chemical reactions among tuples taking place in the shared container are now driven by logic inferences processes, which produce either data tuples or new logic programs during the
“execution” of logic programs (Figure 1c). This process brings the idea promulgated by [11] in the context of chemical based coordination models: the logic components of an algorithm are expressed in terms of logic programs, here embedded in LSAs, which can react among each others in a chemical fashion. Similarly, agents implement the control components (written in a programming language such as Java), and they perform computations according to the knowledge inferred by logic programs. This approach to separation and mapping of concepts helps designing coordination algorithms from an abstract point of view. On the one hand, algorithms are thought as interactions of atomic logic entities which define the meaning (in Kowalski’s terminology) of subparts of the original algorithm. On the other hand, once logic entities have been defined, a specific problem-solving strategy can be chosen to be implemented for each subpart of the original problem. The intuition of using logic programs is twofold: (i) tuples exchanges represent the basic mechanism to carry out indirect communication among agents, thus the state and the evolution of a coordination process can be defined by analysing the set of tuples in the containers; (ii) tuples are used as inputs (facts) and produced as outputs of logic programs (models and formulae obtained by resolution rules). By considering points (i) and (ii), logic programs provide a natural formal tool to express coordination, allowing for inferred formulae to state relationships among entities of the system, depicting the evolution of coordination processes and proving system properties.

3.2 Logic programs

Logic programs [8] are sets of logic formulae and are expressed in a logic language (e.g. first-order logic). Executing a logic program means either: (i) providing queries to the program and testing whether they logically follow from the program by using a proof engine (logic inference) or (ii) inferring all sentences that logically follow from the program (logic semantics). An interpretation of a formal language is an interpretation (see [13]) of constants, predicate and functions of the language over a given domain. The truth-value of a logic sentence is determined by the interpretation of the logic connectives. Given a logic program P, a model is an interpretation M such that every formula in P is true (depicted as $M \models P$). Here we are interested in Herbrand interpretations ([13]): (i) the implicit domain is the Herbrand Universe, the closure of the set of constants under all the functions symbols of the language; (ii) constants are interpreted as themselves and every function symbol as the function
it refer to. In classical 2-valued logic programs, Herbrand interpretation can be defined through sets of atoms implicitly interpreted as true.

**Example:** \( P = (C(x) \leftarrow A(x), B(x); \ A(c) \leftarrow \square; B(c) \leftarrow \square;) \) is a definite logic program. Clauses are implicitly universally quantified. This is a definite logic program (i.e. containing Horn clauses): \( x \) is a variable, \( c \) is a constant and here they range over an (implicitly) defined domain. The first rule is composed of the head \( C(X) \) and the body \( A(X), B(X) \) and it can be read as “\( C(X) \) is true if both \( A(X) \) and \( B(X) \) are true”. Rules with empty bodies (\( \square \)) are named facts and they state sentences whose heads must be considered satisfied; in this case \( A(c) \) and \( B(c) \) hold. \( M = \{A(c), B(c), C(c)\} \) is a model for the program in the example, because it satisfies all the rules.

### 3.3 Logic languages

In our model, logic programs are executed by the Logic eco-law. An important point in our approach is the generality of the coordination model w.r.t. the logic. We consider only logic languages that supports Herbrand’s interpretations, whereas we do not put any constraint on the inference methods or the semantics. Both inference methods and semantics are treated as parameters associated with logic programs. From the practical point of view, for each logic language we require the implementation of a dedicated Logic eco-law that executes the corresponding logic programs. This feature makes possible to use, possibly simultaneously: (i) several types of logic programs (e.g. definite, general logic programs, several types DATALOG or DATALOG-inspired programs) associated with two-valued, multi-valued (e.g. Belnap’s logic) or paraconsistent logics; (ii) several inference procedures (e.g. SLD, SLDNF) and semantics (e.g. Apt-van Emden-Kowalski, Kripke-Kleen, stable, well-founded model semantics) [10,12,1,8,17].

### 3.4 Logic Fragments

In our model, logic programs are embedded in logic units named Logic Fragments. The following set of definitions will be used to clarify the concept. We assume that Prop, Const and Var are finite mutually disjoint sets of relation symbols, constants and variables respectively. We will identify variables with letters \( x, y, \ldots \) and constants with letters \( a, b, \ldots \).

**Definition 1** (Literals, Ground Literals): A literal \( \hat{P} \) is an expression of type \( P(X_1, \ldots, X_n) \) or \( \neg P(X_1, \ldots, X_n) \) where \( P \in \text{Prop} \) and \( X_i \in (\text{Const} \cup \text{Var}) \) for \( i = 1, \ldots, n \). A ground literal is a literal without
variables. The set of all ground literals w.r.t. a set $\text{Const}$ is denoted $G(\text{Const})$. The power set of $G(\text{Const})$ is depicted $\mathcal{P}(G)$.

**Definition 2** (Valuations): A valuation $w$ is a function from $\text{Var}$ to $\text{Const}$ that assigns a constant $c_i$ to each variable $x_i$. The set of all possible valuations is depicted as $W = \{w|w: \text{Var} \rightarrow \text{Const}\}$.

**Definition 3** (Instances of Literal): If $\hat{P}$ is a literal and $w$ is a valuation, with $\hat{P}_w$ we identify the ground literal where every variable of $\hat{P}$ has been replaced by a constant according to the definition of $w$. $\hat{P}_w$ is named an instance of $\hat{P}$. We denote $I_{\hat{P}} = \{\hat{P}_w|w \in W\} \subseteq G(\text{Const})$.

**Definition 4** (Logic Programs): A logic program is as a set of logic formulae written in a logic language using (i) literals $\hat{P}_1,...,\hat{P}_n$ defined over $\text{Prop}$, $\text{Const}$ and $\text{Var}$ and (ii) logic operators.

**Definition 5** (A-generator): Given a literal $P(X_1,...,X_n)$, an A-generator w.r.t. a function $U: \text{Const}^n \rightarrow \{T,F\}$ is the finite set:

$$P^U(X_1,...,X_n) = \{P(c_1,...,c_n) \in I_{P(X_1,...,X_n)}|U(c_1,...,c_n) = T\}.$$

**Example:** $A^U(X) = \{A(X)|X \in \{a,b,c\}\} = \{A(a),A(b),A(c)\}$, with $U(a) = U(b) = U(c) = T$.

**Definition 6** (I-generator): Given a literal $P(X_1,...,X_n)$, an I-generator w.r.t. a function $V: \mathcal{P}(G) \rightarrow \mathcal{P}(G)$ and a finite set $H \subseteq \mathcal{P}(G)$ is the set:

$$P^{H,V}(X_1,...,X_n) = \{P(c_1,...,c_n) \in I_{P(X_1,...,X_n)} \cap V(H)\}.$$  

If $V$ is omitted, we assume that $V(H) = H$ (identity function).

**Example:** if $N = \{2,3,4\}$ and $V(N) = \{\text{Even}(x)|x \in N \land x \text{ is even}\}$, then $\text{Even}^{N,V}(X) = \{\text{Even}(2),\text{Even}(4)\}$.

The rationale of such definitions is to provide the program with a set of facts built from conditions holding on tuples stored in the container, whose “execution” produces in turn new tuples as result of the interpretation of the logic program on the basis of the facts passed as input.

By $\mathcal{LF}$ we identify the algebraic structure of Logic Fragments, recursively defined as follows:

**Definition 7** (Logic Fragments $\mathcal{LF}$):

(I) $\triangle \in \mathcal{LF}$

$\triangle$ is a special symbol used only in Logic Fragments to depict all the tuples in the container (both LSAs and Logic Fragments).

(II) (Grouping) If $e \in \mathcal{LF}$ then $(e) \in \mathcal{LF}$

(III) (Parallel-and) If $e_1,e_2 \in \mathcal{LF}$ then $e_1 \cap e_2 \in \mathcal{LF}$

(IV) (Parallel-or) If $e_1,e_2 \in \mathcal{LF}$ then $e_1 \cup e_2 \in \mathcal{LF}$

(V) (Composition) If $P$ is a logic program, $M$ an execution modality, $S$ a set of A,I-generators, $\varphi : \mathcal{P}(G) \rightarrow \{T,F\}$ and $e_p \in \mathcal{LF}$ then $(P,M,e_p,S,\varphi) \in \mathcal{LF}$.

$M$ is the identifier of the way $P$ is “executed” (we will use $M = A$ for the
Apt-van Emden-Kowalski and \( M = K \) for the Kripke-Kleen semantics). \( e_P \) is named constituent of the Logic Fragment and it is interpreted as a set of tuples used as support to generate the facts for the program. \( S \) is a set of A and I-generators used to derive new facts from \( P \). The function \( \varphi : \mathcal{P}(G) \to \{T, F\} \) returns \( T \) if the tuples represented by the constituent \( e_p \) satisfy some constraints; the logic program is executed if and only \( \varphi(e_P) = T \) (Def. 8). \( \varphi_T \) is constant and equal to \( T \). For style reasons, we will write \( P^M(e_P, S, \varphi) \) instead of \( (P, M, e_P, S, \varphi) \).

Every Logic Fragment is executed by the Logic eco-law; its semantics is defined by using the function \( v_L \).

**Definition 8** (Semantic function): \( v_L : \mathcal{LF} \to \mathcal{P}(C) \cup \{\boxtimes\} \) associates the fragment with the set of tuples inferred by the logic program (consequent) or with \( \boxtimes \), which stands for undefined interpretation. \( L \) denotes the set of tuples in the container before executing a Logic Fragment. Operators are ordered w.r.t. these priorities: grouping (highest priority), composition, \( \cap \) and \( \cup \) (lowest priority). \( v_L \) is recursively defined as follows:

I) \( v_L(\triangle) \triangleq L \)

II) \( v_L(\{\epsilon\}) \triangleq v_L(\epsilon) \)

III) \( v_L(e_1 \cap \ldots \cap e_n)_{n \geq 2} \triangleq \begin{cases} \boxtimes & \text{if } \exists i \in \{1, \ldots, n\}. v_L(e_i) = \boxtimes \\ \bigcup_{i=1}^n v_L(e_i) & \text{otherwise} \end{cases} \)

IV) \( v_L(e_1 \cup \ldots \cup e_n)_{n \geq 2} \triangleq \begin{cases} \bigcup_{i \in \mathcal{I}} v_L(e_i) & \text{if } \mathcal{I} = \{e_i | v_L(e_i) \neq \boxtimes, 0 \leq i \leq n\} \neq \emptyset \\ \boxtimes & \text{otherwise} \end{cases} \)

V) \( v_L(P^M(e_P, S, \varphi)) \triangleq Q \)

\( Q \) is the consequent of \( P^M \) and it is defined as follows: if \( M \) is not compatible with the logic program \( P \) or if \( v_L(e_p) = \boxtimes \) or if \( \varphi(v_L(e_p)) = F \) then \( Q = \boxtimes \). \( \varphi \) “blocks” the execution of the program as long as a certain condition over \( e_p \) is not satisfied. Otherwise, based on \( S = \{P^0_{H_0, V_0}(X_{01}, \ldots, X_{0n}), \ldots, P^n_{H_n, V_n}(X_{n1}, \ldots, X_{nt}), P_0(Y_{01}, \ldots, Y_{0z0}), \ldots, P_m(Y_{m1}, \ldots, Y_{mz0})\} \), the Logic eco-law produces the set of facts \( F_S = \bigcup_{i=0}^n P^i_{V_i(H_i), V_i}(X_{i1}, \ldots, X_{ii}) \cup \bigcup_{i=0}^m P_i(Y_{i1}, \ldots, Y_{izi}) \). A,I-generators are then used to define sets of ground literals for the logic program which satisfy specific constraints; moreover, for every set \( H_i \) we have either \( H_i = e_p \) or \( H_i = \triangle \). \( Q \) is finally defined as the set of atoms inferred by the applying \( M \) on the new logic program \( P^M = P^M \cup \{l \leftarrow \square | l \in F_S\} \), enriched by all the facts contained in \( F_S \). Note that there may be no need to explicitly calculate all the literals of I,A-generators beforehand: the membership of literals to generators sets may be tested one literal at a time or skipped because of the short-circuit evaluation.
Lemma 1 (Properties of operators): Given \( a, b \in \mathcal{L} \mathcal{F} \) with \( a \equiv b \) we state that \( v_L(a) = v_L(b) \) for every set of literals \( L \). Then for any \( a, b, c \in \mathcal{L} \mathcal{F} \):

I) \[ a \sqcup a \equiv a \] (Idempotence of \( \sqcup \))

II) \[ a \sqcup b \equiv b \sqcup a \] (Commutativity of \( \sqcup \))

III) \[ a \sqcup (b \sqcup c) \equiv (a \sqcup b) \sqcup c \] (Associativity of \( \sqcup \))

IV) \[ a \sqcap a \equiv a \] (Idempotence of \( \sqcap \))

V) \[ a \sqcap b \equiv b \sqcap a \] (Commutativity of \( \sqcap \))

VI) \[ a \sqcap (b \sqcap c) \equiv (a \sqcap b) \sqcap c \] (Associativity of \( \sqcap \))

VII) \[ a \sqcap (b \sqcup c) \equiv (a \sqcap b) \sqcup (a \sqcap c) \equiv (b \sqcup c) \sqcap a \] (Distrib. of \( \sqcap \) over \( \sqcup \))

Intuitively, composing two Logic Fragments means calculating the inner one first and considering it as constituent for the computation of the second one. Parallel-and (\( \sqcap \)) means executing all the Logic Fragments them in a row, whereas Parallel-or (\( \sqcup \)) means executing only those ones that can be executed at a given time.

For example if \( e = P^3_{M3}(t, S_3) \), with \( t = P^1_{M1}(e_1, S_1) \sqcup P^2_{M2}(e_2, S_2) \), it may be evaluated as \( P^3_{M3}(P^1_{M1}(e_1, S_1), S_3) \), \( P^3_{M3}(P^2_{M2}(e_2, S_2), S_3) \) or \( P^3_{M3}(t, S_3) \), whereas in \( e = P^3_{M3}(P^1_{M1}(e_1, S_1) \sqcap P^2_{M2}(e_2, S_2), S_3) \), we need to first evaluate both logic programs \( P^1_{M1} \) and \( P^2_{M2} \).

3.5 Update of the container

In our model, all the Logic Fragments are carried on a snapshot image of the container, i.e. given a Logic Fragment \( e \) in the container, if \( v_L(e) \neq \emptyset \), then it is evaluated as an atomic operation (every symbol \( \Delta \) in the sub Logic Fragments which composes \( e \) is always translated with the same set of tuples). Multiple Logic Fragments ready to be evaluated are computed in a non-deterministic order. The tuples inferred by the logic programs are inserted in the container only when the evaluation of the whole logic terminates. At that point, the Logic eco-law injects the inferred tuples in the container and notifies the end of inference process to the agent. The Logic Fragment is subject to a new evaluation process as soon as the set \( F \) changes due to updates of the shared container, but there are no concurrent parallel evaluations of the same Logic Fragment at a given time (unless it appears twice); this aspect can potentially hide tuples updates in the evaluation process (Section 5). The representation of the functions associated with \( A,I \) generators depend on the implementation.

4 Case studies

By using Logic Fragments we can easily tackle interesting coordination problems and properties.
4.1 Palindrome recognition

As a first example we show an easy scenario. Assuming that an agent $A$ inserts integers into the container, we want to discover which ones of the injected integers are palindromic numbers (i.e. numbers that can be read in the same way from left to right and from right to left) greater or equal to 1. Let’s consider that these integers are represented by tuples of type $N(a)$, where $a$ is a number, e.g. $N(3)$ represents the number 3. Agent $A$ inserts the following logic fragment:

\[
LF_p : P^A_p(\triangle, \{N^{\triangle}, TestPalin\}, \varphi_p) \\
\varphi_p(\triangle) = T \text{ iff } \exists N(X)_{\triangle} \in \triangle \\
TestPalin(x) = \{TestPalin(a)|1 \leq a \leq d_{max}, a \text{ is a palindromic number}\}
\]

Code 1.1: Definite logic program $P_p$

$Palin(x) \leftarrow N(x), TestPalin(x)$

Figure 2: Evolution of the container for the example of Section 4.1

$P_p$ is the logic program in Code 1.1 and it is evaluated with the Apt-van Embden Kowalski semantics. The set $S$ of $A$- and $I$- generators is composed of two elements, evaluated in this way: $N^{\triangle}$ contains all literals $N(a)$ (numbers) existing in the container ($\triangle$); $TestPalin(x)$ contains all the literals of type $TestPalin(a)$, where $a$ is a positive palindromic number less then a certain limit $d_{max}$. These two sets of literals are treated as facts for the program $P_p$. Finally, $\varphi$ is the function that controls the
execution of $P_p$: if at least a number $N(a)$ is inserted into the container, the program is executed. The rule of the logic program $P_p$ states that a number $a$ is a palindromic number ($\text{Palin}(a)$) if $a$ is a number ($N(a)$) and $a$ passes the test for being palindromic ($\text{TestPalin}(a)$).

To show what happens when some numbers are injected let’s consider the tuple space shown in Figure 2a, 2b and 2c. At the beginning, agent $A$ injects the logic fragment (Figure 2a). Later, $A$ injects the number $N(22)$, the logic fragment is then executed. In this case, $N\triangle$ is evaluated as $\{N(22)\}$. Moreover, $\text{TestPalin}(a)$ will contain $\text{TestPalin}(22)$, because it is palindromic. This means that the rule of the logic program is satisfied and the consequent $Q$ (Figure 2b) will contain $\text{Palin}(22)$ (along with all the facts generated by the A-I generators). If now agent $A$ injects a second number $N(12)$, the logic fragment is re-executed and $N\triangle$ is evaluated as $\{N(22), N(12)\}$. This time the number does not satisfy the palindromic test ($N(12) \notin \text{TestPalin}(x)$), so the 12 will not be considered as palindromic. Finally, if $A$ injects $N(414)$, then during the re-execution of $P_p$, $N\triangle = \{N(22), N(12), N(414)\}$ and $N(414) \in \text{TestPalin}(x)$, so we will obtain that the consequent changes, including elements $\{\text{Palin}(22), \text{Palin}(414)\}$ (Figure 3). Note that if numbers were injected by agents different from $A$ (like the sensor), the same reactions would take place in the container.

Let’s now assume that instead of finding out general palindromic numbers, $A$ wants to find all palindromic numbers composed of an even number of decimal digits. The new logic fragment $LF_{pe}$ injected by $A$ is:

$$LF_{pe} : P_e^A(c, \{\text{TestEvenDigits}\}, \varphi_e) \quad c = P_p^A(\Delta, \{N\triangle, \text{TestPalin}\}, \varphi_p)$$
$$\varphi_e(c) = T \iff |c| > 0$$
$$\text{TestEvenDigits}(x) = \{\text{TestEvenDigits}(a) \mid \log_{10}(a) \equiv 0 \pmod{2}, 1 \leq a \leq d_{max}\}$$

Code 1.2: Definite logic program $P_e$

$$\text{EvenDigitPalin}(x) \leftarrow \text{Palin}(x), \text{TestEvenDigits}(x)$$

$c$ is the consequent of the logic fragment $LF_{pe}$, now treated as constituent for the program $P_e^A$; the latter is now used to test if the palindromic integer is composed of an even number of decimal digits. Its constituent is the semantics of $P_p'$; this means that all the literals belonging to the semantics $c P_p^{A'}(\Delta, \{N\triangle, \text{TestPalin}\}, \varphi_p)$ are then considered as facts for the program $P_e^A$, whose semantics is calculated after obtaining the Herbrand model for $P_p'$. Let’s assume now that $A$ injects
numbers $N(22)$ and $N(12)$. As explained before, the model $c$ for $P^A_p$ will contain $Palin(22)$. Given that 22 is a number composed of an even number of decimal digits, we will obtain that $TestEvenDigitPalin(22) \in TestEvenDigitPalin(x)$, then $EvenDigitPalin(22)$ will be contained in the Herbrand model of $P^A_e$. In contrast, if $N(414)$ is injected by $A$, $Palin(414)$ will be contained in the semantics of the Logic Fragment $c$, but $TestEvenDigits(414) \notin TestEvenDigits(x)$, so $EvenDigitPalin(414)$ will not be inferred as literals in the semantics of $P'_e$ (Figure 3).

**Property 1:** Assuming $LFe$ and the behavior of agent $A$ defined above, a palindromic positive integer $a$ composed of an even number of decimal digits exists in the shared space if and only if $EvenDigitPalin(a)$ is contained in the least Herbrand model of $P'_e$.

The property above states that by using the logic fragment $LFe$ we are able to correctly find out all the palindromic integers with even number of decimal digits. The proof is simple: thanks to the logic programs and the semantic of logic fragments, we can easily verify that if such integers exist in the shared space then their literals will be inferred in the model of $P'_e$. Moreover, given that in this scenario the literals associated with palindromic numbers are only generated from the logic fragments, if such literals exist in the model of $P_e$ then there must be palindromic integers with even number of decimal digits in the shared space.
4.2 Pattern recognition - definite logic programs

We now imagine a scenario composed of three agents: \( A_g, A_b \) and \( A_p \). Agents \( A_g \) and \( A_b \) move on the finite map shown in Figure 4a and they detect respectively single grey (\( A_g \)) and black (\( A_b \)) cells. When a grey cell is discovered in position \((a, b)\), \( A_g \) inserts a tuple \( G(a, b) \) in the container (and so does \( A_b \) respect to \( B(a, b) \)), as reported in Code 1.3 and 1.4. The goal of agent \( A_p \) is to discover, on the same map, all the grey and black patterns reported in Figures 4b and 4c.

![Figure 4: Maps and patterns of the example of Section 4.2](image)

To find all grey-black patterns, \( A_p \) inserts the logic fragment

\[
LF_p : P_g^A(\triangle, S_g, \varphi_g) \cap P_b^A(\triangle, S_b, \varphi_b)
\]

where

\[
S_g = \{G^{\triangle}, G^{\triangle,gr_4}, G^{\triangle,gr_5}, G^{\triangle,gr_7}, G^{\triangle,gr_8}\}
\]

\[
S_b = \{B^{\triangle}, B^{\triangle,b_{r_1}}, B^{\triangle,b_{r_3}}, B^{\triangle,b_{r_4}}, B^{\triangle,b_{r_8}}\}
\]

\[
\varphi_g(\triangle) = T \iff \exists w : G(x, y)_w \in \triangle
\]

\[
\varphi_b(\triangle) = T \iff \exists w : B(x, y)_w \in \triangle
\]
The logic fragment is composed of two logic programs, $P_g$ and $P_b$ (Code 1.5 and 1.6), composed with the parallel-and operator. This means that both of them must terminate before the agent $A_p$ receives the consequent of the logic fragment, i.e. at least one black and one grey pattern must be found. According to $\varphi_g$ and $\varphi_b$, the logic programs are executed every time a black cell or a grey cell is found on the map, i.e. as soon as the corresponding tuple is inserted in the container.

To define the identified patterns, we use $G_1(x,y)$ and $G_2(x,y)$ to state that grey patterns have been found w.r.t. coordinates $x, y$. Similarly, we use $B_1(x,y)$ and $B_2(x,y)$ to depict black patterns. The coordinates $x,y$ of the literals above represents the coordinates of a reference cell contained in the patterns, as showed in the Figures 4b and 4c.

$S_b$ and $S_g$ are sets of I-generators, used to generate facts for the logic programs $P_g$ and $P_b$. According to their definitions (6), $B^\triangle$ and $G^\triangle$ contain respectively all the black and grey cells found in the container. The remaining generators are used to define the topological structure of black and grey patterns, as explained with the following example.

The logic program works as follows: we assume that we found all black cells. If a cell $(a,b)$ is black, we suppose that it is the reference cell for a black pattern, for example the one on the bottom of Figure 4c. If this particular black pattern really exists, then other two black cells must exist: one with coordinates $(a-1,b)$ and the second one on $(a-1,b-1)$.
This means that \( G(a - 1, b) \) and \( G(a - 1, b) \) are in the container, because we assumed all black cells were found. Then if we calculate the sets \( B_{r_1}^{\Delta, b_1}(x, y) \) and \( B_{r_8}^{\Delta, b_8}(x, y) \), they contain (respectively) \( B_{r_1}(a, b) \) and \( B_{r_8}(a, b) \), i.e. the first rule of the program \( P_b \) is satisfied and \( B_1(a, b) \) is correctly inferred. The I-atoms use vectors \( r_1, \ldots, r_8 \) depicted in Figure 4e to define the position of the other cells that must be found close to the reference cell. Numerically if we imagine for example that \( A_g \) has inserted \( B(1, 2), B(1, 3) \) and \( B(2, 3) \), then we obtain that they are all contained in \( B^{\Delta}(x, y) \). Moreover, after computing the I-generators, we obtain also that \( B_{r_1}(2, 3) \in B_{r_1}^{\Delta, b_1}(x, y), B_{r_8}(2, 3) \in B_{r_1}^{\Delta, b_1}(x, y) \). According to Definition 8, all these three values will be treated as facts for the program \( P_b \). This means that when the semantic function \( v_M \) calculates the least Herbrand model of \( P_b' \), it will infer that \( B_1(2, 3) \) is contained in it, i.e. the cell \( (2, 3) \) is the reference cell for a found black pattern. Similarly, after inserting the position of all grey cells, in the least Herbrand model of \( P_b' \) we automatically discover all patterns \( A \triangleq B_1(2, 3), B \triangleq B_1(4, 2), C \triangleq B_2(3, 6) \), depicted in Figure 4d. Globally, we obtain that \( D \triangleq G_1(2, 6), E \triangleq G_2(3, 3) \) and \( F \triangleq G_2(5, 2) \) are in the least Herbrand model of \( P_b' \). Analogous considerations hold for grey patterns. If not patterns are found the consequent of \( LF_p \) is the empty set.

**Property 2:** Assuming \( LF_p \) and the behavior of agent \( A_g \) defined above, once all grey cells are discovered, a grey pattern exists in the map w.r.t. coordinates \((a, b)\) if and only if its corresponding literal \( G_1(a, b) \) (or \( G_2(a, b) \)) is contained in the least Herbrand model of \( P_g' \).

An analog statement holds for black cells and patterns. These lemmas prove the correctness of the algorithm and they are derived as follows: the concept of pattern is translated, by using the logic formalisation, in terms of properties \( G_1(x, y), G_2(x, y), B_1(x, y), B_2(x, y) \). Each one of these statement is obtained in the consequent only if it has been derived by a rule of the logic program. This means that, topologically, a pattern has been found in the map. Similarly, each rule of the program topologically defines a pattern. This means that if all the cells of the patterns have been found and marked by \( A_b \) or \( A_g \), the rule is satisfied and the head of the rules appears in the consequent.

We consider now a new scenario where an additional agent \( A_z \) wants to find all the patterns reported in Figure 5a on the same map. This pattern can be constructed from the two patterns in Figure 5b. We can reuse the previous logic fragment \( LF_p \) in order to get access to the literals inferred for programs \( P_b' \) and \( P_g' \). The logic fragment inserted by \( A_z \) is then: \( LF_z : P_z^A( P_m^A( P_g^A(\Delta, S_g, \varphi_g)) \cap P_b^A(\Delta, S_b, \varphi_b), S_m, \varphi_T), S_z, \varphi_T) \).
\[ S_m = \{ G_1^\Delta, G_2^\Delta, B_1^\Delta, B_2^\Delta, C_G \} \]
\[ S_z = \{ M_1^\Delta, M_2^\Delta, D_G \} \]
\[ C_G(x_1, x_2, y_1, y_2) = \{ C(a, b, c, d) | b = a + 1, c = d \} \]
\[ H_G(x_1, x_2, y_1, y_2) = \{ H(a, b, c, d) | a = b, d = c + 3 \} \]

Code 1.7: Program \( P_m \)
\[ M_1(x_1, y_1) \leftarrow G_1(x_1, y_1), B_2(x_2, y_2), C(x_1, x_2, y_1, y_2) \]
\[ M_2(x_1, y_1) \leftarrow B_1(x_1, y_1), G_2(x_2, y_2), C(x_1, x_2, y_1, y_2) \]

Code 1.8: Program \( P_z \)
\[ Z(x_1, y_1) \leftarrow M_1(x_1, y_1), M_2(x_2, y_2), H(x_1, x_2, y_1, y_2) \]

\( M_1(x, y) \) and \( M_2(x, y) \) state that \( M \) patterns have been found w.r.t. coordinates \( x, y \) as shown in Figure 5b: they are used to bind together one gray pattern with a black one. Finally, the A-generator \( C_G \) states what is the relation among the coordinates of the gray-black patterns in order to obtain a proper \( M \) pattern. All ground literals of type \( M_1(x, y) \) and \( M_2(x, y) \) inferred in the Herbrand model of \( P_m' \) are then used as constituent for the program \( P_g' \). This program is composed of a single rule, which identifies which \( M \) patterns can be bound together to create a \( Z \) pattern (the use of the A-generator \( H_G \) is similar to \( C_G \)). This time (Figure 5c), after inserting in the container the position of all grey and black cells, we will obtain that \( M \triangleq M_2(4, 2), M_2(2, 3) \) and \( M_1(2, 6) \) are contained in the Herbrand model of \( P_m' \), whereas \( Z \triangleq Z(2, 3) \) is the only pattern existing.
in the consequent of the logic fragment $LF_z$. In this case we can state that:

**Lemma 2:** Assuming $LF_z$ and the behaviors of agents $A_b$ and $A_g$ defined above, once all grey-black cells are discovered, a $Z$ pattern exists with coordinate $a, b$ in the map if and only if $Z(a, b)$ is contained in the Herbrand model $P'_z$.

### 4.3 Gradient and Chemotaxis patterns - general programs

In this last example we use Logic Fragments to implement the gradient and chemotaxis design patterns ([9]), which are two bio-inspired mechanisms used to build and follow shortest paths among nodes in a network. The chemotaxis is based on the gradient pattern. A gradient is a message spread from one source to all the nodes in the network, carrying a notion of distance from the source (hop-counter). Gradient messages can also carry user-information. Once a node receives a gradient from a known source whose hop-counter is less than the local one (i.e. a new local shortest-path has been found), the node updates its local copy of the hop-counter (aggregation) and spreads it towards the remaining neighbours with a hop-counter incremented by one unit. The chemotaxis pattern resorts to gradient shortest-paths to route messages towards the source of the gradient. We can implement the gradient and chemotaxis patterns by using an agent $A_{gc}$ associated with the Logic Fragment:

$$LF_{gc} : P^A_g \left( P^A_a(\triangle \sqcap P^K_n(\triangle, S_n, \varphi_n), S_a, \varphi_T), S_g, \varphi_T \right) \sqcup P^A_{\text{ch}}(\triangle, S_{\text{ch}}, \varphi_{\text{ch}})$$

Values null, $d_{\text{max}}$ and local (replaced with the name of the node in which this keyword is used) are considered constants.

**Code 1.9:** Program $P_n$ - Next hop initialization

```
GPath(x,d_{\text{max}},null) \leftarrow -\text{existsGPath}(x)
```

**Code 1.10:** Program $P_a$ - Aggregation

```
cmpGradient(x_1, x_2, y_1, y_2, z) \leftarrow \text{Gmsg}(x_1, x_2, y_1, z), \text{GPath}(x_1, y_2, w)
updateGPath(x_1, y_1, x_2, z) \leftarrow \text{cmpGradient}(x_1, x_2, y_1, y_2, z), \text{less}(y_1, y_2)
```

**Code 1.11:** Program $P_g$ - Spreading

```
\text{spreadGradient}(x_1, \text{local}, y, z, x_2) \leftarrow \text{updateGPath}(x_1, z, x_2, y)
```

**Code 1.12:** Program $P_{\text{ch}}$ - Chemotaxis

```
\text{sendChemo}(x, y, z) \leftarrow \text{Cmsg}(x, y), \text{GPath}(y, w, z)
```
\[ \varphi_n(\Delta) = T \iff \exists w : Gmsg(x_1, x_2, y, z)_w \in \Delta, \quad \varphi_{ch}(\Delta) = T \iff \exists w : Cmsg(x, y)_w \in \Delta \]

\[ S_{ch} = \{ Cmsg^\Delta, GPath^\Delta \} \quad S_g = \{ updateGPath^{e^g} \} \quad S_n = \{ existsGPath^\Delta, V, Gmsg^\Delta \} \]

\[ S_a = \{ existsGPath^\Delta, V \} = \{ \neg existsGPath^\Delta \} \]

\[ V(\Delta) = \{ \neg existsGPath(a) \} \]

Gradients are represented by tuples \( Gmsg(a, b, c, d) \) where \( a \) is the ID of the source, \( b \) is the ID of the last sender of the gradient, \( c \) is the hop-counter and \( d \) is the content of the message. Local hop-counter are stored in tuples of type \( GPath(a, c, e) \), where \( a \) and \( c \) are as above and \( e \) is the previous node in the path, used to route the chemotaxis message downhill towards the source. \( LF_{gc} \) is composed of several Logic Fragments; the parallel-or operator makes the agent \( A_{gc} \) to react simultaneously to chemotaxis and gradients messages. The innermost fragment \( e_{Pa} = \Delta \cap P_n^c(\Delta, S_n, \varphi_n) \) is executed when a gradient message is received from a neighbour (\( \Delta \) can always be executed directly but the parallel-and operator blocks the execution of outer fragments until \( P_n^c(\Delta, S_n, \varphi_n) \) finishes); it initializes the \( GPath \) tuple for the source of the gradient. By using the composition operator, the literals inferred in the model of \( P_n^c \) along with all the tuples in the container (fragment \( \Delta \)) are then treated as constituent for the fragment \( e_{Pa} = P_n^c(\Delta, S_n, \varphi_n) \), i.e. they are used to generate facts for the program \( P_n' \). This one is used to aggregate the hop-counter for the source with the one stored in the local container. \( e_{Pa} \) is finally treated as constituent for the fragment \( e_{Pg} = P_g^c(\Delta, S_g, \varphi_T) \). \( P_g^c \) is used to verify whether the gradient message must be spread to the neighbours. If so, a literal \( spreadGradient \) is inferred during the computation of its semantics. Simultaneously, the Logic Fragment \( P_{ch}^c(\Delta, S_{ch}, \varphi_{ch}) \) is executed as soon as a chemotaxis message is received (described as \( Cmsg(f, g) \), with \( f \) content of the message and \( g \) ID of the receiver). That Logic Fragment uses the local copy of the hop-counter to infer which is the next hop to which the chemotaxis message must be sent to (relay node). If the local hop-counter exists, a literal \( sendChemo(f, g, h) \) is generated in the model of \( P_{ch}' \), with \( h \) representing the ID of the next receiver of the chemotaxis message. Otherwise, the message remains in the container as long as such a literal finally inferred. All the literals contained in the consequent \( Q \) of \( LF_{gc} \) are used by the agent \( A_{gc} \) to manage the control part of the algorithm, described in the following code.

Code 1.13: Behaviour of agent \( A_{gc} \)
\[
\text{if } \text{spreadGradient}(a, \text{local}, d, c, b) \in Q \text{ then }
\begin{align*}
\quad & \text{send } Gmsg(a, \text{local}, c + 1, d) \text{ to all neighbours but } b \\
\quad & \text{remove } \text{container.Gra}(a, x, y, z)_w \text{ for any } w
\end{align*}
\]
\[
\text{if } \text{updateGPath}(a, c, b, d) \in Q \text{ then }
\begin{align*}
\quad & \text{update } \text{container.GPath}(a, x, y)_w = \text{GPath}(a, c, b) \text{ for any } w
\end{align*}
\]
\[
\text{if } \text{sendChemo}(f, g, h) \in Q \text{ then }
\begin{align*}
\quad & \text{send } Cmsg(f, g) \text{ to node } h \\
\quad & \text{remove } \text{container.Cmsg}(f, g)
\end{align*}
\]

We consider now the network of Figure 6; the Logic Fragment can be used to provide the gradient and chemotaxis functionalities as services to other agents running on the same nodes. Considering that agent \(\text{AGm}\) on node \(A\) wants to send a query message \(m_1\) to all the nodes of the network, it creates and injects the gradient message \(Gmsg(A, A, 0, m_1)\). At this point a reaction with \(\text{LF}_{gc}\) takes place, generating in the consequent \(Q\) of \(\text{LF}_{gc}\) literals \(\text{GPath}(A, 0, A)\) (semantics of \(P'_n\)) and \(\text{spreadGradient}(A, A, m_1, 0, A)\) (semantics of \(P'_g\)). The second literal causes the spreading of the gradient message to nodes \(B\) and \(C\). Similar reactions take place in the remaining nodes. If we assume that the gradient passed by node \(D\) is the first one to reach \(E\), \(\text{GPath}(A, 3, D)\) is then inferred in the consequent \(Q\) on node \(E\). When the gradient message coming from \(B\) reaches \(E\), \(\text{updateGPath}(A, 1, B, m_1)\) is inferred in the semantics of program \(P'_a\), so the hop-counter tuple is updated in \(\text{GPath}(A, 2, B)\). Assume that agent \(\text{ACm}\) on node \(E\) wants to send a reply-message \(m_2\) to node \(A\), it creates and injects a chemotaxis message \(Cmsg(m_2, A)\). On the basis of the tuple \(\text{GPath}(A, 2, C)\), the literal \(\text{sendChemo}(m_2, A, C)\) will be inferred in the semantics of \(P'_g\), so the message will be sent to node \(B\). Similar reactions take place on node \(B\), which will finally send the chemotaxis message to node \(A\).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{network.png}
\caption{Network of 5 nodes}
\end{figure}
Property 3: Let \( N \) be a network with no disconnected hosts. We assume that: (I) nodes do not move; (II) every node has a Logic Fragment of type \( LF_{gc} \); (III) every information sent from one node to another one arrives at destination in a finite time (eventually due to multiple spreading); (IV) a gradient message is created by an agent \( AG_m \) on one node \( S \) of \( N \). Then there exists a finite time \( t^* \) for which the following statement holds: if an agent \( AC_m \) on node \( R \) creates a chemotaxis message for \( A \) at time \( t > t^* \), then the chemotaxis message reaches the destination \( S \) following a shortest-path between \( R \) and \( S \).

Before showing the details of the proof, it should be noticed that the gradient algorithm we presented can be considered a special version of the distance-vector algorithm.

Proof: The property is proven resorting to Lemma 7. We need the following definitions.

Definition 9: Let \( g \) be a gradient message and \( n_1 \) and \( n_2 \) two nodes of the network. With \( n_1 \to_g n_2 \) we identify the event of sending \( g \) from node \( n_1 \) to node \( n_2 \).

Definition 10: Let \( g \) be a gradient message. With \( \text{Hop}(g) \) we identify the hop-counter associated with \( g \).

Definition 11: Let \( g \) be a gradient message. A path \( \pi_g = (n_0 = S,...,n_k) \) of length \( k \) is an ordered list of \( n_0,...,n_k \) nodes such that \( S \) is the source of the gradient and \( n_i \to_g n_{i+1} \forall i \in \{1,...,k-1\} \). Two paths \( \pi_1 \) and \( \pi_2 \) are different if there exists at least one index \( i \in \{0,...,k\} \) such that the node of \( \pi_1 \) in position \( i \) and the one of \( \pi_2 \) at the same position \( i \) are different.

Lemma 3 (No loops): Let \( g \) be a gradient message managed by the Logic Fragment \( LF_{gc} \) and \( \pi_g \) a path of \( k \geq 1 \) nodes. Every node of the path appears only once in the first \( k \) positions, i.e. all nodes \( n_0,...,n_{k-1} \) are distinct.

Proof: We prove the statement by contradiction. Let \( S \) be the source of the gradient \( g \). Let \( n_s \) be last-sender of \( g \) the first time that a literal \( Gmsg \) is created in the container of \( n \) and let \( a = \text{Hop}(g) \) be the hop-counter of \( g \). Given that \( n \) spreads \( g \), there must be a literal \( \text{updateGPath} \) for the source of \( g \) in the consequent of \( LF_{gc} \). This means that the tuple \( GPath(S,a,n_s) \) is created in the container. Before spreading, \( \text{Hop}(g) \) is incremented by one. This means that the second time \( g \) passes by \( n \) it has \( \text{Hop}(g) > a \) and the logic program \( P_a \) does not infer any literal of type \( \text{updateGPath} \) for the source \( S \), so \( g \) is discarded and it is not re-spread. We then obtain a contradiction.
Lemma 3 states that every path of a gradient of length $k \geq 1$ contains no loops in the first $k$ positions. Given that the size of the network is fixed we can now prove the following statement.

**Lemma 4** (Maximum path length): Let $g$ be a gradient message managed by the Logic Fragment $LF_{gc}$ and $m$ the number of nodes in the network. The maximum length of a path $\pi_g$ is $m$. If $\pi_g$ does not contain loops then its maximum length is $m - 1$.

**Proof:** The number of nodes in the network is $m$ and the statement of Lemma 3 holds.

**Lemma 5** (Local property): Let $g$ be a gradient. Any node of the network spreads $g$ only for a finite number of times.

**Proof:** Let $\pi = \{\pi_g_1, ..., \pi_g_k\}$ the set of all the paths for $g$ without loops. Given that the number of nodes in the network is finite and the statement of Lemma 4 holds, the cardinality of the set $\pi$ is finite. This means that every node spreads the gradient a finite number of times.

Given that the number of times that a gradient $g$ is spread in the network is finite, we can define a temporal instant $t^*$ as the end of computation of the last Logic Fragment $LF_{gc}$ executed on any of the nodes of the network. Moreover $t^*$ is finite because of condition (III) of Property 3.

**Lemma 6** (Global property I - Shortest-path): Let $t^*$ be the instant when a Logic Fragment $LF_{gc}$ is executed after receiving the last gradient message $g$ existing in the network. In every node $n_k$ of the network exists a literal of type $GPath(S, k, n_{k-1})$, where $S$ is the source of $g$, $k$ is the distance of $n_k$ from $S$ and $n_{k-1}$ is a neighbour of $n_k$ belonging to a shortest path $\pi = (n_0 = S, ..., n_{k-1}, n_k = n)$.

**Proof:** We prove this statement by induction on the distance $d$ of the nodes from $S$.

**Base $d=1$:** Let $n_1$ be a neighbour node of $S$. The condition of $t^*$ assures that every message for $g$ spread by a node has been received by the receiver node; this means that $n_1$ has received the gradient message $g$ from $S$, with $Hop(g) = 1$. Then considering $P_n$ and $P_a$, a literal $GPath(S, 1, S)$ has been inferred in the semantics of $LF_{gc}$, eventually aggregating the local hop-counter if $n_1$ previously received a gradient message $g$ with $Hop(g) > 1$ (second rule of logic program $P_a$). Moreover, $P_a$ preserves the minimum hop-counter, so the literal $GPath(S, 1, S)$ is never updated. Clearly, $\pi_g = (n_0 = S, n_1)$ represents the shortest-path between $S$ and $n_1$.

**Step:** We assume the statement holds for all nodes $n_k$ at distance $k$.

Let $n_{k+1}$ be a node at distance $k + 1$ and $N_k$ the set of all nodes at
distance $k$. Then in every $n_k \in N_k$ a literal of type $GPath(S, k, n_{k-1})$ has been inferred and a gradient message $g$ with $Hop(g) = k + 1$ has been sent because of program $P_g$. An important property of shortest-paths states that every sub-path of a shortest-path is in turn a shortest-path. Then for the induction hypothesis any node of $N_k$ represents a candidate for a shortest-path between $S$ and $n_{k+1}$, because the latter is at the minimum distance (1 hop) from all the nodes of $N_k$, which are in the shortest-path towards $S$. The condition of $t^*$ assures that all the messages of type $Gmsg$ sent by the nodes of $N_k$ has been received by $n_{k+1}$. Considering again $P_n$ and $P_a$, a literal of type $GPath(S, k + 1, n_k)$ is then inferred for at least one $n_k \in N_k$, eventually updating the local hop-counter if a gradient message $g$ with $Hop(g) > k + 1$ was previously received by $n_{k+1}$ (second rule of logic program $P_a$). Also in this case, $P_a$ preserves the minimum value for the hop-counter, so the tuple $GPath$ for $S$ is never updated at a later stage. Summarising, the statement holds for any node at distance $k + 1$.

Lemma 7 (Global property 2 - Chemotaxis): Let $t^*$ be the instant when a Logic Fragment $LF_{gc}$ is executed after receiving the last gradient message $g$ existing in the network. Assuming that the source of $g$ is $S$, if at instant $t > t^*$ a node $n_k$ at distance $k$ from $S$ receives a chemotaxis message for the source, the message is re-sent by $n_k$ and it reaches the destination following a shortest-path $\pi = (n_0 = S, \ldots, n_{k-1}, n_k = n)$.

**Proof:** We prove this statement by induction on the distance $d$ of the nodes from $S$.

**Base $d=1$:** Let $n_1$ be a neighbour node of $S$. By considering that $t > t^*$ and by resorting to Lemma 6, we can state that a literal of type $GPath(S, 1, S)$ exists in the shared container of $n_1$. If now a chemotaxis message $Cmsg(m, S)$ appears in the shared container, a literal of type $sendChemo(m, S, S)$ is inferred by $P_g$, so the chemotaxis message is sent to $S$. Given the hypothesis (III) of Property 3, the message will be received by $S$. Clearly, sending a message from $n_1$ to $S$ means following the unique shortest-path between $S$ and $n_1$.

**Step:** We assume the statement holds for all nodes $n_k$ at distance $k$.

Let $n_{k+1}$ be a node at distance $k + 1$. By considering that $t > t^*$ and by resorting to Lemma 6, we can state that a literal of type $GPath(S, k, n_{k-1})$ exists in the shared container of $n_k$. By proceeding as we did in the base, we can prove that if a chemotaxis message $Cmsg(m, S)$ appears in the shared container of $n_{k+1}$, then the message will be sent to a node $n_k$, following the shortest-path between $n_{k+1}$ and $n_k$ (one hop path). Moreover, the message will be finally received by $n_k$. For the induction hypothesis,
the message will then arrive to the destination \( S \) following a shortest path between \( S \) and \( n_k \), so globally the chemotaxis message has reached the destination following a shortest-path between \( S \) and \( n_{k+1} \).

5 Conclusion and Future works

In this paper we have presented a chemical based coordination model based on a logic framework. Virtual chemical reactions are lead by logic deductions, implemented in terms of combination of logic programs. This approach combines the benefits of using a chemical based coordination model along with the expressiveness of several distinct types of logic languages to formalise coordination logically. Intuitively, even though no formal verification or validation methods were presented, the rationale behind the proof of the correctness of coordination algorithm follows from a formalisation of the system properties to be proved in terms of logical formulae. This paves the way for at least two formal analysis: (i) what-if assessment - coordination events can be modeled in terms of injected/removed tuples and deduced literals can be used to test the satisfaction of the system properties formulae. This first kind of verification can be done at design time, to assess properties of the whole system under certain conditions (events) and partially at run-time, to infer how the system will evolve assuming a knowledge restricted to a certain subset of locally perceived events; (ii) the second type of design time analysis starts from the literals that satisfy the properties formulae and proceeds backwards, to derive what are the events that lead the system to that given state. Future works will focus on such aspects, to derive formal procedures for correctness verification of algorithm built on top of Logic Fragments.

Several kinds of logics present interesting features to model and validate coordination primitives, such as: (i) paraconsistent logics (e.g. \cite{17}) and (ii) spatial-temporal logics, to assert properties depending on location and time parameters of system components. Future works should investigate how to integrate such kind of logics in the logical framework of Logic Fragments, to make it more expressive and powerful. Finally, we plan to realize an implementation of the model, including several kinds of semantics for Logic Fragments taking inspiration from the coordination primitives presented in \cite{6}.

References
