ACE: a Flexible Environment for Complex Event Processing in Logical Agents

Stefania Costantini(✉)

Department of Information Science and Engineering, and Mathematics (DISIM)
University of L'Aquila, Italy, email: stefania.costantini@univaq.it

Abstract. In this paper we propose the general software engineering approach of transforming an agent into an Agent Computational Environment (ACE) composed of: 1) the "main" agent program; 2) a number of Event-Action modules for Complex Event Processing, including generation of complex actions; 3) a number of external contexts that the agent is able to access in order to gather information. In our view an ACE is composed of heterogeneous elements: therefore, we do not make assumptions about how the various components are defined, except that they are based upon Computational Logic. In order to show a concrete instance of ACE, we discuss an experiment based upon the DALI agent-oriented programming language and Answer Set Programming (ASP).

1 Introduction

Event processing (also called CEP, for "Complex Event Processing") has emerged as a relevant new field of software engineering and computer science [1, 2]. In fact, a lot of practical applications have the need to actively monitor vast quantities of event data to make automated decisions and take time-critical actions (the reader may refer to the Proceedings of the RuleML Workshop Series). Several products for event processing have appeared on the market, provided by major software vendors and by start-up companies. Many of the current approaches are declarative and based on rules, and often on logic-programming-like languages and semantics: for instance, [3] is based upon a specifically defined interval-based Event Calculus [4].

Complex Event Processing is particularly important in software agents. Naturally most agent-oriented languages, architectures and frameworks are to some extent event-oriented and are able to perform event-processing. The issue of Event Processing Agents (EPAs) is of growing importance in the industrial field, since agents and multi-agent systems are able to manage rapid change and thus to allow for scalability in applications aimed at supporting the ever-increasing level of interaction.

This paper is concerned with logical agent-oriented languages and frameworks, i.e., those approaches whose semantics is rooted in Computational Logic. There are several such approaches, some mentioned below (for a recent survey cf., e.g., [5]). For lack of space, we are not able here to discuss and compare their event-processing features. Rather, we recall only the ones that have more strongly influenced the present work.

A recent but well-established and widely used approach to CEP in computational logic is ETALIS [6, 7], which is an open source plug-in for Complex Event Processing implemented in prolog which runs in many Prolog systems (available at URL `http://code.google.com/p/etalis/`). ETALIS is in fact based on a declarative semantics, grounded in Logic Programming. Complex events can be derived from simpler events by means of deductive rules. ETALIS, in addition, supports reasoning about events, context, and real-time complex situations, and has a nice representation of time and time intervals aimed at stream reasoning. Relations among events can be expressed via several operators, reminiscent of those of causal reasoning and Event Calculus.

In the realm of logical agents, some work about CEP is presented in [8] and [9], which discuss the issue of complex reactivity by considering the possibility of selecting among different applicable reactive patterns by means of simple preferences. In [10], more complex forms of preferences among possible reactive behaviors are introduced. Such preferences can be also defined in terms of "possible worlds" elicited from a declarative description of a current or hypothetical situation, and can depend upon past events, and the specific sequence in which they occurred. [11] and [12, 13] discuss event-based memory-management, and temporal-logic-based constraints for complex dynamic self-checking and reaction.

Teleo-Reactive Computing by Kowalski and Sadri [14, 15] is an attempt to reconcile and combine conflicting approaches in logic programming, production systems, active and deductive databases, agent programming languages, and the representation of causal theories in AI, also considering complex events. In this approach, enhanced reactive rules determine the interaction of an agent with the environment in a logical but not necessarily "just" deductive way. The semantics relies upon an infinite Herbrand-like model which is incrementally constructed.

In this paper, we propose a novel conceptual view of Complex Event Processing in logical agents and a formalization of the new approach. We observe that a complex event cannot always result from simple deterministic incremental aggregation of simple events. Rather, an agent should be able to possibly interpret a set of simple events in different ways, and to choose among possible interpretations. We also consider complex actions, seen as agent-generated events. To this aim, we propose to equip agents with specific modules, that we call *Event-Action modules* (whose first general idea was provided in [16, 17]), describing complex events and complex actions. Such modules are activated by a combination of simple events, and may return: (i) possible interpretations of a set of simple events in terms of complex events; (iii) detection of anomalies; (iv) (sets of) actions to perform in response. An Event-Action module is re-evaluated whenever new instances of the "triggering" events become available, and may adopt any reasoning technique, including preferences, to identify plausible scenarios and make a choice in case of several possibilities.

Each agent can be in principle equipped with a number of such modules, possibly defined in different languages/formalisms. Also, in order to reason about events an agent may have to resort to extracting knowledge from heterogeneous external sources, that in general cannot be "wrapped" and considered as agents. We draw inspiration from the Multi-Context Systems (MCS) approach, which has been proposed to model information exchange among several knowledge sources [18–20]. MCSs are purposely defined so as to avoid the need to make such sources in some sense homogeneous: rather, the approach deals explicitly with their different representation languages and semantics. Heterogeneous sources are called "contexts" and in the MCS understanding are fundamentally different from agents: in fact, contexts do not have reactive, proactive and social capabilities, while it is assumed that they can be queried and updated. MCSs have evolved from the simplest form [18] to managed MCS (mMCS) [21], and reactive mMCS [20] for dealing with external inputs such as a stream of sensor data. MCSs adopt "bridge rules" for knowledge interchange, which are special rules assumed to be applied whenever applicable, so that contexts are constantly "synchronized".

In this paper we propose the software engineering approach of transforming an agent into an Agent Computational Environment (ACE) composed of: 1) the "main" agent program, or "basic agent"; 2) a number of Event-Action modules; 3) a number of external contexts that the agent is able to access. We assume the following. (i) Agents and modules can query (sets of) contexts, but not vice versa. (ii) Agents and modules are equipped, like contexts in MCSs, with bridge rules for knowledge interchange. Their application is however not only aimed at extracting knowledge from contexts, but also at knowledge interchange among the basic agent and Event-Action modules. On the one hand modules can access the agent's knowledge base, on the other hand the agent can access modules' conclusions. (iii) We do not make assumptions about how the various components are defined, except that they are based upon Computational Logic. We propose a full formalization with a semantics, where again we draw inspiration from MCSs' equilibrium semantics, on which we make necessary non-trivial enhancements. However, we devise a smooth extension which introduces as little additional technical machinery as possible. The approach proposed here introduces substantial advancements with respect to preliminary work presented in [16, 17]. The formalization and the semantics are fully novel.

To demonstrate practical applicability of ACEs, we discuss a prototypical example that we have experimented by using the DALI agent-oriented language [22, 23]. In the experimental setting we have adopted Answer Set Programming (ASP) for implementing Event-Action modules. In fact, Answer Set Programming (cf., among many, [24–27]) is a well-established logic programming paradigm where a program may have several (rather than just one) "model", called "answer set", each one representing a possible interpretation of the situation described by the program. We show how ASP-based Event-Action modules can be defined in a logic-programming-like fashion (we adopt in particular a DALI-like syntax) and then translated into ASP and executed via an ASP plugin integrated into the DALI interpreter. We define precise guidelines for the translation, and provide a practical example.

The paper is organized as follows. In Section 2 we provide the necessary background on MCSs. In Section 3 we present the proposal, its formal definition and its semantics. In Sections 4 and 5 we discuss one particular instance, based upon DALI and ASP modules. Finally, in Section 6 we discuss some related work and conclude.

2 Background

Managed Multi-Context systems (mMCS) [19, 21, 20]) model the information flow among multiple possibly heterogeneous data sources. The device for doing so is constituted by "bridge rules", which are similar to prolog or, more precisely, datalog rules (cf., e.g., [28] a for survey about datalog and prolog and the references therein for more information) but allow for knowledge acquisition from external sources, as in each element of their "body" the "context", i.e. the source, from which information is to be obtained is explicitly indicated. In the short summary of mMCS provided below we basically adopt the formulation of [20], which is simplified w.r.t. [21].

Reporting from [19], a logic L is a triple $(KB_L; Cn_L; ACC_L)$, where KB_L is the set of admissible knowledge bases of L, which are sets of KB-elements ("formulas"); Cn_L is the set of acceptable sets of consequences, whose elements are data items or "facts" (in [19] these sets are called "belief sets"; we adopt the more neutral terminology of "data sets"); $ACC_L : KB_L \to 2^{Cn_L}$ is a function which defines the semantics of L by assigning each knowledge-base an "acceptable" set of consequences. A managed Multi-Context System (mMCS) $M = (C_1, \dots, C_n)$ is a heterogeneous collection of contexts $C_i = (L_i; kb_i; br_i)$ where L_i is a logic, $kb_i \in KB_{L_i}$ is a knowledge base (below "knowledge base") and br_i is a set of bridge rules. Each such rule is of the following form, where the left-hand side $o(s)$ is called the *head*, also denoted as $hd(\rho)$, the right-hand side is called the *body*, also denoted as $body(\rho)$, and the comma stand for conjunction.

$$\begin{aligned} o(s) \leftarrow\ & (c_1 : p_1), \dots, (c_j : p_j), \\ & \mathit{not}\,(c_{j+1} : p_{j+1}), \dots, \mathit{not}\,(c_m : p_m). \end{aligned}$$

For each bridge rule included in a context C_i, it is required that $kb_i \cup o(s)$ belongs to KB_{Li} and, for every $k \leq m$, c_k is a context included in M, and each p_k belongs to some set in KB_{L_k}. The meaning is that $o(s)$ is added to the consequences of kb_i whenever each atom p_r, $r \leq j$, belongs to the consequences of context c_r, while instead each atom p_w, $j < w \leq m$, does not belong to the consequences of context c_s. While in standard MCSs the head s of a bridge rule is simply added to the "destination" context's knowledge base kb, in managed MCS kb is subjected to an elaboration w.r.t. s according to a specific operator o and to its intended semantics: rather than simple addition. Formula s itself can be elaborated by o, for instance with the aim of making it compatible with kb's format, or via more involved elaboration.

If $M = (C_1, \dots, C_n)$ is an MCS, a data state or, equivalently, belief/knowledge state, is a tuple $S = (S_1, \dots, S_n)$ such that each S_i is an element of Cn_i. Desirable data states are those where each S_i is acceptable according to ACC_i. A bridge rule ρ is applicable in a knowledge state iff for all $1 \leq i \leq j : p_i \in S_i$ and for all $j+1 \leq k \leq m : p_k \notin S_k$. Let $app(S)$ be the set of bridge rules which are applicable in a data state S.

For a logic L, $F_L = \{s \in kb \,|\, kb \in KB_L\}$ is the set of formulas occurring in its knowledge bases. A *management base* is a set of operation names (briefly, operations) OP, defining elaborations that can be performed on formulas, e.g., addition of, revision with, etc. For a logic L and a management base OP, the set of operational statements that can be built from OP and F_L is $F_L^{OP} = \{o(s) \,|\, o \in OP, s \in F_L\}$. The semantics of such statements is given by a management function, which maps a set of operational statements and a knowledge base into a modified knowledge base. In particular, a management function over a logic L and a management base OP is a function $mng : 2^{F_L^{OP}} \times KB^L \to 2^{KB_L} \setminus \emptyset$. The management function is crucial for knowledge incorporation from external sources, as it is able to perform any elaboration on the knowledge base given the acquired information.

Semantics of mMCS is in terms of *equilibria*. A data state $S = (S_1, \dots, S_n)$ is an equilibrium for an MCS $M = (C_1, \dots, C_n)$ iff, for $1 \leq i \leq n$, $kb'_i = S_i \in ACC_i(mng_i(app(S), kb_i))$. Thus, an equilibrium is a global data state composed of acceptable data states, one for each context, encompassing inter-context communication determined by bridge rules and the elaboration resulting from the operational statements and the management functions.

Equilibria may not exist (where conditions for existence have been studied, and basically require the avoidance of cyclic bridge-rules application), or may contain inconsistent data sets (local inconsistency, w.r.t. *local consistency*). A management function is called *local consistency (lc-) preserving* iff, for every given management base, kb' is consistent. It can be proved that a mMCS where all management functions are lc-preserving is locally consistent. Algorithms for computing equilibria have recently been proposed (see, e.g., [29] and the references therein). Notice that bridge rules are intended to be applied whenever they are applicable. In [20], where mMCS are adapted so as to continuous reasoning in dynamic environments, where contexts' contents are updated by external input, the notion of a "run" is in fact introduced. A run of mMCS M under a sequence $Obs^0, Obs^1, \dots$ of observations is a sequence $R = \langle S^0, KB^0 \rangle, \langle S^1, KB^1 \rangle \dots$ such that $\langle S^0, KB^0 \rangle$ is a *full equilibrium* of M under Obs^0, and for $i > 0$ $\langle S^i, KB^i \rangle$ is a full equilibrium of M under Obs^i, where a full equilibrium is obtained by taking the observations into consideration in every context for bridge rules application: in fact, observation literals can occur in bridge rule bodies.

3 Agents as Computational Environments

In the approach that we present here, an agent is equipped with a number of *Event-Action modules* for performing Complex Event Processing, and with a number of contexts which are known to the agent and to which the agent may resort for gathering information. We assume the agent to be based upon its own underlying logic, and so are the Event-Action modules and the contexts. Different Event-Action modules may be based on different logics, depending upon the task they are supposed to perform: for instance, some modules might be aimed at event interpretation, some others at learning patterns from event occurrences, some others at evaluating possible courses of action, etc.

In order to finalize an agent's operation, we assume that each Event-Action module admits just one acceptable sets of consequences, differently from MCSs where each context may in principle admit several. In such case, we assume to choose one by means of some kind of selection function. In [20] the problem is mentioned in the conclusions, referring to unwanted sources of non-determinism that may arise. They thus suggest to adopt a global preference criteria to fix the problem, and also mention some existing preference functions that might be exploited. However, as seen below we will take the problem as solved for contexts to which agents are able to refer to, so we will care only about consequences selection for Event-Action modules.

Let a logic L be defined as reported in previous section.

Definition 1. *Let a* Logic with Preferences $L^{\mathcal{P}}$ *be a quadruple* $(KB_{L^{\mathcal{P}}}; Cn_{L^{\mathcal{P}}}; ACC_{L^{\mathcal{P}}}; \mathcal{P})$ *where* $ACC_{L^{\mathcal{P}}}$ *is a function which extends the one defined before for a logic L since it selects the "preferred" one among acceptable set of consequences of the given knowledge base, according to the preference criterion* $\mathcal{P}$.

As seen, we leave the preference criterion as an open parameter, as each module may in principle employ a different one. In general, a preference criterion is some kind of device which induces a total order on $Cn_{L^{\mathcal{P}}}$. On one extreme it can even be random choice, though in general domain/application-dependent criteria will be better suited.

Similarly to what is done in Linear Time Logic (LTL), we assume a discrete, linear model of time where each state/time instant can be represented by an integer number. States $t_0, t_1, \ldots$ can be seen as time instants in abstract terms. In practice we will have $t_{i+1} - t_i = \delta$, where δ is the actual interval of time after which we assume a given system to have evolved. In particular, agent systems usually evolve according to the perception of events (among which we include communications with other agents).

Definition 2. *Let* $\Pi = \Pi_1, \Pi_2, \ldots$ *be a sequence of sets of events, where* Π_i *is assumed to have been perceived by given agent at time* $i > 0$*. Each event in* Π*, say* E*, can be denoted as* $E : t_i$ *where* t_i *is a time-stamp indicating time* i*, and meaning that* $E \in \Pi_i$*. By* $E : [t_i, t_j]$ *with* $1 \leq i \leq j$ *we mean that* E *persists during an interval, i.e., we have* $E : t_s$ *for every* $i \leq s \leq j$.

A number of expressions can be defined on events, for instance: $E_1, \ldots, Ek : [t_i, t_j]$ to mean that all the E_is, $i \leq k$, persist in given interval; $E_1, \ldots, Ek \setminus E : [t_i, t_j]$ intending that all the E_is persist in given interval, where E does not occur therein. We do not go into the detail, but we assume that some syntax is provided for defining *Event Expressions*, where each such expression can be evaluated to be true or false w.r.t. Π.

Definition 3. *Let* $\mathcal{H}$ *be a set of sequences of sets of events as defined above, i.e.,* $\Pi \in \mathcal{H}$ *is of the form* $\Pi = \Pi_1, \Pi_2, \ldots$ *Let* $\mathcal{E}$ *be a set of event expressions and let* $ev^{\mathcal{E}} : \mathcal{E}, \mathcal{H} \rightarrow \{true, false\}$ *be an evaluation function which establishes whether an event expression* $\epsilon \in \mathcal{E}$ *is true/false w.r.t.* $\Pi \in \mathcal{H}$.

Below we define Event-Action modules, which include an event expression that functions as a trigger, meaning that the module is evaluated whenever the given event expression is entailed by the present event sequence. Event-Action modules may resort to bridge rules for obtaining knowledge from both external contexts, and from the agent's knowledge base. They elicit, by means of some kind of reasoning, complex events that may have occurred and/or actions that the agent might perform. In case several possibilities arise, preferences are employed to finalize the reasoning.

Definition 4. *We let an* Event-Action module *be defined as* $M = ({L_M}^{\mathcal{P}}; kb_M; br_M; mng_M; tr_M)$ *where* ${L_M}_i^{\mathcal{P}}$ *is a Logic with Preferences (as defined above) and* $kb_M \in KB_{{L_M}^{\mathcal{P}}}$ *is a knowledge base.* br_M *is a set of bridge rules of the form defined for mMCS (seen in previous section), and* mng_M *is the management function adopted by the module.* tr_M *is an event expression which triggers the module evaluation, where* tr_M *belongs to a given set* $\mathcal{E}$ *associated to evaluation function* $ev^{\mathcal{E}}$.

Definition 5. *An Event-Action module M is* active *w.r.t. sequence Π of sets of events (or simply "active" if leaving Π implicit) iff $ev^{\mathcal{E}}(tr_M, \Pi) = true$, i.e., if Π enables the module evaluation.*

Complex events and/or actions derived from the evaluation of an active Event-Action module will be included in its set of consequences, whose contents will also depend upon bridge-rules application.

An agent program can be defined in any agent-oriented computational-logic-based programming language, such as, e.g., DALI (cf. [22, 23]), AgentSpeak (cf. [30, 31] and the references therein), GOAL (cf. [32] and the references therein) 3APL (cf. [33] and the references therein), METATEM (cf. [34] and the references therein), KGP (cf [36] and the references therein), or any other (cf. [35] for a survey). So, to our purposes we provide a very simple general definition of a basic agent, able to encompass any of the mentioned approaches. Only, we add bridge rules, in a form which allows an agent to access contexts, and to incorporate Event-Action modules results that can be either complex events or complex actions.

Definition 6. *We let a* basic agent *be defined as $A = (L_A; kb_A; br_A, mng^A)$ where L_A is a logic, $kb_A \in KB_{L_A}$ is a knowledge base (encompassing the agent program), and br_A is a set of bridge rules of the form:*

$$\begin{aligned} o(s) \leftarrow {} & B_1, \dots, Bj, \\ & not\, C_{j+1}, \dots, not\, C_k. \end{aligned}$$

where, for $j > 0$, $k \geq 0$, each of the Bs and Cs can be in one of the following forms, where p is an atom: (i) $(c : p)$ where c is a context. (ii) $(m : ce : p)$ or $(m : act : p)$ where m is an Event-Action module, ce is a constant meaning "complex event" and act is a constant meaning "complex action". mng^A is the management function which, analogously to what seen before for mMCSs, incorporates the conclusion $o(s)$ of bridge rules into the agent's knowledge base.

Thus, agent A can update its knowledge base according to what can or cannot be concluded by a set of contexts and Event-Action modules and according to its own knowledge management policies.

Definition 7. *An Agent Computational Environment (ACE) $\mathcal{A}$ is a tuple*

$$\langle A, M_1, \dots, M_r, C_1, \dots, C_s \rangle$$

where, for $r, s \geq 0$, A is a basic agent, the M_is are Event-Action modules and the C_is are contexts in the sense of MCSs[1]*. All components can include bridge rules. For the basic agent A they are of the form just seen above. For the other components they are of the form seen for mMCSs, with the following restrictions on bridge rule bodies: both contexts and the basic agent A can be mentioned in bodies of bridge rules in the M_is; only contexts can be mentioned in bodies of bridge rules in the C_is.*

[1] The acronym "ACE" emerged by chance: nevertheless, with the occasion the author wishes to dedicate the ACE approach to the memory of Alan Turing.

That is, contexts can only query other contexts; Event-Action modules can query contexts, but also the basic agent (thus, they have some access to its knowldedge base); the basic agent can query every component (and will in general interact with the environment and with other agents).

Definition 8. *Let $\mathcal{A} = \langle A_1, \dots, A_h \rangle$ be an ACE, defined as above (i.e., the A_is include the basic agent, and, possibly, Event-Action modules and contexts). A data state of $\mathcal{A}$ is a tuple $S = (S_1, \dots, S_h)$ such that each of the S_is is an element of Cn_i, according to the logic in which S_i is defined.*

As for MCSs, desirable data states are those where each S_i is acceptable according to ACC_i, taking bridge rules application into account. However, bridge rules applicability here is different. In fact, it is required that each Event-Action module which is queried is also active. So, the *app* function must be extended w.r.t. mMCSs, as for determining which bridge rules can be applied in a certain data state it will have to take into consideration also the sequence of sets of events occurred so far.

Definition 9. *Let S be a data state of ACE $\mathcal{A}$, and let Π be a sequence of sets of events. A bridge rule ρ is applicable in S given Π iff every Event-Action module mentioned in the body is active w.r.t. Π, and for every positive literal in the body referring to component A_i the atom occurring therein belongs to S_i and for every negative literal in the body referring to component A_i the atom occurring therein does not belong to S_i. Let $app(S, \Pi)$ be the set of bridge rules which are applicable in a data state S w.r.t. sequence of sets of events Π.*

We can extend to ACEs the definition of equilibrium already provided for mMCSs.

Definition 10. *A data state $S = (S_1, \dots, S_n)$ of ACE $\mathcal{A}$ is an equilibrium w.r.t. sequence of sets of events Π, and is then denoted as ${\Xi_{\mathcal{A}}}^{\Pi}$, iff for $1 \leq i \leq n$, $kb'_i = S_i \in ACC_i(mng_i(app(S, \Pi), kb_i))$.*

For every component which based upon a preferential logic (i.e., at least Event-Action modules) ACC_i is, as said before, univocal. It is easy to see that if the set of contexts included in ACE $\mathcal{A}$ constitutes in itself an mMCS which admits equilibria, then also $\mathcal{A}$ does so. As soon as the sequence of set of events acquires more elements over time, this determines new equilibria to be formed.

Definition 11. *Given ACE $\mathcal{A}$ and sequence of sets of events $\Pi = \Pi_1, \Pi_2, \dots, \Pi_k, \dots$, the corresponding ACE-Evolution is the sequence of equilibria ${\Xi_{\mathcal{A}}}^{\Pi_1}$,${\Xi_{\mathcal{A}}}^{\Pi_1,\Pi_2}$, $\dots$, ${\Xi_{\mathcal{A}}}^{\Pi_1,\Pi_2,\dots,\Pi_k}$, $\dots$*

This implies that each Event-Action module is either evaluated or not in different stages of an ACE's evolution. In case a bridge rule queries a module which at that stage is not active, no result will be returned. This is a departure from MCSs, where each literal in a bridge rules is supposed to always evaluate to either true or false. In case of ACEs, some bridge rules will be "idle" at some evolution stages, i.e., unable to return results. Results may anyway have been returned previously or may be returned later, whenever the involved modules become active. Event-Action modules might be for instance defined in ETALIS, or in Reactive Answer Set Programming [37], or in Abductive Logic Programming or in many other formalisms.

For lack of space we cannot discuss verification. However we may notice that via LTL (Linear Temporal Logic), interesting properties of an ACE can be defined and verified. For instance, for proposition φ it can be checked whether φ holds for agent A in some equilibrium reached at a certain time or within some time interval.

4 Event-Action Modules in DALI

The ACE framework is especially aimed at designing agent-based computational environments involving heterogeneous components. Purposely, the proposal does not make assumptions about the logics and the preference rules the various component are based upon. In order to make the proposal less abstract by demonstrating its practical applicability, in this section we however report about an experiment that we have been developing in DALI, where: the basic agent is a DALI agent; contexts are simply prolog knowledge bases; Event-Action modules are defined in a DALI-like syntax, and are then translated into Answer Set Programming (ASP), and thus executed by means of the ASP plugin which has been integrated into the DALI interpreter. ASP is in fact quite suitable for obtaining plausible scenarios from a set of constraints. Several approaches to preferences have been defined for ASP: cf., e.g., [20] and the references therein, and also [38–41] and [42, 43]). The translation is discussed in the next section.

In the examples below syntax is reminiscent of DALI, which is a prolog-like language with predicates in lowercase and variables in uppercase. Postfix E designs a predicate as an event, postfix A as an action, and postfix P an event which has occurred in the past. Special keywords indicate, for the convenience of programmers and readers, different parts of each module. However, there is no special reason for adopting these keywords rather than any other syntax.

4.1 Examples of Event-Action Modules

Deriving Complex Events The following example illustrates an *Event-Action module* evaluating symptoms of either pneumonia, or just flu, or both (clearly, we do not aim at medical precision). The Event-Action module will be activated whenever its *triggering events* occur within a certain time interval, and according to specific conditions: in the example, the module is evaluated whenever in the last two days both high temperature and intense cough have been recorded. For the sake of conciseness the example is propositional, thus referring to an unidentified single patient. In general, it might, by means of introducing variables, refer to a generic patient/person.

EVENT-ACTION-MODULE diagnosis

$TRIGGER$
$(high\_temperatureE\ AND\ intense\_coughE) : [2days]$
$COMPLEX\_EVENTS$
$suspect\_flu\ OR\ suspect\_pneumonia$

$suspect\_flu$:- $high\_temperatureP$.
$suspect\_pneumonia$:- $high\_temperatureP : [4days], intense\_coughP$.
$suspect\_pneumonia$:-
$diagnosis(clinical\_history, suspect\_pneumonia) : diag\_knowledge\_base$.
PREFERENCES
$suspect\_flu$:- $patient\_is\_healty$.
$suspect\_pneumonia$:- $patient\_is\_at\_risk$.
ACTIONS
$stay\_in\_bedA$:- $suspect\_flu$.
$take\_antibioticA$:- $suspect\_flu$,
$high\_temperatureP : [4days], not\ suspect\_pneumonia$.
$take\_antibioticA$:- $suspect\_preumonia$.
$consult\_lung\_doctorA$:- $suspect\_preumonia$.
MANDATORY
$suspect\_preumonia$:- $high\_temperatureP : [4days]$,
$suspect\_fluP, take\_antibioticP : [2days]$.

From given symptoms, either a suspect flu or a suspect pneumonia or both can be derived. This is stated in the *COMPLEX_EVENTS* section, which in general lists the complex events that the module might infer from the given definition. For suspecting pneumonia high temperature should have lasted for at least four days, accompanied by intense cough. Pneumonia is also suspected if the patient's clinical history suggests this might be the case. This is an example of a bridge rule, as the analisys of clinical history is demanded to an external context, here indicated as $diag\_knowledge\_base$. Notice that, in our implementation, every predicate not defined within the module is obtained from the agent's knowldge base via a standard bridge rule, that might look, for agent Ag, of the form A :- $A : Ag$. As stated before in fact, in an ACE every Event-Action module has access, via bridge rules, to the basic agent knowledge base.

Explicit preferences are expressed in the *PREFERENCES* section. A conclusion is preferred if the conditions are true: therefore, in this case it is stated that hypothesizing a flu should be preferred in case the patient is healthy, while pneumonia is the preferred option for risky patients. Actions to undertake in the two cases are specified, and the agent can access them via bridge rules. In this case, if a flu is suspected then the patient should stay in bed, and if the high temperature persists then an antibiotic should also be assumed (even if pneumonia is not suspected). In case of suspect pneumonia, an antibiotic is mandatory, plus a consult with a lung doctor.

The *MANDATORY* section of the module includes constraints, that may be of various kinds: in this case, it specifies which complex events must be mandatorily inferred in module (re)evaluations if certain conditions occur. Specifically, pneumonia is to be assumed *mandatorily* whenever flu has been previously assumed, but high temperature persists despite at least two days of antibiotic therapy.

Monitoring the Environment The next Event-Action-module models an agent's behavior if encountering a traffic light. The triggering events are the presence of the traffic light, and the color of the traffic light as perceived by the agent. The objective of the module is to assess whether the observed color is correct (*CHECK* section), to detect and manage possible anomalies, and to determine what to do then. The module evaluates as correct any color which is either red or yellow or green. Section *ANOMALIES* detects violations to the the expected color or color sequence which is, namely, yellow after green, red after yellow and green after red. Actions for both the normal and anomalous case are specified. Postfix P indicates the last previous value of an event.

Thus, if the agent meets a traffic light which is, say, red, then the agent stops, and the event $colorE(tl, red)$ is recorded as a *past event* in the form $colorP(tl, red)$. If, after some little while, the event $colorE(tl, green)$ arrives, then the module is re-evaluated and the agent passes. The *ANOMALIES* section copes with two cases: (i) the color is incorrect, e.g., the traffic light might be dark or flashing; (ii) the agent has observed the traffic light for a while, and the color sequence is incorrect. This is deduced by comparing the present color $colorE(tl, c1)$ with previous color $colorP(tl, c2)$. Actions to undertake in case of anomaly are defined, that in the example imply passing with caution and reporting to the police in the former case, and choosing another route and reporting to the police in the latter. Anomaly detection is in our opinion relevant, as anomalies in event occurrence may be considered themselves as particular (and sometimes important) instances of complex events.

EVENT-ACTION-MODULE traffic

$TRIGGER\ traffic\_lightE(tl)\ AND\ colorE(tl, C)$
$CHECK$
$color\_ok(tl, C), C = red\ XOR$
$color\_ok(tl, C), C = green\ XOR$
$color\_ok(tl, C), C = yellow$:- $colorE(tl, C)$
$ANOMALIES$
$anomaly1(tl)$:- $colorE(tl, C), not\ color\_ok(tl, C)$.
$anomaly2(tl)$:- $colorE(tl, red), not\ colorP(tl, yellow)$.
$anomaly2(tl)$:- $colorE(tl, yellow), not\ colorP(tl, green)$.
$anomaly2(tl)$:- $colorE(tl, green), not\ colorP(tl, red)$.
$ACTIONS$
$stopA$:- $color\_ok(tl, red)$.
$stopA$:- $color\_ok(tl, yellow)$.
$passA$:- $color\_ok(tl, green)$.
$ANOMALY\_MANAGEMENT\_ACTIONS$
    $pass\_with\_cautionA$,
    $report\_to\_policeA(tl)$:- $anomaly1(tl)$.
    $stopA$,
    $change\_wayA$,
    $report\_to\_policeA(tl)$:- $anomaly2(tl)$

Generating Complex Actions The last example is related to what happens when two persons meet. In such a situation, it is possible that the one who first sees the other smiles, and then either simply waves or stops to shake hands: section *RELATED_EVENTS* specifies, as a boolean combination, events that may occur contextually to the triggering ones. Some conditions are specified on these events, for instance that one possibly smiles and/or waves if (s)he is neither in a bad temper nor angry at the other person. Also, one who is in a hurry just waves, while good friends or people who meet each other in a formal setting should shake hands. In this sample formulation, actions simply consist in returning what the other one does, and it is anomalous not doing so (e.g., if one smiles and the other does not smile back). The expression $meet\_friend(A, F)$ means that agent A meets agent F: then, each one will possibly make some actions and the other one will normally respond. This module is totally revertible, in the sense that it manages both the case where "we" meet a friend and the case where vice versa somebody else meets us. This is the reason why in some module sections events have no postfixes. In fact, *meet_friend(A,F)*, *smile*, *wave* and *shake_hands* are present events if a friend meets "us", and are actions if "we" meet a friend.

Postfixes appear in the *ACTIONS* and *ANOMALY* sections, where all elements (whatever their origin) have become past events to be coped with. The *PRECONDITIONS* section expresses action preconditions, via connective $:<$. Section *MANDATORY* defines obligations, here via a rule stating that it is mandatory to shake hands in a formal situation. The anomaly management section may include counter-measures to be taken in case of unexpected behavior, that in the example may go from asking for explanation to getting angry, etc.

EVENT-ACTION-MODULE meet

```
TRIGGER meet_friend(A, F),
RELATED_EVENTS
smile(A, F) OR (wave(A, F) XOR shake_hands(A, F))
PRECONDITIONS
smileA(A, F) :< not angry(A, F), not bad_temper(A).
waveA(A, F) :< not angry(A, F).
shake_handsA(A, F) :<
    good_friends(A, F), not angry(A, F), not in_a_hurry(A), not in_a_hurry(F).
MANDATORY
shake_handsA(A, F) :- formal_situation(A, F).
ACTIONS
smiled(X, Y) :- smileP(X, Y).
waved(X, Y) :- waveP(X, Y).
shaken_hands(X, Y) :- shake_handsP(X, Y).
smileA(A, F) :- smiled(F, A).
waveA(A, F) :- waved(F, A).
shake_handsA(A, F) :- shaken_hands(F, A).
ANOMALY
anomaly1(meet_friend(A, F)) :-  smileP(A, F), not smileA(F, A).
anomaly2(meet_friend(A, F)) :-  waveP(A, F), not waveA(F, A).
anomaly3(meet_friend(A, F)) :-  shake_handsP(A, F), not shake_handsA(F, A).
ANOMALY_MANAGEMENT_ACTIONS
...
```

5 ASP Representation of DALI Event-Action Modules

The examples that we have illustrated above have been presented in a DALI-like syntax. However, DALI (being a prolog-like language with a minimal model semantics [44]) cannot account for the different scenarios outlined by Event-Action modules. In fact, each module can perform a selection (according to conditions and preferences) among different complex events or complex actions that might result from the given simple events and the available complex events/actions description. In order to suitably implement such intended behavior, we have devised a prototypical implementation where Event-Action modules are translated into Answer set programs. Answer set programming (ASP) is nowadays a well-established and successful programming paradigm based upon answer set semantics [24, 45–47], with applications in many areas (cf., e.g., [25, 27, 26] and the references therein). An answer set program may have several answer sets, each one representing a solution of the problem encoded in the program. As seen below, each Event-Action module can be translated in a fully automated way into an ASP module.

The way of evaluating Event-Action modules within a DALI ACE basic functioning is the following.

– At each agent's evolution step, i.e., when new events have been perceived, ASP modules corresponding to Event-Action modules are (re-)evaluated given the history of all events perceived, and the agent's current knowledge base. It is required to re-evaluate a module whether the condition in the *TRIGGER* headline is satisfied. As seen, this condition is specified in terms of a boolean combination of present and/or past events. DALI is equipped with timestamps and time intervals and is thus able to perform such evaluation.
– A module will admit as a result of evaluation none, one or more answer sets. Non-existence of answer sets can result from constraint violation, and implies that no reaction to triggering events can be determined at present.
– If the module admits answer sets, one answer set among the available ones must be selected. Answer set selection is performed according to the preferences expressed in section *PREFERENCES*. If there are answer sets which are equally preferred, the current solution in the prototypical implementation is random choice. Methods for choosing answer sets according to preferences are discussed for instance in [48, 38].

5.1 Answer Set Programming (ASP) in a Nutshell

Answer Set Programming (ASP) is a logic programming paradigm based upon logic programs with default negation under the *answer set semantics*, which [24, 45]. This semantics considers logic programs as sets of inference rules (more precisely, default inference rules). In fact, one can see an answer set program as a set of constraints on the solution of a problem, where each answer set represents a solution compatible with the constraints expressed by the program. The reader may refer, among many, to [24, 45, 25, 27] for a presentation of ASP as a tool for declarative problem-solving.

Syntactically, an answer set program (or, for short, just "program") Π is a collection of *rules* of the form $H \leftarrow L_1, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_{m+n}$ where H and each L_is, $m \geq 0$ and $n \geq 0$, are atoms. Symbol $\leftarrow$ is usually indicated with :- in practical systems. An atom L_i and its negative counterpart $\mathit{not}\, L_i$ are called *literals*. The left-hand side and the right-hand side of the clause are called *head* and *body*, respectively. A rule with empty body is called a *fact*. A rule with empty head is a *constraint*, where a constraint of the form $\leftarrow L_1, ..., L_n.$ states that literals $L_1, \ldots, L_n$ cannot be simultaneously true in any answer set.

A program may have several answer sets, each of which represents a solution to the given problem which is consistent w.r.t. the problem description and constraints. If a program has no answer set, this means that no such solution can be found. and the program is said to be *inconsistent* (w.r.t. *consistent*).

In practical terms, a problem encoded by means of an ASP program is processed by an ASP solver which computes the answer set(s) of the program, from which the solutions can be easily extracted by abstracting away from irrelevant details. Several well-developed answer set solvers [49] can be freely downloaded by potential users. All solvers provide a number of additional constructs and features useful for practical programming, that for simplicity we do not consider here. Solvers are periodically checked and compared over well-established benchmarks, and over challenging sample applications proposed at the yearly ASP competition (cf. [50]). The expressive power of ASP and its computational complexity have been deeply investigated [51].

5.2 Translation Guidelines

The answer set programming (module) Π corresponding to a given Event-Action module is obtained by translating into ASP the contents of sections *COMPLEX_EVENTS*, *CHECK*, *RELATED_EVENTS*, *ANOMALIES* and *MANDATORY*. The translation can be fully defined and automated. Sections *ACTIONS*, *ANOMALY_MANAGEMENT_ACTIONS* and *PRECONDITIONS* do not need translation, as they are in fact composed of logic programming rules which by definition are ASP rules. So, these sections are just copied (with some minor modifications seen below) into the ASP version of the given Event-Action module. Notice that we do not need stream or reactive answer set programming, as triggers and time intervals are coped with by the underlying DALI interpreter. Each module resulting from the translation is evaluated in the standard ASP fashion whenever the conditions for doing so occur. The translation can be in particular performed by exploiting the following ASP patterns. Please consider that ASP solvers provide sophisticated and flexible programing constructs for expressing many of these patterns. However, for the sake of clarity we consider only the basic simple forms listed below.

conj In ASP, the conjunction among a number of elements $a_1, \ldots, a_n$ is simply expressed as $conj \leftarrow a_1, \ldots, a_n$.

or-xor Disjunction between two elements a and b is expressed by the cycle $a \leftarrow not\, b \quad b \leftarrow not\, a$. This disjunction is not exclusive, since either a or b or both might be derived elsewhere in the program. To obtain exclusive disjunction, a constraint $\leftarrow a, b$ must be added. A constraint in ASP can be read as *it cannot be that all literals in the body are true*. In the case of the exclusive disjunction of a and b, it cannot be that both a and b belong to the same answer set. Disjunction can also be expressed on several elements.

choice Choice, or possibility, or hypothesis, expressing that some element a may or may not be included in an answer set, can be expressed by means of a cycle involving a fresh atom, say na. The cycle is of the form $a \leftarrow not\, na \quad na \leftarrow not\, a$. Therefore, an answer set will contain either a or na, the latter signifying the absence of a.

choyf Makes the *choice* pattern stronger: element a can be in fact chosen only if certain conditions $Conds$ are satisfied, is expressed by a choice pattern plus a rule $c \leftarrow Conds$ and a constraint $\leftarrow a, not\, c$. The constraint states that a cannot be hypothesized in an answer set if c does not hold, i.e., if $Conds$ are not implied by that answer set.

mand Mandatory presence in an answer set of atom a defined by rule $a \leftarrow Body$ whenever $Body$ is implied by that answer set can be obtained as follows. In addition to the defining rule $a \leftarrow Body$, a constraint must be added of the form $\leftarrow not\, a, Body$ stating that it cannot be that an answer set implies $Body$ but does not contain a. The constraint is necessary for preventing a to be ruled out by some other condition occurring elsewhere in the program.

Specifically, the translation can be performed by means of the following guidelines (a full and formal definition of the translation is not possible here for lack of space).

– Sections *COMPLEX_EVENTS* and *RELATED_EVENTS* are basically coped with by the *conj* and *choice* patterns. More involved combinations of events may require the *choyf* and *or-xor* patterns.
– Constraints in the *MANDATORY* section can be expresses by means of the *mand* pattern.
– Sections *CHECK* and *ANOMALIES* can be either translated by a plain transposition of their rules into ASP, or by exploiting the *conj* and *or-xor* patterns.

5.3 Translation Example

We provide below an example of translation, considering the Event-Action module 'diagnosis' that we have presented before. Notice preliminarly that, for each past event evP, it is possible to specify atoms of the form $evP(N, M)$ where M is a unit of time (specifically, M can be seconds, minutes, days) and N is a number of units of time. Such an atom is evaluated by means of a plugin, and returns true (succeeds) in case event ev has been recorded at least once for each of the N time units. E.g., $evP(4, days)$ succeeds whenever event ev has occurred, and has consequently been recorded as a part event, at least once a day for four days. A plugin is also provided for bridge rules: in fact, each atom $p(args) : c$ occurring in the body of such a rule is transformed into $p(args, c)$ and evaluates to true (with suitable instantiations of the arguments) if context c returns the corresponding answer.

Concerning the *Complex Events* section, the translation procedure exploits the **or** pattern for the expression:

$suspect\_flu\ OR\ suspect\_preumonia$

and then just copies the remaining rules of the section, with suitable syntactic rearrangements. The result is the following:

$suspect\_flu$:- $not\ suspect\_pneumonia.$
$suspect\_pneumonia$:- $not\ suspect\_flu.$
$suspect\_flu$:- $high\_temperatureP.$
$suspect\_pneumonia$:- $high\_temperatureP(4, days), intense\_coughP.$
$suspect\_pneumonia$:-
    $diagnosis(clinical\_history, suspect\_pneumonia, diag\_knowledge\_base).$

Translation of the *MANDATORY* section, i.e.:

$suspect\_preumonia$:-
    $high\_temperatureP : [4days], suspect\_fluP, take\_antibioticP : [2days].$

exploits the **mand** pattern, with result

:- $not\ suspect\_preumonia,$
    $high\_temperatureP(4, days), suspect\_fluP, take\_antibioticP : (2, days).$

Rules in the *ACTIONS* section are just copied (modulo minor rearrangements), with result:

$stay\_in\_bedA$:- $suspect\_flu.$
$take\_antibioticA$:- $suspect\_flu,$
    $high\_temperatureP(4, days), not\ suspect\_pneumonia.$
$take\_antibioticA$:- $suspect\_preumonia.$
$consult\_lung\_doctorA$:- $suspect\_preumonia.$

Adapting the notation of [42], the *PREFERENCES* section

$suspect\_flu$:- $patient\_is\_healty.$
$suspect\_pneumonia$:- $patient\_is\_at\_risk.$

would be translated into the *conditional p-lists*:

$suspect\_flu > suspect\_pneumonia$:- $patient\_is\_healty.$
$suspect\_pneumonia > suspect\_flu$:- $patient\_is\_at\_risk.$

The (prototypical) *Raspberry* inference engine [52] would then be able to execute the resulting program, thus returning the preferred answer set. The recent *aspirin* system [38] might also be used.

6 Related Work Concluding Remarks

In this paper we have proposed ACE, as a framework for the design of component-based agent-oriented environments where a "main" agent program, the basic agent, is enriched with a number of Event-Action modules for Complex Event Processing and complex actions generation, and with a number of external data sources that can accessed via bridge rules, borrowed from MCSs. Components of an ACE are in principle heterogeneous, though we assume them to be based upon Computational Logic. The only condition for employing any computational-logic-based language for defining ACE agents or Event-Action modules is that such language must be extended with the possibility of defining bridge rules: this improvement should not however imply either semantic or technical difficulties. We have proposed a formalization and a semantics for ACE. We have also discussed a prototypical experimentation of the approach in the DALI agent-oriented programming language, employing ASP as an implementation tool.

A research work which is related to the present one is DyKnow [53], which is a knowledge processing middleware framework providing software support for creating streams representing high-level events concerning aspects of the past, current, and future state of a system. Input is gathered from distributed sources, can be processed at many different levels of abstraction, and finally transformed into suitable forms to be used by reasoning functionalities. A knowledge process specification is understood as a function. DyKnow is fully implemented, and has been experimented in UAVs (Unmanned Aerial Vehicles) applications.

ACE can be considered as a generalization of such work, in that ACE: (i) is agent-oriented; (ii) is aimed at managing heterogeneity in the definition/description of knowledge sources, that moreover can interact among themselves and with external sources; (iii) is aimed at providing a uniform semantics of single components and of the overall system; (iv) is aimed at allowing for verification of properties.

Several future directions are ahead of us. First, simple preferences are just one possible way of selecting among plausible alternatives. More generally, we plan to consider also informed choice deriving from a learning process: i.e., an agent should learn from experience what is the "best" interpretation to give to a situation, or which are the preference criteria to (dynamically) adopt. Learning should be a never-ending process, as different outcomes might be more plausible in different contexts and situations. Verification of ACE systems is a very relevant aspect to be coped with. We believe that both a priori verification and run-time assurance (cf., e.g., [54]) should be combined for ensuring desirable properties of this kind of systems. Formalization and verification of MASs (Multi-Agent Systems) composed of ACE agents is a further important issue that we intend to consider. ACE agent systems can in principle be part of DACMACSs ("Data-Aware Commitment-based MASs"). The approach of DACMACS, recently proposed in [55, 56] as an extension of DACMAS [57], includes (like in DACMAS) the element of logical ontologies within Multi-Agent systems, but also allows agents of the MAS to query heterogeneous external contexts, possibly with bi-directional interchange of ontological definitions.

References

1. Chandy, M.K., Etzion, O., von Ammon, R.: 10201 Executive Summary and Manifesto – Event Processing. In Chandy, K.M., Etzion, O., von Ammon, R., eds.: Event Processing. Number 10201 in Dagstuhl Seminar Proc., Dagstuhl, Germany, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2011)
2. Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications Co. (2010)
3. Paschke, A., Kozlenkov, A.: Rule-based event processing and reaction rules. In: RuleML. Volume 5858 of Lecture Notes in Computer Science., Springer (2009) 53–66
4. Kowalski, R., Sergot, M.: A logic-based calculus of events. New Generation Computing **4** (1986) 67–95
5. Fisher, M., Bordini, R.H., Hirsch, B., Torroni, P.: Computational logics and agents: a road map of current technologies and future trends. Computational Intelligence Journal **23**(1) (2007) 61–91
6. Anicic, D., Rudolph, S., Fodor, P., Stojanovic, N.: Real-time complex event recognition and reasoning - a logic programming approach. Applied Artificial Intelligence **26**(1-2) (2012) 6–57
7. Anicic, D., Rudolph, S., Fodor, P., Stojanovic, N.: Stream reasoning and complex event processing in ETALIS. Semantic Web **3**(4) (2012) 397–407

8. Costantini, S., Dell'Acqua, P., Tocchio, A.: Expressing preferences declaratively in logic-based agent languages. In: Proceedings of Commonsense'07, the 8th International Symposium on Logical Formalizations of Commonsense Reasoning. AAAI Spring Symposium Series (2007) a special event in honor of John McCarthy.
9. Costantini, S.: Answer set modules for logical agents. In de Moor, O., Gottlob, G., Furche, T., Sellers, A., eds.: Datalog Reloaded: First Intl. Workshop, Datalog 2010. Volume 6702 of LNCS. Springer (2011) Revised selected papers.
10. Costantini, S., De Gasperis, G.: Complex reactivity with preferences in rule-based agents. In Bikakis, A., Giurca, A., eds.: Rules on the Web: Research and Applications - 6th Intl. Symposium, RuleML 2012, Proc. Volume 7438 of Lecture Notes in Computer Science., Springer (2012) 167–181
11. Costantini, S., De Gasperis, G.: Memory, experience and adaptation in logical agents. In Casillas, J., Martínez-López, F.J., Vicari, R., la Prieta, F.D., eds.: Management Intelligent Systems: Second Intl. Symposium, Proc. Advances in Intelligent and Soft Computing, Springer (2013)
12. Costantini, S.: Self-checking logical agents. In Gini, M.L., Shehory, O., Ito, T., Jonker, C.M., eds.: Proceedings of AAMAS 2013, 12th Intl. Conf. on Autonomous Agents and Multi-Agent Systems, IFAAMAS/ACM (2013) 1329–1330 Extended Abstract.
13. Costantini, S., De Gasperis, G.: Meta-level constraints for complex event processing in logical agents. In: Online Proc. of Commonsense 2013, the 11th Intl. Symposium on Logical Formalizations of Commonsense Reasoning. (2013)
14. Kowalski, R.A., Sadri, F.: Reactive computing as model generation. New Generation Comput. **33**(1) (2015) 33–67
15. Kowalski, R.A., Sadri, F.: Teleo-reactive abductive logic programs. In Artikis, A., Craven, R., Cicekli, N.K., Sadighi, B., Stathis, K., eds.: Logic Programs, Norms and Action - Essays in Honor of Marek J. Sergot on the Occasion of His 60th Birthday. Volume 7360 of Lecture Notes in Computer Science., Springer (2012)
16. Costantini, S., Riveret, R.: Event-action modules for complex reactivity in logical agents. In Bazzan, A.L.C., Huhns, M.N., Lomuscio, A., Scerri, P., eds.: Proceedings of AAMAS 2013, 13th Intl. Conf. on Autonomous Agents and Multi-Agent Systems, IFAAMAS/ACM (2014) 1503–1504 Extended Abstract.
17. Costantini, S., Riveret, R.: Complex events and actions in logical agents. In Giordano, L., Gliozzi, V., Pozzato, G.L., eds.: Proceedings of the 29th Italian Conference on Computational Logic. Volume 1195 of CEUR Workshop Proceedings., CEUR-WS.org (2014) 256–271
18. Brewka, G., Eiter, T.: Equilibria in heterogeneous nonmonotonic multi-context systems. In: Proc. of the 22nd AAAI Conf. on Artificial Intelligence, AAAI Press (2007) 385–390
19. Brewka, G., Eiter, T., Fink, M.: Nonmonotonic multi-context systems: A flexible approach for integrating heterogeneous knowledge sources. In Balduccini, M., Son, T.C., eds.: Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday. Volume 6565 of Lecture Notes in Computer Science., Springer (2011) 233–258
20. Brewka, G., Ellmauthaler, S., Pührer, J.: Multi-context systems for reactive reasoning in dynamic environments. In Schaub, T., ed.: ECAI 2014, Proc. of the 21st European Conf. on Artificial Intelligence, IJCAI/AAAI (2014)
21. Brewka, G., Eiter, T., Fink, M., Weinzierl, A.: Managed multi-context systems. In Walsh, T., ed.: IJCAI 2011, Proc. of the 22nd Intl. Joint Conf. on Artificial Intelligence, IJCAI/AAAI (2011) 786–791
22. Costantini, S., Tocchio, A.: A logic programming language for multi-agent systems. In: Logics in Artificial Intelligence, Proceedings of the 8th European Conf., JELIA 2002. LNAI 2424, Springer-Verlag, Berlin (2002)

23. Costantini, S., Tocchio, A.: The DALI logic programming agent-oriented language. In: Logics in Artificial Intelligence, Proc. of the 9th European Conf., Jelia 2004. LNAI 3229, Springer-Verlag, Berlin (2004)
24. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In Kowalski, R., Bowen, K., eds.: Proc. of the 5th Intl. Conf. and Symposium on Logic Programming (ICLP/SLP'88). The MIT Press (1988) 1070–1080
25. Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press (2003)
26. Gelfond, M.: Answer sets. In: Handbook of Knowledge Representation. Elsevier (2007)
27. Truszczyński, M.: Logic programming for knowledge representation. In Dahl, V., Niemelä, I., eds.: Logic Programming, 23rd Intl. Conf., ICLP 2007. (2007) 76–88
28. Apt, K.R., Bol, R.: Logic programming and negation: A survey. The Journal of Logic Programming **19-20** (1994) 9–71
29. Dao-Tran, M., Eiter, T., Fink, M., Krennwallner, T.: Distributed evaluation of nonmonotonic multi-context systems. JAIR, the Journal of Artificial Intelligence Research (2015) To appear.
30. Bordini, R.H., Hübner, J.F.: BDI agent programming in agentspeak using *Jason* (tutorial paper). In Toni, F., Torroni, P., eds.: Computational Logic in Multi-Agent Systems, 6th International Workshop, CLIMA VI, Revised Selected and Invited Papers. Volume 3900 of Lecture Notes in Computer Science., Springer (2006) 143–164
31. Rao, A.S., Georgeff, M.P.: Modeling agents within a BDI-architecture. In Fikes, R., Sandewall, E., eds.: Proc. of Intl. Conf. on Principles of Knowledge Representation and Reasoning (KR), Cambridge, Massachusetts, Morgan Kaufmann (1991)
32. Hindriks, K.V., van der Hoek, W., Meyer, J.C.: GOAL agents instantiate intention logic. In Artikis, A., Craven, R., Cicekli, N.K., Sadighi, B., Stathis, K., eds.: Logic Programs, Norms and Action - Essays in Honor of Marek J. Sergot on the Occasion of His 60th Birthday. Volume 7360 of Lecture Notes in Computer Science., Springer (2012) 196–219
33. Dastani, M., van Riemsdijk, M.B., Meyer, J.C.: Programming multi-agent systems in 3apl. In Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E., eds.: Multi-Agent Programming: Languages, Platforms and Applications. Volume 15 of Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer (2005) 39–67
34. Fisher, M.: MetateM: The story so far. In Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E., eds.: PROMAS. Volume 3862 of Lecture Notes in Computer Science., Springer (2005) 3–22
35. Bordini, R.H., Braubach, L., Dastani, M., Fallah-Seghrouchni, A.E., Gómez-Sanz, J.J., Leite, J., O'Hare, G.M.P., Pokahr, A., Ricci, A.: A survey of programming languages and platforms for multi-agent systems. Informatica (Slovenia) **30**(1) (2006) 33–44
36. Bracciali, A., Demetriou, N., Endriss, U., Kakas, A., Lu, W., Mancarella, P., Sadri, F., Stathis, K., Terreni, G., Toni, F.: The KGP model of agency: Computational model and prototype implementation. In: Global Computing: IST/FET International Workshop, Revised Selected Papers. LNAI 3267. Springer-Verlag, Berlin (2005) 340–367
37. Gebser, M., Grote, T., Kaminski, R., Schaub, T.: Reactive answer set programming. In Delgrande, J.P., Faber, W., eds.: Logic Programming and Nonmonotonic Reasoning - 11th Intl. Conf., LPNMR 2011, Proc. Volume 6645 of Lecture Notes in Computer Science., Springer (2011)
38. Brewka, G., Delgrande, J.P., Romero, J., Schaub, T.: asprin: Customizing answer set preferences without a headache. In Bonet, B., Koenig, S., eds.: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI Press (2015) 1467–1474
39. Bienvenu, M., Lang, J., Wilson, N.: From preference logics to preference languages, and back. In: Proc. of the Twelfth Intl. Conf. on the Principles of Knowledge Repr. and Reasoning (KR 2010). (2010) 414–424

40. Brewka, G., Niemelä, I., Truszczyński, M.: Preferences and nonmonotonic reasoning. AI Magazine **29**(4) (2008)
41. Delgrande, J., Schaub, T., Tompits, H., Wang, K.: A classification and survey of preference handling approaches in nonmonotonic reasoning. Computational Intelligence **20**(12) (2004) 308–334
42. Costantini, S., Formisano, A., Petturiti, D.: Extending and implementing RASP. Fundamenta Informaticae **105**(1-2) (2010) 1–33
43. Costantini, S., Formisano, A.: Modeling preferences and conditional preferences on resource consumption and production in ASP. Journal of of Algorithms in Cognition, Informatics and Logic **64**(1) (2009)
44. Costantini, S., Tocchio, A.: About declarative semantics of logic-based agent languages. In Baldoni, M., Torroni, P., eds.: Declarative Agent Languages and Technologies. Number 3904 in Lecture Notes in Computer Science. Springer (2006) 106–123
45. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–385
46. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Ann. Math. Artif. Intell. **25**(3-4) (1999) 241–273
47. Marek, V.W., Truszczyński, M.: Stable logic programming - An alternative logic programming paradigm. In: 25 years of Logic Programming Paradigm. Springer (1999) 375–398
48. Costantini, S., Formisano, A., Petturiti, D.: Extending and implementing RASP. Fundam. Inform. **105**(1-2) (2010) 1–33
49. Web-references: Some ASP solvers Clasp: `potassco.sourceforge.net`; Cmodels: `www.cs.utexas.edu/users/tag/cmodels`; DLV: `www.dbai.tuwien.ac.at/proj/dlv`; Smodels: `www.tcs.hut.fi/Software/smodels`.
50. Calimeri, F., Ianni, G., Krennwallner, T., Ricca, F.: The answer set programming competition. AI Magazine **33**(4) (2012) 114–118
51. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Computing Surveys **33**(3) (2001) 374–425
52. Formisano, A., Petturiti, D.: Raspberry: an implementation of RASP (2010) URL: `http://www.dmi.unipg.it/~formis/raspberry/`.
53. Heintz, F., Kvarnström, J., Doherty, P.: Bridging the sense-reasoning gap: Dyknow - stream-based middleware for knowledge processing. Advanced Engineering Informatics **24**(1) (2010) 14–26
54. Costantini, S., De Gasperis, G.: Runtime self-checking via temporal (meta-)axioms for assurance of logical agent systems. In Bulling, N., van der Hoek, W., eds.: Proceedings of LAMAS 2014, 7th Workshop on Logical Aspects of Multi-Agent Systems, held at AAMAS 2014. (2014) 241–255 Also in: Proceedings of the 29th Italian Conf. on Computational Logic, CEUR Workshop Proceedings 1195.
55. Costantini, S.: Knowledge acquisition via non-monotonic reasoning in distributed heterogeneous environments. In Truszczyński, M., Ianni, G., Calimeri, F., eds.: 13th Int. Conf. on Logic Programming and Nonmonotonic Reasoning LPNMR 2013. Proc. Volume 9345 of Lecture Notes in Computer Science., Springer (2015)
56. Costantini, S., Gasperis, G.D.: Exchanging data and ontological definitions in multi-agent-contexts systems. In Adrian Paschke, Paul Fodor, A.G.T.K., ed.: RuleMLChallenge track, Proceedings. CEUR Workshop Proceedings, CEUR-WS.org (2015)
57. Montali, M., Calvanese, D., De Giacomo, G.: Specification and verification of commitment-regulated data-aware multiagent systems. In: Proc. of AAMAS 2014. (2014)