HAL open science

Logic Fragments: A Coordination Model Based on Logic Inference

Francesco Angelis, Giovanna Di Marzo Serugendo

To cite this version:

Francesco Angelis, Giovanna Di Marzo Serugendo. Logic Fragments: A Coordination Model Based on Logic Inference. 17th International Conference on Coordination Languages and Models (COORDINATION), Jun 2015, Grenoble, France. pp.35-48, 10.1007/978-3-319-19282-6_3 . hal-01774943

HAL Id: hal-01774943

https://inria.hal.science/hal-01774943v1

Submitted on 24 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Logic Fragments: a coordination model based on logic inference

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 Chemical-based coordination models have proven useful to engineer self-organising and self-adaptive systems. Formal assessment of emergent global behaviours in self-organising systems is still an issue, most of the time emergent properties are being analysed through extensive simulations. This paper aims at integrating logic programs into a chemical-based coordination model in order to engineer self-organising systems as well as assess their emergent properties. Our model is generic and accommodates various logics. 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. This paper discusses our logic-based coordination model and shows how to engineer and verify a simple pattern detection example and a gradient-chemotaxis example.

1 Introduction

Coordination models have been 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 [18]. A well-known difficulty in the design of self-organising systems stems 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 two case studies: a simple pattern recognition example and another one with the gradient and chemotaxis patterns. Finally, section 5 concludes the paper.

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.

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 as well as termination of programs is

easy to prove ([8]). Its major drawback is represented by the complexity of modeling real large applications.

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*. LSAs 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 program. 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 when specific commands (named *operators*) are present in 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 program 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 [18]. 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 formalisation.

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 [15] 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) represent the basic tuple matching mechanism.

In the ACLT model [7], 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.

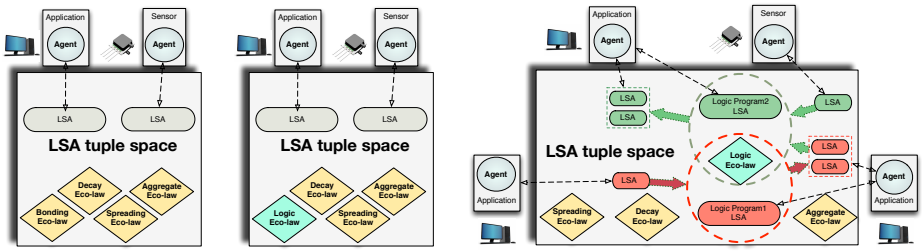
In our model we do not express coordination in terms of rewriting rules; moreover, the logic layer is enhanced by considering several types of logic languages.

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 specifications; (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 [12]: 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).

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.



(a) SAPERE model (b) Our model (c) Reactions among logic programs

Figure 1: The generalization of the SAPERE Model

The remaining components of the model are exactly the same as the 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 [12] 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 other 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 [14] 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 [14]) 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 ([14]): (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 refers to. In classical 2-valued logic programs, Herbrand interpretation can be defined through sets of atoms implicitly interpreted as *true*.
Example: $P = (C(x) \leftarrow A(x), B(x); \quad A(c) \leftarrow \Box; B(c) \leftarrow \Box;)$ is a *definite* logic program [14]. 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 ($\Box$) 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 support 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) [11,13,1,9,17].

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 $P \in Prop$ and $X_i \in (Const \cup Var)$ for $i = 1, \ldots, n$. A *ground literal* is a literal without variables. The set of all ground literals w.r.t. a set $Const$ is denoted $G(Const)$. The power set of $G(Const)$ is depicted $\mathcal{P}(G)$.

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

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 $I_{\hat{P}} = \{\hat{P}_w | w \in \mathcal{W}\} \subseteq G(Const)$.

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

Definition 5 (A-generator)**:** Given a literal $P(X_1, \ldots, X_n)$, an A-generator w.r.t. a function $U : Const^n \rightarrow \{T, F\}$ is the finite set:
$P^U(X_1, \ldots, X_n) = \{P(c_1, \ldots, c_n) \in I_{P(X_1,\ldots,X_n)} | U(c_1, \ldots, c_n) = T\}$.
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 $P(X_1, \ldots, X_n)$, 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:
$P^{H,V}(X_1, \ldots, X_n) = \{P(c_1, \ldots, c_n) \in I_{P(X_1,\ldots,X_n)} \cap V(H)\}$
If V is omitted, we assume that $V(H) = H$ (identity function).
Example: if $N = \{2, 3, 4\}$ and $V(N) = \{Even(x) | x \in N \wedge x \text{ is even}\}$, then $Even^{N,V}(X) = \{Even(2), Even(4)\}$.

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. The unfolding of these generators produces new facts for the interpretation of the logic program.

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

Definition 7 (Logic Fragments $\mathcal{LF}$)**:**
(I) $\triangle \in \mathcal{LF}$
(II) *(Grouping)* If $e \in \mathcal{LF}$ then $(e) \in \mathcal{LF}$
(III) *(Parallel-and)* If $e_1, e_2 \in \mathcal{LF}$ then $e_1 \sqcap e_2 \in \mathcal{LF}$
(IV) *(Parallel-or)* If $e_1, e_2 \in \mathcal{LF}$ then $e_1 \sqcup e_2 \in \mathcal{LF}$
(V) *(Composition)* If P is a logic program, $\mathcal{M}$ an execution modality, S a set of A,I-generators, $\varphi : \mathcal{P}(G) \to \{T, F\}$ and $e_p \in \mathcal{LF}$ then $(P, \mathcal{M}, e_P, S, \varphi) \in \mathcal{LF}$.

$\triangle$ is a special symbol used only in Logic Fragments to depict *all* the tuples in the container (both LSAs and Logic Fragments). $\mathcal{M}$ is the identifier of the way P is "executed" (we will use $\mathcal{M} = \mathcal{A}$ for the Apt-van Emden-Kowalski and $\mathcal{M} = \mathcal{K}$ 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,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). φ_T is constant and equal to T. For style reason, we will write $P^{\mathcal{M}}(e_P, S, \varphi)$ instead of $(P, \mathcal{M}, e_P, S, \varphi)$.

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

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

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

II) $v_L\big((e)\big) \triangleq v_L(e)$

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

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

V) $v_L\big(P^{\mathcal{M}}(e_P, S, \varphi)\big) \triangleq Q$

Q is the *consequent* of $P^{\mathcal{M}}$ and it is defined as follows: if $\mathcal{M}$ is not compatible with the logic program P or if $v_L(e_p) = \bowtie$ or if $\varphi\big(v_L(e_p)\big) = F$ then $Q = \bowtie$. φ "blocks" the execution of the program as long as a certain condition over e_p is not satisfied. Otherwise, based on $S = \{P_0^{H_0,V_0}(X_{01},...,X_{0t_0}),...,P_n^{H_n,V_n}(X_{n1},...,X_{nt_n}), P_0(Y_{01},...,Y_{0z_0}),...$ $P_m(Y_{m1},...,Y_{mz_m})\}$, the Logic eco-law produces the set of facts $Fs = \bigcup_{i=0}^{n} P_i^{v_L(H_i),V_i}(X_{i1},\ldots,X_{it_i}) \cup \bigcup_{i=0}^{m} P_i(Y_{i1},\ldots,Y_{iz_i})$. A,I-generators are then used to define sets of ground literals for the logic program which satisfy specific constraints; during the evaluation, 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 applying $\mathcal{M}$ on the new logic program $P' = P \cup \{l \leftarrow \square | l \in Fs\}$, enriched by all the facts contained in Fs. Note that there may be no need to explicitly calculate all the literals of A,I-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 $a, b \in \mathcal{LF}$ with $a \equiv b$ we state that $v_L(a) = v_L(b)$ for every set of literals L. Then for any $a, b, c \in \mathcal{LF}$:

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 or none, whereas *Parallel-or* ($\sqcup$) means executing only those ones that can be executed at a given time.

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 $v_L(e) \neq \bowtie$, then it is evaluated as an atomic operation (every symbol $\triangle$ in the sub Logic Fragments which composes e is always translated with the same set of actual tuples). Multiple Logic Fragments ready to be evaluated are computed in a non-deterministic order. The tuples inferred by the logic programs (with all used facts) are inserted in the container only when the evaluation of the whole logic program 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 Fs 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 depends on the implementation.

4 Case studies

By using Logic Fragments we can easily tackle interesting coordination problems and properties. Additional examples are reported in [6].

4.1 Palindrome recognition

As a first example we show an easy pattern recognition scenario. Assuming that an agent A inserts positive integers into the container, we want to discover which ones are palindromic numbers (i.e. numbers that can be read in the same way from left to right and from right to left). We assume 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 Logic Fragment $LF_p : P_p^{\mathcal{A}}(\triangle, \{N^{\triangle}, TestPalin\}, \varphi_p)$.

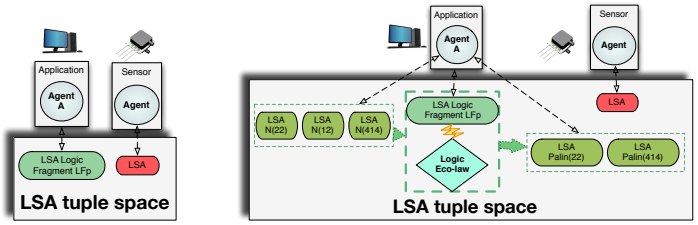
$\varphi_p(\triangle) = T \Leftrightarrow \exists w : N(X)_w \in \triangle$
$TestPalin(x) = \{TestPalin(a) | a \text{ is a positive palindromic number less than } d_{max}\}$

Logic code 1.1 Definite logic program P_p

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

P_p is the logic program in Code 1.1, evaluated with the Apt-van Embden Kowalski semantics ($\mathcal{A}$). The set S of A,I-generators is composed of two elements: $N^{\triangle}$ 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 d_{max}. These two sets of literals are treated as facts for P_p. According to φ, P_p is executed as soon as a number $N(a)$ is inserted into the container. The rule of the logic program P_p states that a number a is a palindromic number ($Palin(a)$) if a is a number ($N(a)$) and a passes the test for being palindromic ($TestPalin(a)$). We consider the tuple space shown in Figure 2a and 2b. At the beginning, agent A injects LF_p (Figure 2a). At a

later stage A injects $N(22)$ and the Logic Fragment is then executed. In this case, $N^{\triangle}$ is evaluated as $\{N(22)\}$. Moreover, $TestPalin(a)$ will contain $TestPalin(22)$, because it is palindromic. This means the consequent Q of LF_p contains $Palin(22)$, along with all the facts generated by the A,I-generators used in the logic program. If now agent A injects $N(12)$, the Logic Fragment is re-executed and $N^{\triangle}$ is evaluated as $\{N(22), N(12)\}$. This second number does not satisfy the palindromic test ($N(12) \notin TestPalin(x)$), so the 12 will not be considered as palindromic. Finally A injects $N(414)$ and during the re-execution of LF_p we obtain: $N^{\triangle} = \{N(22), N(12), N(414)\}$ and $N(414) \in TestPalin(x)$, so the consequent Q will contain $Palin(22)$ and $Palin(414)$ (Figure 2b). Note that if numbers were injected by agents different from A (like a sensor), the same reactions would take place.



(a) Injection of LF_p (b) Injection of numbers

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

Property 1: A palindromic integer $a \geq 0$ exists in the container if and only if $Palin(a)$ exists in the least Herbrand model of P'_p (the extension of P_p with all the facts created by A,I-generators).

Proof sketch: The property above states that by using the Logic Fragment LF_p we are able to correctly find out all the palindromic integers. Thanks to the logic programs and the semantic of Logic Fragments, we can easily verify that if such integers exist in the container then their literals are inferred in Herbrand model of P'_p. Moreover, given that such literals are only generated by LF_p, if such literals exist in the model then there must be the associated palindromic integers in the shared space.

4.2 Gradient and Chemotaxis patterns - general programs

In this second example we use Logic Fragments to implement the gradient and chemotaxis design patterns ([10]), 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. In these terms, the gradient algorithm is similar to the distance-vector algorithm. 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:

$$LF_{gc} : P_g^{\mathcal{A}}\Big(P_a^{\mathcal{A}}\big(\triangle \sqcap P_n^{\mathcal{K}}(\triangle, S_n, \varphi_n), S_a, \varphi_T\big), S_g, \varphi_T\Big) \sqcup P_{ch}^{\mathcal{A}}(\triangle, S_{ch}, \varphi_{ch})$$

Logic code 1.2 Program P_n - Next hop initialization

GPath(x,d_{max},$null$) $\leftarrow$ $\neg$existsGPath(x)

Logic code 1.3 Program P_a - Aggregation

cmpGradient$(x_1, x_2, y_1, y_2, z) \leftarrow$ Gmsg(x_1, x_2, y_1, z), GPath(x_1, y_2, w)
updateGPath$(x_1, y_1, x_2, z) \leftarrow$ cmpGradient(x_1, x_2, y_1, y_2, z), less(y_1, y_2)

Logic code 1.4 Program P_g - Spreading

spreadGradient$(x_1, local, z, y, x_2) \leftarrow$ updateGPath(x_1, y, x_2, z)

Logic code 1.5 Program P_{ch} - Chemotaxis

sendChemo$(m, x, w) \leftarrow$ Cmsg(m, x), GPath(x, y, w)

$\varphi_n(\triangle) = T \Leftrightarrow \exists w : Gmsg(x_1, x_2, y, z)_w \in \triangle, \quad \varphi_{ch}(\triangle) = T \Leftrightarrow \exists w : Cmsg(x, y)_w \in \triangle$
$S_{ch} = \{Cmsg^{\triangle}, GPath^{\triangle}\}$ $S_g = \{updateGPath^{e_{P_g}}\}$ $S_n = \{existsGPath^{\triangle,V}, Gmsg^{\triangle}\}$
$S_a = \{Gmsg^{e_{P_a}}, GPath^{e_{P_a}}, less\}$ $less(x, y) = \{less(a, b) | a < b,\ a, b \in \{1, ..., d_{max}\}\}$
$existsGPath(x)^{\triangle,V} = \{\neg existsGPath(a) \in I_{\neg existsGPath(x)} \cap V(\triangle)\}$
$V(\triangle) = \{\neg existsGPath(a) | \exists w : Gmsg(a, x, y, z)_w \in \triangle \wedge \neg\exists GPath(a, y, w)_w \in \triangle\}$

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. Values $null$, d_{max} and $local$ are considered constants. 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, this will be 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 $e_{P_a} = \triangle \sqcap P_n^{\mathcal{K}}(\triangle, S_n, \varphi_n)$ is executed when a gradient message is received from a neighbour ($\triangle$ can be executed directly but the *parallel-and* operator blocks the execution of outer fragments until $P_n^{\mathcal{K}}(\triangle, S_n, \varphi_n)$ 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, along with all the tuples in the container (fragment $\triangle$) are then treated as constituent for the fragment $e_{P_a} = P_a^{\mathcal{A}}(e_{P_e}, S_a, \varphi_T)$, i.e. they are used to generate facts for the program P'_a. This one is used to aggregate the hop-counter for the source with the one stored in the local container. e_{P_a} is finally treated as constituent for the fragment $e_{P_g} = P_g^{\mathcal{A}}(e_{P_a}, S_g, \varphi_T)$. Note that aggregation happens before spreading, imposing an order on the reactions. $P_g^{\mathcal{A}}$ is used to verify whether the gradient message must be spread to the neighbours. If so, a literal $spreadGradient(a, local, d, c, b)$ is inferred during the computation of its semantics, where *local* is translated with the name of the current node. Simultaneously, the Logic Fragment $P_{ch}^{\mathcal{A}}(\triangle, S_{ch}, \varphi_{ch})$ 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 until such a literal is 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.

Control code 1.6 Behaviour of agent A_{gc}

```
if spreadGradient(a, local, d, c, b) ∈ Q then
    send Gmsg(a, local, c + 1, d) to all neighbours but b
    remove container.Gmsg(a, x, y, z)_w for any w
if updateGPath(a, c, b, d) ∈ Q then
    update container.GPath(a, x, y)_w = GPath(a, c, b) for any w
if sendChemo(f, g, h) ∈ Q then
    send Cmsg(f, g) to node h
    remove container.Cmsg(f, g)
```

We consider the network of Figure 3; the Logic Fragment can be used to provide the gradient and chemotaxis functionalities as services to other

agents running on the same nodes. Assuming that agent A_{Gm} 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 LF_{gc} takes place, generating in the consequent Q of LF_{gc} literals $GPath(A, 0, A)$ (semantics of P'_n) and $spreadGradient(A, A, m_1, 0, A)$ (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 then $GPath(A, 3, D)$ is inferred in the consequent Q on node E. When the gradient message coming from B reaches E, $updateGPath(A, 1, B, m_1)$ is inferred in the semantics of program P'_a, so the hop-counter tuple is updated in $GPath(A, 2, B)$. Now assuming that agent A_{Cm} 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 $GPath(A, 2, C)$, the literal $sendChemo(m_2, A, C)$ is inferred in the model of P'_g, so the message is sent to node B. Similar reactions take place on node B, which finally sends the chemotaxis message to node A.

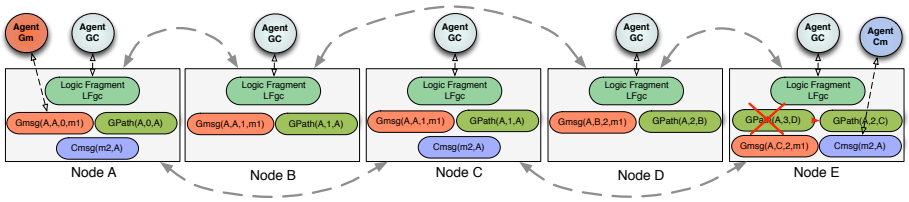


Figure 3: Network of 5 nodes

Property 2: Let $\mathcal{N}$ be a network with no disconnected hosts. If 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 A_{Gm} on one node S of $\mathcal{N}$, then there exists a finite time t^* for which the following statement holds: if an agent A_{Cm} 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.

Proof sketch: The rationale behind the proof consists in proving two categories of properties: (i) a *local property* which states that the number of gradient messages sent by each node is finite, due to the decrements of

the hop-counter caused by the applications of the aggregation-function; (ii) *global properties*, based on the local property holding in each node (e.g. we prove the creation of the shortest-path). The details are reported in [6]. Additional studies focusing on the integration of *spatial-temporal* logics in Logic Fragment are needed to prove the analogous statement when considering mobile nodes.

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 deducted 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: (i) paraconsistent logics (e.g. [17]) and (ii) spatial-temporal logics, to assert properties depending on location and time parameters of system components. We plan also to realise an implementation of the model, including several semantics for Logic Fragments taking inspiration from the coordination primitives presented in [7].

References

1. Apt, K.R., van Emden, M.H.: Contributions to the theory of logic programming. J. ACM 29(3), 841–862 (1982)

2. Banâtre, J.P., Le Métayer, D.: The gamma model and its discipline of programming. Sci. Comput. Program. 15(1), 55–77 (1990)
3. Berry, G., Boudol, G.: The chemical abstract machine. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 81–94. POPL '90, ACM (1990)
4. Castelli, G., Mamei, M., Rosi, A., Zambonelli, F.: Pervasive middleware goes social: The sapere approach. In: Proceedings of the 2011 Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops. pp. 9–14. SASOW '11 (2011)
5. Ciancarini, P., Franzè, F., Mascolo, C.: A coordination model to specify systems including mobile agents. In: Proceedings of the 9th International Workshop on Software Specification and Design. IWSSD '98, IEEE Computer Society, Washington, DC, USA (1998)
6. De Angelis, F.L., Di Marzo Serugendo, G.: Towards a logic and chemical based coordination model (2015), https://archive-ouverte.unige.ch/
7. Denti, E., Natali, A., Omicini, A., Venuti, M.: Logic tuple spaces for the coordination of heterogeneous agents. In: Baader, F., Schulz, K.U. (eds.) Frontiers of Combining Systems, Applied Logic Series, vol. 3, pp. 147–160. Kluwer Academic Publishers (1996), 1st International Workshop (FroCoS'96), Munich, Germany, 26–29 Mar. 1996. Proceedings
8. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. Commun. ACM 22(8) (1979)
9. Emden, M.H.V., Kowalski, R.A.: The semantics of predicate logic as a programming language. Journal of the ACM 23, 569–574 (1976)
10. Fernandez-Marquez, J.L., Di Marzo Serugendo, G., Montagna, S., Viroli, M., Arcos, J.L.: Description and composition of bio-inspired design patterns: a complete overview. Natural Computing 12(1), 43–67 (2013)
11. Fitting, M.: Fixpoint semantics for logic programming a survey. Theoretical Computer Science 278(1–2), 25 – 51 (2002), mathematical Foundations of Programming Semantics 1996
12. Kowalski, R.: Algorithm = logic + control. Commun. ACM 22(7), 424–436 (Jul 1979)
13. Kowalski, R., Kuehner, D.: Linear Resolution with Selection Function. Artificial Intelligence 2(3-4), 227–260 (Dec 1971)
14. Nilsson, U., Maluszynski, J.: Logic, Programming, and PROLOG. John Wiley & Sons, Inc., New York, NY, USA, 2nd edn. (1995)
15. Omicini, A., Denti, E.: From tuple spaces to tuple centres. Science of Computer Programming 41(3), 277–294 (nov 2001)
16. Viroli, M., Casadei, M., Omicini, A.: A framework for modelling and implementing self-organising coordination. In: Shin, S.Y., Ossowski, S., Menezes, R., Viroli, M. (eds.) 24th Annual ACM Symposium on Applied Computing (SAC 2009). vol. III, pp. 1353–1360. ACM, Honolulu, Hawai'i, USA (8–12 Mar 2009)
17. Vitória, A., Maluszyński, J., Szałas, A.: Modeling and reasoning in paraconsistent rough sets. Fundamenta Informaticae 97(4), 405–438 (2009)
18. Zambonelli, F., Omicini, A., Anzengruber, B., Castelli, G., De Angelis, F.L., Di Marzo Serugendo, G., Dobson, S., Fernandez-Marquez, J.L., Ferscha, A., Mamei, M., Mariani, S., Molesini, A., Montagna, S., Nieminen, J., Pianini, D., Risoldi, M., Rosi, A., Stevenson, G., Viroli, M., Ye, J.: Developing pervasive multi-agent systems with nature-inspired coordination. Pervasive and Mobile Computing 17, 236–252 (2015), special Issue 10 years of Pervasive Computing In Honor of Chatschik Bisdikian