Complex Events and Actions in Logical Agents

Stefania Costantini[1] and Regis Riveret[2]

[1] Università di L'Aquila, Italy stefania.costantini@univaq.it
[2] Imperial College London, UK, r.riveret@imperial.ac.uk

Abstract. Complex Event Processing (CEP) has emerged as a relevant new field of software engineering and computer science. Many of the current approaches to CEP are declarative and based on rules, and often on logic-programming-like languages and semantics. Some work on CEP is situated within the field of logical agents. Usually, event processing is based upon Event-Condition-Action rules, which are however able to manage only simple events or unstructured sets of events. In this paper, we refine the notion of "complex event", which is an event that cannot be detected directly, but rather can be identified (not necessarily in a deterministic way) by interrelating sets of simple events. We propose a formalization and a possible implementation of such notion, that we extend to complex actions.

1 Introduction

"Complex Event Processing" (CEP) has emerged as a relevant new field of software engineering and computer science [1]. 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, e.g., to the Proceedings of the RuleML Workshop Series). Complex Event Processing is discussed in depth in [2]. With cloud computing, the importance of event processing is even more visible, and a connection of "event pattern languages" with ontologies and with the semantic web is envisaged. 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. In particular this paper is concerned with logical agents, i.e., agents whose syntax and semantics is rooted in Computational Logic. There are several approaches to logical agents (for a recent survey cf., e.g., [5]). For lack of space, we are not able here to properly discuss and compare their event-processing features. Rather, we recall only the approaches that have more strongly influenced the present work.

A recent but well-established and widely used approach to CEP in computational logic is ETALIS [6–8], which is an open source plug-in for Complex Event Processing

implemented prolog which runs in many prolog systems. 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 causal reasoning and Event Calculus.

In logical agents, some relevant work about CEP is related to DALI, a logic agent-oriented language (first appeared in 1999 [9]) that extends prolog with reactive and proactive features [9–12] and is fully implemented and publicly available [13]. DALI, like virtually all agent-oriented languages, is equipped with "event-condition-action" rules (ECA rules) for defining the behavior of an agent in consequence of perception of external events. The feature that makes DALI proactive and strongly event-oriented is however the *internal events* construct: i.e., the programmer can indicate internal conditions to be interpreted as events, to which a reaction can be defined. Management of events which occur "together" (with the possibility to specify in which time interval), priorities between events, and aggregation of simple events into complex ones via the internal event construct are other useful features. Related to (the pre-existing) DALI approach is for instance the work of [14], also providing ECA rules and a "snapshot" semantics similar to the one already introduced in [12]. Work on CEP in DALI is presented in [15] and [16], which discuss the issue of of selecting different reactive patterns by means of simple preferences, then extended to more complex forms of preferences in [17]. 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. [18] and [19, 20] discuss event-based memory-management, and temporal-logic-based constraints for complex dynamic self-checking and reaction.

Teleo-Reactive Computing by Kowalski and Sadri [21–23] 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. 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.

The motivation of the present work relies in the observation that a complex event cannot always be defined and recognized from simple deterministic incremental aggregation of simple events. Rather, complex events sometimes require a more involved and not necessarily deterministic description. In fact, in order to be flexible and versatile instead of brittle and rigid an agent should be able to possibly interpret a set of simple events in different ways, and to assign/learn the plausibility and reliability of each interpretation (for a discussion of brittleness w.r.t. versatility in agents, cf. [24–26]). For instance, in diagnosis a number of symptoms, if occurring together, might lead to hypothesize the presence of one or more illness/fault, possibly with some certainty/probability. Another reason why event patterns should be carefully described is to

make an agent able to detect unknown events or wrong patterns, and take the necessary counter-measures (see, e.g., [19, 20]).

In this paper, we propose a novel conceptual view of complex events and a possible formalization of the new concepts. In our view, an agent should be able to possibly interpret a set of simple events in different ways, and to assign/learn a plausibility and reliability of each interpretation. We also consider complex actions, which we consider as agent-generated events. To this aim, we propose to equip agents with specific modules, that we call *Event-Action modules*, describing complex events and complex actions. Such modules receives as "input" a set of simple events, either perceived by the agent at its present stage of operation (external events), or generated internally by the agent itself. Each module returns: (i) possible interpretations of a set of simple events in terms of complex events; (ii) an ordering of such interpretations (if more than one is possible) in terms of preferences and/or of certainty factors; (iii) detection of anomalies; (iv) (sets of) actions to perform in response. Modules are (automatically) re-evaluated whevener new instances of the "triggering" events become available.

From a practical point of view, for both providing a formal semantics and an implementation of modules we have presently adopted Answer Set Programming (ASP, cf., among many, [27, 28]). ASP 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. So, in the proposed framework, a logical agents can be seen as composed of a "main program" including event processing, planning, action making, etc. and a set of Event-Action modules (implemented in ASP) for event/action recognition, generation, aggregation and management. We propose to exploit, in order to discern among alternatives, both preferences and probabilities, refined via reinforcement learning. The approach is encompassed into the general declarative semantics for logical agents introduced in [12] (and summarized below). This makes our proposal immediately applicable to many agent-oriented logic languages/framework.

To define Event-Action modules we presently refer to a quite simple logical setting which can be in fact translated into ASP. However, in principle such modules might be based upon more expressive logics (e.g., modal and/or probabilistic logics which would certainly be suitable to express event recognition/aggregation). The structure of our proposal is such that the basic concept, i.e, modules for (non-deterministic, defeasible) reasoning about event aggregation, might be easily rephased upon another logic. Clearly, in such case a suitable semantic and execution model (other than ASP) should then be provided.

The paper is organized as follows. In Section 2 we provide some background about the building blocks of the proposed apprach. In Section 3 we present the proposal, and in Sections 4 and 5 its formalization, and some considerations about learning. In Section 6 we conclude. This paper is the extended version of [29].

2 Background

In this section, we recall notions that define some basic foundation elements of the proposed approach.

2.1 Evolutionary Semantics of Logical Agent-Oriented Languages

The Evolutionary semantics (introduced in [12]) is meant to provide a high-level general account of evolving agents, trying to abstract away from the details of specific agent-oriented frameworks. As seen below, the Evolutionary semantics in fact generalizes the specific semantic approaches underlying well-known logical agent-oriented languages. Actually, the Evolutionary semantics can be seen as a meta-semantics, as it accounts for agent evolution in terms of transformation steps. The precise definition of an agent and how a transformation step is determined and described attains to the specific agent-oriented language.

We define in fact, in very general terms, an agent as the tuple $Ag = < P_{\mathcal{A}}, E >$ where $\mathcal{A}$ is the agent name and $P_{\mathcal{A}}$ (that we call "agent program", but can be in turn a tuple) describes the agent according to some agent-oriented formalism $\mathcal{L}$. E is the set of the events that the agent is able to recognize or determine (so, E includes actions that the agent is able to perform), according to the specific agent-oriented framework.

Let H be the *history* of an agent as recorded by the agent itself (in a form that will depend upon the specific agent-oriented framework), i.e., H includes agent's perceptions and *memories*. We assume that events that either happen externally or are generated internally, actions which are performed, and other relevant agent's activities are recorded in H.

We assume that program $P_{\mathcal{A}}$ as written by the programmer is in general transformed into an initial agent program P_0 by means of an *initialization phase* (possibly doing nothing). When agent $\mathcal{A}$ is activated P_0 will go into execution, and will evolve according to the evolution of H.

Evolution is represented via program-transformation steps, each one transforming P_i into P_{i+1} according to H_i, which is the partial history up to stage i. The choice of which elements of H_i do actually trigger an evolution step is part of the definition of a specific agent framework.

Thus, we obtain a Program Evolution Sequence $PE = [P_0, \ldots, P_n, \ldots]$. The program evolution sequence will imply a corresponding Semantic Evolution Sequence $ME = [M_0, \ldots, M_n, \ldots]$ where M_i is the semantics of P_i according to $\mathcal{L}$. Notice in fact that the approach is parametric w.r.t $\mathcal{L}$.

Definition 1 (Evolutionary semantics). *Let Ag be an agent. The evolutionary semantics ε^{Ag} of Ag is a tuple $\langle H, PE, ME \rangle$, where H is the history of Ag, and PE and ME are respectively its program and semantic evolution sequences.*

The next definition introduces the notion of instant view of ε^{Ag}, at a certain stage of the evolution (which is in principle of unlimited length).

Definition 2 (Evolutionary semantics snapshot). *Let Ag be an agent, with evolutionary semantics $\varepsilon^{Ag} = \langle H, PE, ME \rangle$. The* snaphot at stage i *of ε_i^{Ag} is the tuple $\langle H_i, P_i, M_i \rangle$, where H_i is the history up to the events that have determined the transition from P_{i-1} to P_i.*

In [12], program transformation steps associated with DALI language constructs [10, 11] are defined in detail. They can easily be adapted to AgentSpeak (cf. [30] and

the references therein). The evolutionary semantics may also express the notion of *trace* of a GOAL agent (cf. [31] and the references therein) where agent program P_i encompasses the agent's *mental state* and each evolution step, which in GOAL is called *computation step* and is determined by a *conditional action*. For 3APL (cf. [32] and the references therein), agent program P_i encompasses the agent's *initial configuration*, and the related sets GR of *goal rules*, PR of *plan rules*, IR of *interactive rules*; the evolutionary semantics corresponds to a 3APL agent *computation run*, and evolution steps are determined by the 3APL *transition system*.

In Section 4.1 we provide an integration of the proposed approach to Complex Event Processing into the Evolutionary semantics, in order to make it applicable to many agent-oriented logic programming languages among which the above-mentioned ones.

2.2 Background on Answer Set Semantics

The answer set semantics is used in this paper to provide both a formal semantics and an execution model for the proposed modules. In fact, this logic gives rise to answer set programming (ASP), which is a very well-established and fully logical programming paradigm. Many efficient solvers have become freely available for ASP, [33], each one proposing many extension to aid practical programming.

In the answer set semantics (originally named "stable model semantics"), a (logic) program Π (cf., [34]) is a collection of *rules* of the form $H \leftarrow A_1, \ldots, A_n, not\, B_1, \ldots, not\, B_m$ where the A_is ($i \leq n$ and the B_js ($j \leq m$ are atoms, and in literals $not\, B_j$ not is *default negation*. A rule with empty body is called a *fact*. A rule with empty head is a *constraint*, where a constraint $\leftarrow L_1, \ldots, L_n.$ states that literals $L_1, \ldots, L_n$ cannot be all true in an answer set.

The answer sets semantics is a view of a logic program as a set of inference rules (more precisely, default inference rules), or, equivalently, a set of constraints on the solution of a problem: each answer set represents a solution compatible with the constraints expressed by the program. In simple program $\{q \leftarrow not\, p \;\; p \leftarrow not\, q\}$, the first rule is read as "assuming that p is false, we can *conclude* that q is true." This program has two answer sets. In the first one, q is true while p is false; in the second one, p is true while q is false. Thus, this program represent choice among two alternatives.

In the ASP (Answer Set Programming) paradigm, each answer set is seen as a solution of given problem, encoded as an ASP program. An ASP program is basically composed by program fragments like the above ones that generate the search space by defining available alternatives, and by constraints which prune the generated space thus selecting solutions. To find these solutions, one employes one of the several existing ASP solvers [33]. The expressive power of ASP and its computational complexity have been deeply investigated (cf., e.g., [35]).

3 Complex Event Processing in Logical Agent Languages

In this section we present our approach to advanced Complex Event Processing in logical agents. First, we recall how event processing and aggregation can be performed in most existing frameworks. Then, we argue that new methods and tools would be useful, and introduce our proposal.

3.1 Complex Events Recognition and Evaluation: Simple Event Aggregation

Event processing is traditionally based upon Event-Condition-Action (ECA) rules, of the form "IF Event then Reaction" where sometimes reaction is limited to performing a sequence of actions, sometimes is enlarged so as to allow forms of reasoning and various kinds of constraints. Most (virtually all) agent-oriented languages provide ECA rules of some form.

For instance, the following sample ECA rules defined in DALI evaluate medical symptoms, stating that to cope with high temperature one should assume an antipyretic, and to cope with cough one should take a syrup (clearly, we do not aim at medical precision). External events are indicated with postfix E, actions with postfix A, and connective $:>$ indicates reaction. I.e., $:>$ is a specific connective used to define ECA rules. Since DALI is an extension of prolog, the comma represents conjunction.
$high\_temperatureE :> take\_antipyreticA.$
$intense\_coughE :> take\_cough\_syrupA.$

Plain ECA rules can hardly provide more than this kind of simple reaction. Agent languages may offer empowering mechanisms that allow for some form of CEP. In DALI for instance, by means of the "internal event" construct evaluation can become more involved. In the example we may, e.g., consider the possibility that a combination of symptoms suggests the occurrence of a more serious illness. For illustration purposes we assume to hypothesize the presence of either pneumonia, or just flu, or both. In DALI external events are, a short while after perception (that may or may not imply reaction), recorded as past events (postfix P), with a time-stamp. By $high\_temperatureP : [4days]$ it is intended that at least two past events of the form $high\_temperatureP : T1$ and $high\_temperatureP : T2$ have been recorded, where $T2 - T1$ is no less than four days. By default, the most recent records are considered, so the interval refers to the last and to the immediately previous measures of temperature (if the time-stamp of a past event is omitted, the most recent version is implicitly referred to).

The first rule below "reasons" about what has happened: if in fact both high temperature and cough have occurred, and high temperature has lasted for at least four days, one may conclude that occurrence of pneumonia should be suspected. Therefore, from the occurrence of a set of simple events, the occurrence of a complex event is inferred, as an internal event $suspect\_pneumoniaI$. Once inferred, such an event can be reacted to exactly like external ones by an ECA rule, the one below stating that the patient shoul take an antibiotic and consult a lung doctor.
$suspect\_pneumoniaI$:-
$high\_temperatureP : [4days],$
$intense\_coughP,$
$compatible(prenumonia, anamnesis).$
$suspect\_pneumoniaI :> take\_antibioticA,$
$consult\_lung\_doctorA.$

Internal events are periodically re-evaluated at a certain (customizable) frequency. Between one attempt and the next one, the agent's belief base will evolve as new events will have happened, reacted to and recorded. Thus, at each attempt an internal event can be either generated (and reacted to) or not, according to the agent's belief state. In

the example, the suspect of pneumonia will not raise immediately, but rather after four days of high temperature measurements.

Clarly, the proposed example might be formalized also in ETALIS and in many other agent-oriented frameworks, each one offering its own improvements over simple ECAs in view of CEP. In summary, by means of ECA rules little can be done for CEP. By means of suitable empowring mechanisms such as the internal events, complex events can posibly be generated (ad then reacted to) by reasoning on simple external events.

We believe however that such a simple pattern is often not sufficient, as there are many cases where the interpretation of sets of simple events is not univocal or straightforward. Therefore, different hypotheses about the meaning of a situation at hand should be generated and evaluated. We then propose a novel view and a novel formalization of event aggregation.

3.2 Complex Events Recognition and Evaluation: Event-Action Modules

Below we introduce the notion of Event and Action modules. For lack of space we illustrate these modules by means of two examples that we consider as representatives of significant situations. The first example concerns complex event recognition, the second one concerns complex actions generation. We do not intend to propose here a specific syntax for Event and Action modules: rather, we intend to point out what should be the elements and functions of such modules. In the sample syntax, adopted only for illustration purposes, as before postfix 'P' indicates *past events*, i.e., events which have occurred (after the ':' there is the time-stamp or the interval of occurrence). Postfix 'A' indicates *actions*. Connective :> indicated Event-Condition-Action rules, while :- is the usual prolog *if*.

Event Interpretation

The following example illustrates an *Event-Action Module* that re-elaborates the previosly-introduced example concerning medical diagnosis. This should allow the reader to appreciate the improvements over the simple formulation.

An Event-Action Module will be (re-)evaluated whenever the *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.

EVENT-ACTION-MODULE

$TRIGGER$
$(high\_temperatureP\ AND\ intense\_coughP) : [2days]$
$COMPLEX\_EVENTS$
$suspect\_flu\ OR\ suspect\_preumonia$

$suspect\_flu$:- $high\_temperatureP.$
$suspect\_pneumonia$:- $high\_temperatureP : [4days],$
$intense\_coughP.$

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

$ACTIONS$
$suspect\_flu :> stay\_in\_bed A$.
$suspect\_flu, high\_temperature P : [4days], not\ suspect\_pneumonia :>$
$take\_antibiotic A$.
$suspect\_preumonia :>$
$take\_antibiotic A, consult\_lung\_doctor A$.

$MANDATORY$
$suspect\_preumonia$:- $high\_temperature P : [4days]$,
$suspect\_flu P$,
$take\_antibiotic P : [2days]$.

From given symptoms, either a suspect flu or a suspect pneumonia or both can be concluded, though for suspecting pneumonia high temperature should have lasted for at least four days, accompanied by intense cough. This is stated in the *COMPLEX_EVENTS* section, where each of the listed complex events (in the example, *suspect_flu* and *suspect_pneumonia*) can be inferred, though according to the specified conditions. Notice that the whole agent's belief base (including the history) is implicitly included in the definition of an Event-Action module. Explicit preferences are expressed in the *PREFERENCES* section, here stating that hypothesizing a flu should be preferred in case the patient is healthy, while pneumonia is more plausible for risky patients. If either none or both options hold, then they are equally preferred. More involved forms of preferences might be specified, that for lack of space we do not discuss here (cf., e.g., [17] and the references therein). Actions to undertake in the two cases are specified. As mentioned, the module is re-evaluated at subsequent new occurrences of high temperature and intense cough. Re-evaluation is performed on the (possibly) updated belief base.

Actions will be actually performed depending upon the outcome that the agent will prefer to choose. In particular, 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.

Answer set modules for possibility and necessity (cf. [16]) find a fruitful integration in the present approach. In the example, it may be for instance that clinical history and conditions of a patient force to assume a pneumonia. A constraint such as the following might be added:

$MANDATORY\ suspect\_pneumonia$:-
$Necessary(clinical\_history, suspect\_pneumonia)$.

Generating Complex Actions

We have noted before that event interpretation is not necessarily deterministic and straightforward. The same happens for actions: devising which actions and agent should perform can be highly context-dependent, and can be subjected to various kinds of uncertainty. To avoid brittleness, an agent should in our opinion be able to flexibly choose what to do in specific circumstances, and to dynamically adapt to changes of context/role/situation. This is why we propose to adapt previously introduced modules to action generation.

We illustrate the approach by means of an example related to what happens when two persons meet. The representation is simplicistic and is meant only to illustration purposes. A "real" encoding of the module below might be much more involved, and imply various kinds of knowledge representation methods, possibly including a theory of emotions.

In the sample representation, we assume that the person/agent who first gets sight of a person/agent (s)he knows possibly 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. There are some conditions, for instance that one may smil and/or waves if (s)he is neither in a bad temper nor angry at the other. 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. 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). In the formalization below, the expression $meet\_friend(A, F)$ means that agent A meets agent F: then, each one possibly makes 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 (left undefined here) 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

$TRIGGER\ meet\_friend(A, F),$
$RELATED\_EVENTS$
$smile(A, F)\, OR\ (wave(A, F)\ XOR\ shake\_hands(A, F))$

$PRECONDITIONS$
$smileA(A, F) :< not\ angry(A, F), not\ bad\_temper(A).$
$waveA(A, F) :< not\ angry(A, F).$
$shake\_handsA(A, F) :<$
$\quad good\_friends(A, F),$
$\quad not\ angry(A, F), not\ in\_a\_hurry(A), not\ in\_a\_hurry(F).$

$MANDATORY$
$shake\_handsA(A, F)$:- $formal\_situation(A, F).$

$ACTIONS$
$smiled(A, F)$:- $smileP(A, F).$
$waved(A, F)$:- $waveP(A, F).$
$shaken\_hands(A, F) :> shake\_handsP(F, A).$
$smiled(A, F) :> smileA(F, A).$
$waved(A, F) :> waveA(F, A).$
$shaken\_hands(A, F) :> shake\_handsA(F, A).$

$ANOMALY$
$anomaly1(meet\_friend(A, F))$:-
$\quad smileP(A, F), not\ smileA(F, A).$
$anomaly2(meet\_friend(A, F))$:-
$\quad waveP(A, F), not\ waveA(F, A).$
$anomaly3(meet\_friend(A, F))$:-
$\quad shake\_handsP(A, F), not\ shake\_handsA(F, A).$
$ANOMALY\_MANAGEMENT\_ACTIONS$
...

4 Formalization of Event-Action Modules

In this section we provide a formalization of Event-Action modules based upon previously-introduced building blocks. To ensure integration of modules within the most common agent-oriented programming languages, we suitably merge them into the Evolutionary semantics. To provide (as a proof of concept) a formal semantics and an execution model, we define a translation of Event-Action modules into ASP modules.

4.1 Evolutionary Semantics Extended

We now briefly illustrate how to refine the Evolutionary semantics so as encompass the proposed approach. The agent program $P_{\mathcal{A}}$ becomes now a tuple including at least the "main" agent program $P_{\mathcal{M}}$, and the available Event-Action modules $EA_1, \ldots, EA_k$.

Definition 3 (Evolutionary Semantics). *Let Ag be an agent, defined by agent program*

$$P_{\mathcal{A}} = \langle P_{\mathcal{M}}, \langle EA_1, \ldots, EA_k \rangle, \ldots \rangle$$

where $P_{\mathcal{M}}$ is the main agent program, and $EA_1, \ldots, EA_k$, $k \geq 0$, are the available Event-Action modules.

Its evolutionary semantics is $\varepsilon^{Ag} = \langle H, PExt, ME \rangle$. *The program evolution sequence, indicated by* $PExt$, *stems from an* Extended initial program $PExt_0$ *obtained from* $P_{\mathcal{M}}$ *by means of a a suitable initialization phase. In particular,* $PExt_0$ *includes the following elements:*

- *Program* P_0*, obtained by adding to given main agent program the rules included in sections* ACTIONS*,* ANOMALY_MANAGEMENT_ACTIONS *and* PRECONDITIONS *of* $EA_1, \ldots, EA_k$.
- *Tuple of ASP modules* $\langle \Pi_1, \ldots, \Pi_k \rangle$*, where each* Π_i *is obtained by translating Event-Action module* EA_i *into ASP.*

At the i-th evolution step, the agent's history in general evolves, as new events/knowledge items may be recorded/removed. This determines an evolution of both the main agent program P_i and of the ASP modules: in fact, the main program and the modules implicitly encompass the agent's history as the set of given facts. This implies that both the main program and modules semantics needs to be re-evaluated at each step. Each module will admit none, one or several answer sets, among which just one has to be selected. Such answer set will encompass a number of inferred complex events and actions, as well as possibly a number of anomalies, that have to be added to the agent's history in order to be respectively reacted to (for events) or executed (for actions) or coped with (for anomalies). Precisely, at each step we have:

Definition 4 (Evolutionary Semantics Snapshot). *The* snaphot at stage i *of* ε_i^{Ag} *is the tuple* $\langle H_i, PExt_i, M_i \rangle$*, where* M_i *is in turn a tuple, i.e.,*

$$M_i = \langle F_i, S_1, \ldots, S_r \rangle$$

where F_i *is the semantics of* P_i*, and* S_i *(*$i \leq r \leq k$*) is the set composed of the answer sets of each* Π_i *which has been (re-)evaluated at stage* i*. The evolution step to stage* $i+1$ *will imply: (i) the choice of one answer set* A_i *for each* Π_i *(selected among the elements of* S_i*); (ii) and the addition to* H_{i+1} *of all the complex events and actions and anomalies included in* A_i.

Therefore, at step $i+1$, complex events, actions and anomalies generated at step i by each ASP module will be coped with as specified in the original corresponding Event-Action module, and then recorded as past events in the agent's history.

In summary, the integration of Event-Action modules within a logical agent's basic functioning can be described as follows.

- At every agent's evolution step, the *TRIGGER* headline of each Event-Action module has to be checked, in order to state whether the module is to be re-avaluated. We cannot treat here complexity of this check, that will depend upon complexity of expressions involved.
- ASP modules corresponding to Event-Action modules that are to be (re-)evaluated will be fed to an answer set solver. A module will admit as a result of evaluation none, one or more answer sets.

– If the module admits answer sets, each one will encompass a number of complex events, actions and anomalies. One answer set will be selected, according to preferences (if expressed within the module), or by random choice, or by informed choice deriving from a learning process, as discussed in Section 5. The rules for coping with the inferred complex events, actions and anomalies are defined in sections *ACTIONS*, *ANOMALY_MANAGEMENT_ACTIONS* and *PRECONDITIONS*, which can be found in the main agent program.

4.2 ASP Representation of Modules

Each Event-Action module can be translated in a fully automated way into an ASP module. This except sections *ACTIONS*, *ANOMALY_MANAGEMENT_ACTIONS* and *PRECONDITIONS*, whose content as seen before has to be added to the main agent program.

The answer set program (module) Π corresponding to a given Event-Action module will be obtained by translating into ASP the contents of sections *COMPLEX_EVENTS*, *CHECK*, *RELATED_EVENTS*, *ANOMALIES* and *MANDATORY*. The translation is straightforward and can be fully automated.

For the sake of simplicity we outline a translation into basic ASP, thus not considering the various additional useful constructs that have been introduced in the literature and in the implementations. It can be noted that extensions of ASP oriented to stream reasoning have been defined (cf., e.g., [36]). We observe however that we do not perform stream reasoning, rather we perform periodical re-evaluation of modules, so at present we do not deem it necessary to resort to such approach.

For lack of space we cannot properly describe how to cope with time intervals: however, our method consists in representing time-stamps as additional arguments of ASP predicates representing events. Intervals are then computed by trivial arithmetic constraints.

The translation of (the relevant sections of) an Event-Action module into ASP can be performed by means of the following guidelines (a formal definition of the translation, including the treatment of time intervals, is deferred to an extended version of this paper).

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 among two elements a and b is expressed by the cycle $a \leftarrow b \;\; b \leftarrow a$. 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....* In the case of exclusive disjunction, it cannot be that both a and b belong to the same answer set. Disjunction (also exclusive) can be expressed also 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 $a \leftarrow na \;\; na \leftarrow a$. Therefore, an answer set will contain either a or na, the latter signifying the absence of a.

choyf A choice that can be done (by pattern *choice*) on element a 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.

- Events in the *RELATED_EVENTS* section can be expressed by means of the *choice* pattern, and their combinations via the *conj* and *or-xor* patterns. Constraints in the *MANDATORY* can be expresses by means of the *mand* pattern.
- Section *COMPLEX_EVENTS* is coped with by the *choice* and *choyf* patterns.
- Sections *CHECK* and *ANOMALIES* can be translated by a plain transposition of their rules into ASP, possibly exploiting the *conj* and *or-xor* patterns.

5 Evaluating Outcomes and Reinforcement Learning

Outcomes of an Event-Action Module are often not univocal: thus, at each stage the agent face a choice among the answer sets of corresponding ASP module Π. As we have seen in the example, preferences may help the agent in choosing an outcome rather than another one. However, knowledge can be incomplete or partial about reliability of such preferences, and in general about plausibility of the choice. This choice is akin to the "multi-armed bandit problem" [37], and thus machine learning techniques can be used so that an agent will learn over time to make the 'best' choice over the answer sets of each module Π (i.e., over the outcomes of each Event-Action Module).

Therefore, a self-improving process is in order, and for this purpose, an agent can be endowed with a simple reinforcement learning mechanism [38]: at each evolution step, occurring at a time t, an agent: (i) senses its environment; (ii) re-evaluates Π obtaining the answer sets $A_1, \ldots, A_k$, $k > 0$; (iii) adopt one of them, say A, and evaluates its "quality", denoted with $Q(A)$. In order to evaluate the quality of an answer set, we assume that at stage t, a numerical value denoted $V^t(A)$ is associated to it: this value can be either epistemic or practical. Its expression shall be dependent on the application so we leave it unspecified in the description of the present general framework. At time t, the quality of the selected answer A will be computed as a discounted moving average of its value over time:

$$Q^{t+1}(A) = Q^t(A) + \alpha.(V^t(A) - Q^t(A))$$

where α is the discount rate. Then, an agent draws an answer set A amongst all the possible answer sets $A_1, \ldots, A_k$ of Π with probability $P^t(A)$. This probability can be computed using a Gibbs-Boltzmann distribution:

$$P^t(M) = e^{Q^t(A)/\tau} / \sum_i e^{Q^t(A_i)/\tau}$$

where τ is a 'learning temperature' to balance the exploitation and the exploration of possible models.

6 Concluding Remarks

In summary, in this paper we have introduced kinds of Event-Action modules that allow an agent to: aggregate simple events into complex ones, also according to constraints; check events that occur w.r.t. expectations; cope with events possibly occurring contextually to certain other events; detect anomalies; decide actions to be performed in both normal or anomalous cases, according to a number of issues among which we may include context, role, circumstances, past experience, etc.

In future work, other event aggregation and recognition patterns might be introduced. The approach might be extended to a more involved definition of complex actions, and to the choice among possible action patterns. The proposed simple learning mechanism might be refined based on experimentations on suitable test cases. We may notice that the approach is basically independent of the underlying logic, so that more expressive (e.g., modal) logic might be employed in future extensions. Clearly, a semantics and execution model going beyond ASP should then be provided.

We intend to introduce forms of 'deep' learning of Event-Action modules. In our intention, an Event-Action Modules might be initially defined in a tentative or embryonic form: then, module elements might be learnt via a training phase, and refined via reinforcement learning during the agent's 'life' thus adding to agent's flexibility and adaptability. To this aim, we have been developing suitable argomentation-based learning techniques [39].

We do not intend to claim that all events can be recognized and reacted to, and all actions generated, only in a logic-based way. Nevertheless, also based upon relevant literature, we claim that event/actions recognition, generation and management oftens require forms of (even non-trivial) reasoning. Overall, any really "intelligent" agent capable of flexible interaction with complex real-world environment will presumably result from a graceful integration of several kind of components, possibly based upon different approaches.

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 Proceedings, 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. : Etalis web site. `http://code.google.com/p/etalis/`
9. Costantini, S.: Towards active logic programming. In Brogi, A., Hill, P., eds.: Electronic Proceedings of COCL'99, Second Intl. Works. on Component–Based Software Development in Computational Logic (included in PLI'99, Principles, Logics and Implementation of High–level Programming Languages). On-line at the URL: `http://www.di.unipi.it/~brogi/ResearchActivity/COCL99/proceedings/index.html`, year = 1999.
10. Costantini, S., Tocchio, A.: A logic programming language for multi-agent systems. In: Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf.,JELIA 2002. LNAI 2424, Springer-Verlag, Berlin (2002)
11. Costantini, S., Tocchio, A.: The DALI logic programming agent-oriented language. In: Logics in Artificial Intelligence, Proc. of the 9th European Conference, Jelia 2004. LNAI 3229, Springer-Verlag, Berlin (2004)
12. Costantini, S., Tocchio, A.: About declarative semantics of logic-based agent languages. In Baldoni, M., Endriss, U., Omicini, A., Torroni, P., eds.: Declarative Agent Languages and Technologies III, Third International Workshop, DALT 2005, Selected and Revised Papers. Volume 3904 of LNAI. Springer (2006) 106–123
13. Costantini, S., D'Alessandro, S., Lanti, D., Tocchio, A., al.: DALI web site, download of the interpreter (2012) Released: basic DALI features. For beta versions please ask the authors.
14. Alferes, J.J., Banti, F., Brogi, A.: An event-condition-action logic programming language. In: Logics in Artificial Intelligence, 10th European Conference, JELIA 2006. Volume 4160 of Lecture Notes in Computer Science., Springer (2006) 29–42
15. Costantini, S., Dell'Acqua, P., Tocchio, A.: Expressing preferences declaratively in logic-based agent languages. In: Proc. of Commonsense'07, the 8th International Symposium on Logical Formalizations of Commonsense Reasoning, AAAI Press (2007) Event in honor of the 80th birthday of John McCarthy.
16. Costantini, S.: Answer set modules for logical agents. In de Moor, O., Gottlob, G., Furche, T., Sellers, A., eds.: Datalog Reloaded: First International Workshop, Datalog 2010. Volume 6702 of LNCS. Springer (2011) Revised selected papers.
17. 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, RuleML 2012 - Europe, Montpellier, France, August 27-29, 2012. Proceedings. Volume 6826 of Lecture Notes in Computer Science., Springer (2012) 167–181
18. Costantini, S., Gasperis, G.D.: 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 International Symposium, Proceedings. Advances in Intelligent and Soft Computing, Springer (2013)
19. Costantini, S.: Self-checking logical agents. In Gini, M.L., Shehory, O., Ito, T., Jonker, C.M., eds.: International conference on Autonomous Agents and Multi-Agent Systems, AAMAS 2013, Proceedings, IFAAMAS (2013) 1329–1330
20. Costantini, S., Gasperis, G.D.: Meta-level constraints for complex event processing in logical agents. In: Online Proceedings of Commonsense 2013, the 11th International Symposium on Logical Formalizations of Commonsense Reasoning. (2013)
21. Kowalski, R.A., Sadri, F.: Towards a logic-based unifying framework for computing. CoRR **abs/1301.6905** (2013)

22. 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)
23. Kowalski, R.A., Sadri, F.: A logic-based framework for reactive systems. In: Rules on the Web: Research and Applications - 6th International Symposium, RuleML 2012. Proceedings. Volume 7438 of Lecture Notes in Computer Science., Springer (2012)
24. Brachman, R.J.: (AA)AI more than the sum of its parts. AI Magazine **27**(4) (2006) 19–34
25. Anderson, M.L., Perlis, D.: Logic, self-awareness and self-improvement: the metacognitive loop and the problem of brittleness. J. Log. Comput. **15**(1) (2005) 21–40
26. Anderson, M.L., Fults, S., Josyula, D.P., Oates, T., Perlis, D., Wilson, S., Wright, D.: A self-help guide for autonomous systems. AI Magazine **29**(2) (2008) 67–73
27. Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press (2003)
28. Gelfond, M.: Answer sets. In: Handbook of Knowledge Representation, Chapter 7. Elsevier (2007)
29. Costantini, S., Riveret, R.: Event-action modules for complex reactivity in logical agents. In Lomuscio, A., Scerri, P., Bazzan, A., Huhns, M., eds.: International conference on Autonomous Agents and Multi-Agent Systems, AAMAS 2014, Proceedings, IFAAMAS (2014) 1503–1504 Extended Abstract.
30. Bordini, R.H., Hübner, J.F.: Semantics for the jason variant of agentspeak (plan failure and some internal actions). In Coelho, H., Studer, R., Wooldridge, M., eds.: ECAI 2010 - 19th European Conference on Artificial Intelligence, Proceedings. Volume 215 of Frontiers in Artificial Intelligence and Applications., IOS Press (2010) 635–640
31. Hindriks, K.V.: Programming rational agents in GOAL. In El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H., eds.: Multi-Agent Programming:. Springer US (2009) 119–157
32. Dastani, M., van Riemsdijk, M.B., Meyer, J.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
33. 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`.
34. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Logic Programming, Proc. of the Fifth Joint Int. Conf. and Symposium, MIT Press (1988) 1070–1080
35. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and expressive power of logic programming. ACM Computing Surveys **33**(3) (2001) 374–425
36. Gebser, M., Grote, T., Kaminski, R., Obermeier, P., Sabuncu, O., Schaub, T.: Stream reasoning with answer set programming: Preliminary report. In Brewka, G., Eiter, T., McIlraith, S.A., eds.: Principles of Knowledge Representation and Reasoning: Proceedings of the Thirteenth International Conference, KR 2012, year = 2012
37. Cesa-Bianchi, N., Lugosi, G.: Prediction, Learning, and Games. Cambridge University Press, New York, NY, USA (2006)
38. Sutton, R., Barto, A.: Reinforcement learning: An introduction. Volume 116. Cambridge Univ Press (1998)
39. Caianiello, P., Costantini, S., Riveret, R., Draief, M.: Concept learning by a monte-carlo tree search of argumentations. In: Proc. of RCRA2014, 21st RCRA International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion, Part of the 2014 Vienna Summer of Logic. (2014) To appear.