# An Event-Condition-Action Logic Programming Language

J. J. Alferes[1], F. Banti[1], and A. Brogi[2]

[1] CENTRIA, Universidade Nova de Lisboa, Portugal,
jja|banti@di.fct.unl.pt
[2] Dipartimento di Informatica, Università di Pisa, Italy,
brogi@di.unipi.it

**Abstract.** Event-Condition-Action (ECA) languages are an intuitive and powerful paradigm for programming reactive systems. Usually, important features for an ECA language are reactive and reasoning capabilities, the possibilities to express complex actions and events and a declarative semantics. In this paper, we introduce ERA, an ECA language based on the framework of logic programs updates that, together with these features, also exhibits capabilities to integrate external updates and perform self updates to its knowledge (data and classical rules) and behaviour (reactive rules).

## 1 Introduction

Event Condition Action (ECA) languages are an intuitive and powerful paradigm for programming reactive systems. The fundamental construct of ECA languages are *reactive rules* of the form:

$$\textbf{On } Event \ \ \textbf{If } Condition \textbf{ Do } Action \tag{1}$$

which mean: when $Event$ occurs, if $Condition$ is verified, then execute $Action$. ECA systems receive inputs (mainly in the form of *events*) from the external environment and react by performing actions that change the stored information (internal actions) or influence the environment itself (external actions). There are many potential and existing areas of applications for ECA languages such as active and distributed database systems [33, 10], Semantic Web applications [2, 3, 28], distributed systems [19], Real-Time Enterprize and Business Activity Management and agents [16].

To be useful in a wide spectrum of applications an ECA language has to satisfy several properties. First of all, events occurring in a reactive rule can be complex, resulting by the occurrence of several basic ones. A general and widely used way for defining complex events is to rely on some event algebras [15, 4], i.e. to introduce operators that define complex events as the result of compositions of more basic events that occur at the same or at different instants. Also the actions that are triggered by reactive rules might be complex operations involving several (basic) actions that have to be performed concurrently or in a given order and under certain conditions. The possibility to define events and actions in a compositional way (meaning in terms of sub-events and sub-actions that have been already defined), would permit a simpler and more elegant

programming style by breaking complex definitions into simpler ones and by making it possible to use the definition of the same entity in different fragments of code.

An ECA language would also benefit from a declarative semantics that would valorize the simplicity of the basic concepts of the ECA paradigm. Moreover, an ECA language must in general be coupled with a knowledge base, which, in our opinion, should be richer than a simple set of facts, and allow for the specification of data and classical rules, i.e. rules that specify knowledge about the environment, besides the ECA rules that specify reactions to events. Together with the richer knowledge base, an ECA language should exhibit inference capabilities in order to extract knowledge from such data and rules.

Clearly ECA languages deal with systems that evolve. However, in existing ECA languages this evolution is mostly limited to the evolution of the (extensional) knowledge base. But in a truly evolving system, that is able to adapt to changes in the considered domain, there can be evolution of more than the extensional knowledge base: derivation rules of the knowledge base (intensional knowledge), as well as the reactive rules themselves may change over time. We believe another capability that should be considered is that of *evolving* in this broader sense. Here, by evolving capability we mean that a program should be able to automatically integrate external updates and to autonomously perform self updates. The language should allow to update both the knowledge (data and classical rules) and the behaviour (reactive rules) of the considered ECA program due to external and internal changes.

To the best of our knowledge, no existing ECA language provides all the above mentioned features, in particular, none provides the evolving capability (for a detailed discussion see section 6). The purpose of this paper is to define an ECA language based on logic programming that satisfies all these features. Logic programming (LP) is a flexible and widely studied paradigm for knowledge representation and reasoning based on rules. In the last years, in the area of LP, an amount of efforts has been developed to provide a meaning to updates of logic programs by other logic programs. The output of this research are frameworks that provide meaning to sequence of logic programs, also called Dynamic Logic Programs (DyLPs) [5, 7, 9, 14, 17, 24, 29, 34, 32], where the initial program is seen as the initial knowledge base and the subsequent programs are the various updates (see Section 2). DyLPs and the update languages defined on top of them [6, 18, 23, 8] conjugate a declarative semantics and reasoning capabilities with the possibility to specify (self) evolutions of the program. However, unlike ECA paradigms, these languages do not provide mechanisms for specifying the execution of external actions nor they provide mechanism for specifying complex events or actions.

To overcome the limitations of both ECA and LP update languages, we elaborate an ECA language defined starting from DyLPs called ERA (after Evolving Reactive Algebraic programs) incorporating complex events, external and complex actions with the inference and evolving capabilities of updates languages.

Besides *reactive rules* of the form $(1)$, ERA also allows LP rules supporting default negation [26] and the possibility to express negation in the head of rules [25]. Regarding reactive rules, $Event$ is a basic or complex event expressed by an algebra similar to the Snoop algebra [4]. Basic events are passed to a program as external inputs. These inputs are represented by sets of data and rules called *input programs*. The $Condition$ part in

ERA is a set of literals. Like events, actions can be basic or complex. Basic actions can be external (modify the environment) or internal (add or retract data, *rules*, and *reactive rules*). Complex actions are obtained by applying algebraic operators on basic actions that specifies whether two actions have to be executed concurrently or sequentially, or that, if some condition hold an action is executed, otherwise another action is executed instead. ERA also allows *inhibition rules* of the form:

$$\textbf{When } B \textbf{ Do } not\ Action \tag{2}$$

that intuitively mean: when $C$ is satisfied, do not execute $Action$. Inhibition rules are mainly used to update the behaviour of reactive rules. If the inhibition rules above is asserted (either by an external or a self update) all the reactive rules with $Action$ in the head are updated with the extra condition that $C$ must *not* be satisfied in order to execute $Action$.

A semantics for ERA is defined by means of an *inference system* (that specifies what conclusions are derived by a program) and of an *operational semantics* (that specifies the effects of actions). The former is derived from the refined semantics for DyLPs [5]. The latter is defined by a transition system inspired by existing work on process algebras. [27, 22, 30].

The rest of the paper is structured as follows: In section 2 we briefly introduce the syntax and semantics of DyLPs, and establish general notation. We start section 3 by informally illustrating an example of an ECA system, and then we present the syntax of ERA programs formally elaborating (a part of) the example. Section 4 is dedicated to the definition of the semantics of ERA. Section 5 completes the formal elaboration of the example presented in section 3. In section 6 we discuss related works, confront them with ERA and, finally, we draw conclusions and sketch future work.

## 2 Background and Notation

In what follows, we use the standard LP notation and, for the knowledge base, *generalized logic programs* (GLP) [25]. Arguments of predicates (here also called atoms) are enclosed within parenthesis and separated by commas. Names of arguments with capitalized initials stand for variables, names with uncapitalized initials stand for constants.

A GLP over an alphabet (a set of propositional atoms) $\mathcal{L}$ is a set of rules of the form $L \leftarrow B$, where $L$ (called the head of the rule) is a literal over $\mathcal{L}$, and $B$ (called the body of the rule) is a set of literals over $\mathcal{L}$. As usual, a literal over $\mathcal{L}$ is either an atom $A$ of $\mathcal{L}$ or the negation of an atom $not\ A$. In the sequel we also use the symbol $not$ to denote complementary default literals, i.e. if $L = not\ A$, by $not\ L$ we denote the atom $A$.

A (two-valued) *interpretation* $I$ over $\mathcal{L}$ is any set of literals in $\mathcal{L}$ such that, for each atom $A$, either $A \in I$ or $not\ A \in I$. A set of literals $S$ is true in an interpretation $I$ (or that $I$ satisfies $S$) iff $S \subseteq I$. In this paper we will use programs containing variables. As usual in these cases a program with variables stands for the propositional program obtained as the set of all possible ground instantiations of the rules. Two rules $\tau$ and $\eta$ are *conflicting* (denoted by $\tau \bowtie \eta$) iff the head of $\tau$ is the atom $A$ and the head of $\eta$ is $not\ A$, or viceversa.

A Dynamic Logic Program $\mathcal{P}$ over an alphabet $\mathcal{L}$ is a sequence $P_1, \ldots, P_m$ where the $P_i$s are GLPs defined over $\mathcal{L}$. Given a DyLP $P_1 \ldots P_n$ and a set of rules $R$ we denote by $\mathcal{P} \setminus R$ the sequence $P_1 \setminus R, \ldots, P_n \setminus R$ where $P_i \setminus R$ is the program obtained by removing all the rules in $R$ from $P_i$. The *refined stable model semantics* of a DyLP, defined in [5], assigns to each sequence $\mathcal{P}$ a set of refined models (that is proven there to coincide with the set of stable models when the sequence is formed by a single normal [21] or generalized program [25]). The rationale for the definition of a refined model $M$ of a DyLP is made according with the *causal rejection principle* [17, 23]: If the body of a rule in a given update is true in $M$, then that rule rejects all rules in previous updates that are conflicting with it. Such rejected rules are ignored in the computation of the stable model. In the refined semantics for DyLPs a rule may also reject conflicting rules that belong to the same update. Formally the set of rejected rules of a DyLP $\mathcal{P}$ given an interpretation $M$ is:

$$Rej^S(\mathcal{P}, M) = \{\tau \in P_i : \exists\, \eta \in P_j\ i \leq j,\ \tau \bowtie \eta\ \wedge\ B(\eta) \subseteq M\}$$

An atom $A$ is false by default if there is no rule, in none of the programs in the DyLP, with head $A$ and a true body in the interpretation $M$. Formally:

$$Default(\mathcal{P}, M) = \{not\ A\ :\ \nexists\ A \leftarrow B \in \bigcup P_i\ \ \wedge B \subseteq M\}$$

If $\mathcal{P}$ is clear from the context, we omit it as first argument of the above functions.

**Definition 1.** *Let $\mathcal{P}$ be a DyLP over the alphabet $\mathcal{L}$ and $M$ an interpretation. $M$ is a refined stable model of $\mathcal{P}$ iff*

$$M = least\left(\left(\bigcup P_i \setminus Rej^S(M)\right) \cup Default(M)\right)$$

*where $least(P)$ denotes the least Herbrand model of the definite program [26] obtained by considering each negative literal $not\ A$ in $P$ as a new atom.*

In the following, a conclusion over an alphabet $\mathcal{L}$ is any set of literals over $\mathcal{L}$. An inference relation $\vdash$ is a relation between a DyLP and a conclusion. Given a DyLP $\mathcal{P}$ with a unique refined model $M$ and a conclusion $B$, it is natural to define an inference relation $\vdash$ as follows: $P_S \vdash B \ \Leftrightarrow B \subseteq M$ ($B$ is derived iff $B$ is a subset of the unique refined model). However, in the general case of programs with several refined models, there could be several reasonable ways to define such a relation. A possible choice is to derive a conclusion $B$ iff $B$ is a subset of the intersection of all the refined models of the considered program ie, $P_S \vdash B \ \Leftrightarrow\ B \subseteq M\ \forall\ M \in \mathcal{M}(\mathcal{P})$ where $\mathcal{M}(\mathcal{P})$ is the set of all refined models of $\mathcal{P}$. This choice is called *cautious reasoning*. Another possibility is to select one model $M$ (by a selecting function $Se$) and to derive all the conclusions that are subsets of that model ie, $\mathcal{P} \vdash B \ \Leftrightarrow\ B \subseteq Se(\mathcal{M}(\mathcal{P}))$. This choice is called *brave reasoning*. In the following, in the context of DyLPs, whenever an inference relation $\vdash$ is mentioned , we assume that $\vdash$ is one of the relations defined above.

Let $E_S$ be a sequence of programs (ie, a DyLP) and $E_i$ a GLP, by $E_i.E_S$ we denote the sequence with head $E_i$ and tail $E_S$. If $E_S$ has length $n$, by $E_S..E_{n+1}$ we denote the sequence whose first $n^{th}$ elements are those of $E_S$ and whose $(n+1)^{th}$ element

is $E_{n+1}$. For simplicity, we use the notation $E_i.E_{i+1}.E_S$ and $E_S..E_i..E_{i+1}$ in place of $E_i.(E_{i+1}.E_S)$ and $(E_S..E_i)..E_{i+1}$ whenever this creates no confusion. Symbol $null$ denotes the empty sequence. Let $E_S$ be a sequence of $n$ GLPs and $i \leq n$ a natural number, by $E_S^i$ we denote the sequence of the first $i^{th}$ elements of $E_S$. Let $\mathcal{P} = \mathcal{P}'..P_i$ be a DyLP and $E_i$ a GLP, by $\mathcal{P} \uplus E_i$ we denote the DyLP $\mathcal{P}'..(P_i \cup E_i)$.

## 3 Syntax of ERA programs

Before the definition of the syntax of ERA programs, we present a motivating examples that, besides illustrating several features of the languages that are not present in other ECA languages (such as the ones pointed out in the introduction), helps on informally introducing the syntax.

*Example 1.* Consider an (ECA) system for managing several electronic devices of a building, particulary the phone lines and the fire security system. The system receives inputs such as signals of sensors and messages from employees and system administrators, and can activate devises like electric doors or fireplugs, redirect phone calls and send emails. For instance, sensors alert the system whenever an abnormal quantity of smoke is found. If a (basic) event ($alE(S)$) corresponding to a warning from a sensor $S$ occurs, the system opens all the fireplugs $Pl$ in the floor where $S$ is located. This behaviour is encoded by a reactive rule reacting to a basic event with a basic action $openA(Pl)$. The situation is different when the signals are given by several sensors. If two signals from sensors located in different rooms occur without a $stop\_alertE$ event occurring in the meanwhile, the system starts a more sophisticated action $fire\_alarmA$ that applies a security protocol. All the doors are unlocked (by the action $opendoorsA$) to allow people to leave the building. At the same time, the system sends a registered phone call to a firemen station (by the action $firecallA$). Then the system cuts the electricity in the building (the fireplugs work with an independent electric system). To open the doors and call the firemen station are actions that can be executed simultaneously, but the last one (cutting the electricity) has to be executed after the electric doors have been opened (otherwise they would remain closed). To implement such a behaviour we clearly need an ECA language capable to define complex events ($alert2E$) and actions $fire\_alarmA$ starting from basic ones.

Knowledge representation and reasoning also plays an important role. Suppose, for instance, the system receives an event notifying that a meeting of a big working group has been organized. The system must advise by email all the employee in the working group. The big working group is formed by subgroups possibly themselves divided into other subgroups. The system has stored data on which subgroup an employee belongs to; it must be able to represent that, if an employee belongs to a subgroup, he also belongs to its supergroups and so to infer which are the ones in the big working group.

Finally, let us consider evolution. After some false alarms, the administrators decide to update the behaviour of the system. From then onwards, when a sensor rises an alarm, only the fireplugs in the room where the signal is located will be open. Moreover, an employee can communicate to the system to start redirecting phone calls to him and to stop redirection by turning back to the previous behaviour (whatever it was). Such changes could be done by handily modifying the system. ERA, as we shall see,

offers the possibility to update reactive rules instead of rewriting. This second approach could be very useful in large systems possibly developed and modified by several programmers and administrators. In particular, it could be very useful when updates are performed by users that are not aware of the existing rules governing the system (see example in section 5).

Expressions in ERA are formed by atoms with a LP-like syntax (cf. section 2 for details). In the sequel, we use names of atoms ending by $E$ to represent events, and ending by $A$ to represent actions. E.g. atom $openA(D)$ is the action of opening a device $D$, $openE(D)$ is the basic event occurring whenever $D$ is opened, while $open(D)$ is the internal representation of the fact that $D$ is open.

Expressions in an ERA program are divided in *rules* (themselves divided into *active, inference and inhibition rules*,) and *definitions* (themselves divided into *event* and *action definitions*). We already seen in section 1 the form of reactive rules. Atoms in the condition part of a reactive rule are separated by commas. When the $Condition$ part of a reactive rule is the empty set of literals, to simplify notation we omit the operator **if**. For instance, the reactive rule

$$\textbf{On } alE(S) \ \textbf{If } flr(S, Fl), firepl(Pl), flr(Pl, Fl) \ \textbf{Do } openA(Pl). \qquad (3)$$

stands for "on (event) alarm $alE(S)$, if sensor $S$ and fireplug $Pl$ are at the same floor $Fl$, then open $Pl$".

To allow for representing and reasoning about knowledge, ERA uses *inference rules* with a LP form. For instance, the rule:

$$ingroup(Emp, G) \leftarrow ingroup(Emp, S), sub(S, G).$$

specifies that an employee $Emp$ belongs to the working group $G$ if he belongs to $S$ which is a subgroup of $G$ and that every working group is a subgroup of itself. Together with facts of the form $subgroup(s, g)$ and $ingroup(emp, g)$, these rules allow to infer which employees belong to a given working group[1].

The reactive rule $(3)$ executes an external action. While external actions are related to the specific application of the language, internal actions always have one of the following forms: $rise(e), assert(\tau), retract(\tau), define(d)$. Action $rise(e_b)$, where $e_b$ is a basic event, means "$e_b$ occurs in the next input programs", while the remaining internal actions actually modify the program. Actions $assert(\tau)$ and $retract(\tau)$, where $\tau$ is (any kind of) rule, mean, respectively, "update the current program with $\tau$" and "delete $\tau$ from the current program" (see below for the discussion of action $define(d)$). For instance, the following reactive rules update the program each time a device $D$ is opened or closed.

$$\textbf{On } openE(D) \ \textbf{Do } assert(open(D)). \ \textbf{On } closeE(D) \ \textbf{Do } assert(not\ open(D)).$$

Sometimes we need to combine basic events to obtain complex ones. This is done by an event algebra whose operators are: $\triangle \mid \ \bigtriangledown \ \mid A \mid not$ . Given the events $e_1, e_2, e_3$,

[1] The rules above uses recursion, on the predicate $ingroup/2$, a feature that is beyond the capabilities of many ECA commercial system, like e.g. SQL-triggers [33].

event $e_1 \triangle e_2$ occurs at instant $i$ iff $e_1$ *and* $e_2$ occur at instant $i$; event $e_1 \bigtriangledown e_2$ occurs at instant $i$ iff $e_1$ *or* $e_2$ occur at instant $i$. Event $A(e_1, e_2, e_3)$ occurs at instant $i$ iff event $e_1$ occurred at some previous instant $m$, $e_2$ did not occurred at $m$ nor at any instant between $m$ and $i$ and $e_3$ occurs for the first time since $m$ at instant $i$. Given an event $e$, event $not\ e$ occurs at instant $i$ iff $e$ *does not* occur at instant $i$. It is also possible to define a new event $e_{def}$ by associating an event $e$ to the atom $e_{def}$. This is done by expressions called *event definitions* of the form $e_{def}$ **is** $e$ where $e$ is obtained by the event algebra above.

For instance, let $falsE$ be the basic event that *never* occurs. Event $e_1; e_2$ means "$e_1$ occurred in the past and now $e_2$ occurs" and its definition is:
$e_1; e_2$ **is** $A(e_1, e_2, falsE)$. It is also possible to use defined events in the definition of other events. For instance, event $e_1 + e_2$ means " $e_1$ and $e_2$ occurred concurrently or in any order" and its definition is: $e_1 + e_2$ **is** $(e_1 \triangle e_2) \bigtriangledown (e_1; e_2) \bigtriangledown (e_2; e_1)$. Still referring to example 1,event $alert2E(S_1, S_2)$ occurs when two signals $alE(S_1), alE(S_2)$ occur simultaneously or in different instants without event $stop\_alertE$ occurring in the meanwhile. Formally:

$$alert2E(S_1, S_2) \textbf{ is } A(alE(S_1), alE(S_2), stop\_alertE) \bigtriangledown (alE(S_1) \triangle alE(S_2))$$

As for events, often we need to combine basic actions to obtain complex ones. In ERA this is done by an *action algebra* whose operators are: $\triangleright | \ \| \ | \ IF$. Given the actions $a_1, a_2$ and a literal $C$, to execute $a_1 \triangleright a_2$ means to *sequentially* execute first $a_1$, then $a_2$; to execute $a_1 \| a_2$ means to *concurrently* execute $a_1$ and $a_2$. To execute $IF(C, a_1, a_2)$ means that " *if* $C$ is derived, *then* execute $a_1$ *else* execute $a_2$. As for events, it is possible to define new actions. This is done by expressions called *action definitions* of the form $a_{def}$ **is** $a$ where $a$ is obtained by the action algebra above and $a_{def}$ the name of the new actions.

For instance, complex action $fire\_alarmA$ is obtained by first executing concurrently the actions $opendoorsA$ (open all the electric doors in the building) and $firecallA$ (call the firemen station) and after that to execute action $electricityA(off)$ (turn the electricity off) and it is defined as follows:

$$fire\_alarmA \textbf{ is } (opendoorsA \| firecallA) \triangleright electricityX(off)$$

The basic action $define(d)$ introduced above, where $d$ is an event (resp. action) definition $e_{def}$ **is** $e$ (resp. $a_{def}$ **is** $a$) is used to introduce new definitions or to replace the old definitions for $e_{def}$ (resp. $a_{def}$ ) with the new one.

As anticipated in the introduction, *inhibition rules* are rules of the form $(2)$ where $B$ is a set of literals (normal literals or events) that are used to inhibit previous reactive rules. For instance, the inhibition rule

$$\textbf{When } alE(S), room(S, R), not\ room(Pl, R) \textbf{ Do } not\ openA(Pl).$$

updates rule $(3)$. When this rule is asserted, whenever $alE(S)$ occurs, any fire plug $Pl$ which is not in the room $R$ where the sensor $S$ is located is not opened, even if $Pl$ is on the same floor of $S$. A program developed in ERA initially consists of the *ERA program* $P_1$, i.e. an initial set of rules and definitions. Program $P_1$ is then updated by other ERA

programs $P_2, \ldots, Pn$ consisting of facts and rules asserted and new definitions. Such a sequences, called *ERA dynamic programs* determine, at each instant, the behaviour of the system. For this reason the semantics of ERA is given in section $4$ wrt ERA dynamic programs. Formally, the complete syntax of ERA is:

**Definition 2.** *Let $\mathcal{L}$, $\mathcal{E}_B$, $\mathcal{E}_{def}$, $\mathcal{A}_X$ and $\mathcal{A}_{def}$ be sets of atoms called,* condition alphabet *and set of, respectively,* basic events, event names, external actions *and* action names *and let $L$, $e_b$, $e_{def}$, $a_x$ and $a_{def}$ be generic elements of, respectively, $\mathcal{L}$, $\mathcal{E}_B$, $\mathcal{E}_{def}$, $\mathcal{A}_X$ and $\mathcal{A}_{def}$. The set of* positive events *$\mathcal{E}$ over $\mathcal{E}_B$, and $\mathcal{E}_{def}$ is the set of atoms $e_p$ of the form:*

$$e_p ::= e_b \mid e_1 \triangle e_2 \mid e_1 \bigtriangledown e_2 \mid A(e_1, e_2, e_3) \mid e_{def}.$$

*where $e_1, e_2, e_3$ are generic elements of $\mathcal{E}$. An* event over *$\mathcal{E}$ is any literal over $\mathcal{E}$. A* negative event *over $\mathcal{E}$ is any literal of the form $not\ e_p$.*

*A* basic action *$a_b$ over $\mathcal{E}, \mathcal{L}, A_X, \mathcal{A}_{def}$ is any atom of the form:*

$$a_b ::= a_x \mid rise(e_b) \mid assert(\tau) \mid retract(\tau) \mid define(d).$$

*where $\tau$ (resp. $d$) is any ERA rule (resp. definition) over $\mathcal{L}^{ERA}$.*

*The* set of actions *$\mathcal{A}$ over $\mathcal{E}, C, A_X, \mathcal{A}_{def}$ is the set of atoms $a$ of the form:*

$$a ::= a_b \mid a_1 \triangleright a_2 \mid a_1 \| a_2 \mid IF(C, a_1, a_2) \mid a_{def}.$$

*where $a_1$ and $a_2$ are arbitrary elements of $\mathcal{A}$ and $C$ is any literal over $\mathcal{E} \cup \mathcal{L}$.*

*The* ERA alphabet *(or simply the alphabet) $\mathcal{L}^{ERA}$ over $\mathcal{L}$, $\mathcal{E}_B$, $\mathcal{E}_{def}$, $\mathcal{A}_X$ and $\mathcal{A}_{def}$ is the triple $\mathcal{E}, \mathcal{L}, \mathcal{A}$. In the following, let $e$ and $a$ be arbitrary elements of, respectively, $\mathcal{E}$ and $\mathcal{A}$, $B$ any set of literals over $\mathcal{E} \cup \mathcal{L}$ and $Condition$ any set of literals over $\mathcal{L}$.An* ERA expression *is either an ERA definition or an ERA rule. An* ERA definition *is either an event definition or and action definition.An* event definition *over $\mathcal{L}^{ERA}$ is any expression of the form $e_{def}$ **is** $e$.An* action definition *over $\mathcal{L}^{ERA}$ is any expression of the form $a_{def}$ **is** $a$.An* ERA rule *is either an inference, active or inhibition rule over $\mathcal{L}^{ERA}$. An* inference rule *over $\mathcal{L}^{ERA}$ is any rule of the form $L \leftarrow B$.A* reactive rule *over $\mathcal{L}^{ERA}$ is any rule of the form **On** $e$ **If** $Condition$ **Do** $a$.An* inhibition rule *over $\mathcal{L}^{ERA}$ is any rule of the form **When** $B$ **Do** $not\ a$.An* ERA program *over $\mathcal{L}^{ERA}$ is any set of ERA rules and definitions over $\mathcal{L}^{ERA}$. An* ERA dynamic program *is any sequence of ERA programs.*

## 4 Semantics of ERA

Having defined a syntax for programming ECA systems in ERA (called, from now onwards, *ERA systems*) we now provide a semantics to such systems. An ERA system receives inputs in the form of *input programs*. Formally, an input program $E_i$, over an alphabet $\mathcal{L}^{ERA}$, is any set whose elements are either ERA expressions over $\mathcal{L}^{ERA}$ or facts of the form $e_b$ where $e_b$ is an element of $\mathcal{E}_B$ (i.e. a basic event). At any instant $i$, an ERA systems receives a, possibly empty, input program[2] $E_i$. The sequence of input programs $E_1, \ldots E_n$ denotes the sequence of input programs received at instants $1, \ldots, n$.

[2] ERA adopts a discrete concept of time, any input program is indexed by a natural number representing the instant at which the input program occurs.

A basic event $e_b$ *occurs* at instant $i$ iff the fact $e_b$ belongs to $E_i$. Since, by operator $A$, a complex event may be obtained combining basic events occurring at different instants, in order to detect the occurrence of such complex events in general it is necessary to store the sequence of all the received input programs. Formally, an *ERA system* $\mathcal{S}$ is a triple of the form $(\mathcal{P}, E_P, E_i.E_F)$ where $\mathcal{P}$ is an ERA dynamic program, $E_P$ is the sequence of all the previously received input programs and $E_i.E_F$ is the sequence of the current ($E_i$) and the future ($E_F$) input programs. As it will be clear from sections 4.1 and 4.2, the sequence $E_F$ does not influence the system at instant $i$ and hence no "look ahead" capability is required. However, since a system is capable (via action $rise$) of autonomously *rising* events in the future, future input programs are included in the system as "passive" elements that are modified as effects of actions (see rule (5)). Providing a semantics to ERA systems means to specify, at each instant, which conclusions are derived, which actions executed and what are the effects of those actions. Given a conclusion $B$, and an ERA system $\mathcal{S}$, notation $S \vdash_e B$ denotes that $S$ derives $B$ (or that $B$ is inferred by $S$). The definition of $\vdash_e$ is to be found in section 4.1.

At each instant, an ERA system $\mathcal{S}$ *concurrently* executes all the actions $a_k$ such that $\mathcal{S} \vdash_e a_k$. As a result of these actions an ERA system *transits* into another ERA system. While the execution of basic actions is "instantaneous", complex actions may involve the execution of several basic actions in a given order and hence require several transitions to be executed. For this reason, the effects of actions are defined by transitions of the form $< \mathcal{S}, A > \mapsto^G < \mathcal{S}', A' >$ where $\mathcal{S}, \mathcal{S}'$ are ERA systems, $A, A'$ are sets of actions and $G$ is a set of basic actions. The basic actions in $G$ are the first step of the execution of set of actions $A$, while the set of actions $A'$ represents the remaining steps to complete the execution of $A$. For this reason $A'$ is also called the *set of residual actions* of $A$. The transition relation $\mapsto$ is defined by a transition system in section 4.2. At each instant an ERA system receives an input program, derives a new set of actions $A_N$ and starts to execute these actions together with the residual actions not yet executed. As a result, the system evolves according to the transition relation $\rightarrow$. Formally:

$$\frac{A_N = \{a_k \in \mathcal{A} : \ \mathcal{S} \ \vdash_e a_k\} \ \wedge \ < \mathcal{S}, (A \cup A_N) > \mapsto^G < \mathcal{S}', A' >}{< \mathcal{S}, A > \rightarrow^G < \mathcal{S}', A' >} \tag{4}$$

### 4.1 Inferring conclusions

The inference mechanism of ERA is derived from the inference mechanism for DyLPs. In section 2, we provide two distinct ways (called resp. cautious and brave reasoning) to define an inference relation $\vdash$ between a DyLP and a conclusion on the base of the refined semantics. From inference relation $\vdash$, in the following we derive a relation $\vdash_e$ that infers conclusions from an ERA system.

Let $\mathcal{S} = (\mathcal{P}, E_P, E_i.E_F)$ be an ERA system over the alphabet $\mathcal{L}^{ERA} : \ (\mathcal{E}, \mathcal{L}, \mathcal{A})$, with $E_P = E_1, \ldots E_{i-1}$. For any $m < i$, let $\mathcal{S}^m$ be the ERA system $(\mathcal{P}, E^{m-1}, E^m.null)$. Sequence $E_F$ represents future input programs and is irrelevant for the purpose of inferring conclusions in the present, and sequence $E_P$ stores previous events, and is only used for detecting complex events. The relevant expressions are hence those in $\mathcal{P}$ and $E_i$. As a first step we reduce the expressions of these programs to LP rules. An event definition, associates an event $e$ to a new atom $e_{def}$. This is encoded by the rule

$e_{def} \leftarrow e$. Action definitions, instead, specify what are the effects of actions and hence are not relevant for inferring conclusions. Within ERA, actions are executed iff they are inferred as conclusions. Hence, reactive (resp. inhibition) rules are replaced by LP rules whose heads are actions (resp. negation of actions) and whose bodies are the events and conditions of the rules. Formally: let $\mathcal{P}^R$ and $E_i^R$ be the DyLP and GLP obtained by $\mathcal{P}$ and $E_i$ by deleting every action definition and by replacing:

| | | |
|---|---|---|
| every rule | **On** $e$ **If** $Condition$ **Do** $Action$. | with $Action \leftarrow Condition, e$. |
| every rule | **When** $B$ **Do** $not\ Action$ | with $not\ Action \leftarrow B$. |
| every definition | $e_{def}$ **is** $e$. | with $e_{def} \leftarrow e$. |

As the reader will notice, events are reduced to ordinary literals. Since events are meant to have special meanings, we encode these meanings by extra rules. Intuitively, operators $\triangle$ and $\triangledown$ stands for the logic operators $\wedge$ and $\vee$. This is encoded by the set of rules $ER(\mathcal{E})$ defined as follows:

$$ER(\mathcal{E}) : \triangle(e_1, e_2) \leftarrow e_1, e_2.\ \ \triangledown(e_1, e_2) \leftarrow e_1.\ \ \triangledown(e_1, e_2) \leftarrow e_2.\ \ \forall\, e_1, e_2, e_3\ \in \mathcal{E}$$

Event $A(e_1, e_2, e_3)$ occurs at instant $i$ iff $e_2$ occurs at instant $i$ and some conditions on the occurrence of $e_1, e_2$ and $e_3$ where satisfied in the previous instants. This is formally encoded by the set of rules $AR(\mathcal{S})$ defined as follows[3]:

$$AR(\mathcal{S}) = \left\{ \begin{array}{ll} \forall\, e_1, e_2, e_3\ \in \mathcal{E} & A(e_1, e_2, e_3) \leftarrow e_2 : \exists\, m < i\ t.c. \\ \mathcal{S}^m\ \vdash_e e_1 \wedge \mathcal{S}^m\ \nvdash_e e_3 \wedge\ \forall j\ \ t.c.\ m < j < i : & \mathcal{S}^j\ \ \nvdash_e e_2 \wedge \mathcal{S}^j\ \nvdash_e e_3 \end{array} \right\}$$

The sets of rules $E_i^R$, $ER(\mathcal{E})$ and $AR(\mathcal{S})$ are added to $\mathcal{P}^R$ and conclusions are derived by the inference relation $\vdash$ applied on the obtained DyLP[4]. Formally:

**Definition 3.** *Let $\vdash$ be an inference relation defined as in section 2, $\mathcal{S}$, $\mathcal{P}^R$, $E_i^R$, $ER(\mathcal{E})$, $AR(\mathcal{S})$ be as above and $K$ be any conclusion over $\mathcal{E} \cup \mathcal{L} \cup \mathcal{A}$. Then:*

$$(\mathcal{P}, E_P, E_i.E_F)\ \vdash_e K \quad \Leftrightarrow \quad \mathcal{P}^R \uplus (E_i^R \cup ER(\mathcal{E}) \cup D(\mathcal{P}) \cup AR(\mathcal{S})) \vdash K$$

Note that we do not specify rules for operator $not$ . These rules are not needed since event (literal) $not\ e_p$ is inferred by default negation whenever there is no prove for $e_p$. In section 3 we provided the intuitive meanings of the various operators. The following theorem formalizes these intuitive meanings.

**Theorem 1.** *Let $\mathcal{S}$ be as above, $e_b$, a basic event, $e_p$ a positive event, $e_{def}$ an event name and $e_1, e_2, e_3$ three events, the following double implications hold:*

$$\begin{array}{lll} \mathcal{S}\ \vdash_e e_1 \triangle e_2 & \Leftrightarrow \mathcal{S}\ \vdash_e e_1\ \wedge\ \mathcal{S}\ \vdash_e e_2. & \mathcal{S}\ \vdash_e e_b\ \ \Leftrightarrow\ \ e_b \in E_i \\ \mathcal{S}\ \vdash_e e_1 \triangledown e_2 & \Leftrightarrow \mathcal{S}\ \vdash_e e_1\ \vee\ \mathcal{S}\ \vdash_e e_2. & \mathcal{S}\ \vdash_e not\ e_p \Leftrightarrow\ \mathcal{S}\ \nvdash_e e_p. \\ \mathcal{S}\ \vdash_e A(e_1, e_2, e_3) & \Leftrightarrow\ \exists\, m < i\ t.c.\ \mathcal{S}^m\ \vdash_e e_1\ \wedge \mathcal{S}^m\ \nvdash_e e_3\ \wedge\ \forall j\ t.c. & \\ & m < j < i :\ \mathcal{S}^j\ \ \nvdash_e e_2 \quad \wedge \mathcal{S}^j\ \nvdash_e e_3\ \wedge\ \mathcal{S}\ \vdash_e e_2. & \\ \mathcal{S}\ \vdash_e e_{def} & \Leftrightarrow \mathcal{S}\ \vdash_e e\ \wedge\ e_{def}\ \textbf{\textit{is}}\ e\ \in \mathcal{P} & \end{array}$$

[3] The definition of $AR(\mathcal{S})$ involves relation $\vdash_e$ which is defined in terms of $AR(\mathcal{S})$ itself . This mutual recursion is well-defined since, at each recursion, $AR(\mathcal{S})$ and $\vdash_e$ are applied on previous instants until eventually reaching the initial instant (ie, the basic step of recursion)

[4] The program transformation above is functional for defining a declarative semantics for ERA, rather then providing an efficient tool for an implementation. Here specific algorithms for event-detection clearly seems to provide a more efficient alternative.

### 4.2 Execution of actions

We are left with the goal of defining what are the effects of actions. This is accomplished by providing a transition system for the relation $\mapsto$ that completes, together with transition $(4)$ and the definition of $\vdash_e$, the semantics of ERA. As mentioned above, these transitions have the form: $< \mathcal{S}, A > \mapsto^G < \mathcal{S}', A' >$.

The effects of basic actions on the current ERA program are defined by the *updating function* $up/2$. Let $\mathcal{P}$ be and ERA dynamic program $A$ a set of, either internal or external, basic actions. The output of function $up/2$ is the updated program $up(P, A)$ obtained in the following way: First delete from $\mathcal{P}$ all the rules retracted according to $A$, and all the (event or action) definitions $d_{def}$ **is** $d_{old}$ such that action $define(d_{def}$ **is** $d_{new})$ belongs to $A$; then update the obtained ERA dynamic program with the program consisting of all the rules asserted according to $A$ and all the new definitions in $A$. Formally:

$$\begin{aligned} DR(A) &= \{d : define(d) \in A\} \ \cup \ \{\tau : \ assert(\tau) \in A\} \cup D(A) \\ R(\mathcal{P}, A) &= \{\tau : \ retract(\tau) \in A\} \cup \ \{d_{def} \textbf{ is } d_{old} \ \in \mathcal{P} : \ d_{def} \textbf{ is } d_{new} \in D(A)\} \\ up(\mathcal{P}, A) &= (\ \mathcal{P} \setminus R(\mathcal{P}, A)\ ) .. DR(A) \end{aligned}$$

Let $e_b$ be any basic event and $a_i$ an external action or an internal action of one of the following forms: $assert(\tau), retract(\tau), define(d)$. On the basis of function $up/2$ above, we define the effects of (internal and external) basic actions. At each transition, the current input program $E_i$ is evaluated and stored in the sequence of past events and the subsequent input program in the sequence $E_F$ becomes the current input program (see 1st and 3rd rules below). The only exception involves action $rise(e_b)$ that adds $e_b$ in the subsequent input program $E_{i+1}$. When a set of actions $A$ is completely executed its set of residual actions is $\emptyset$. Basic actions (unlike complex ones) are completely executed in one step, hence they have no residual actions. Formally:

$$\begin{aligned} < (\mathcal{P}, E_P, E_i.E_F), \emptyset > \ &\mapsto^{\emptyset} \ < \mathcal{P}, E_P..E_i, E_F, \emptyset > \\ < (\mathcal{P}, E_i.E_{i+1}.E_S), \{rise(e_b)\} > \ &\mapsto^{\emptyset} \ < (\mathcal{P}, E_P..E_i, (E_{i+1} \cup \{e_b\}).E_F), \emptyset > \\ < (\mathcal{P}, E_P, E_i.E_F), \{a_i\} > &\mapsto^{\{a_i\}} < (up(\mathcal{P}, \{a_i\}), E_P..E_i, E_F), \emptyset > \end{aligned} \quad (5)$$

Note that, although external actions do not affect the ERA system, as they do not affect the result of $up/2$, they are nevertheless *observable*, since they are registered in the set of performed actions (cf. 3rd rule above). Unlike basic actions, generally the execution of a complex action involves several transitions. Action $a_1 \triangleright a_2$, where $a_1, a_2$ are arbitrary actions, consists into first executing all basic actions for $a_1$, until the set residual actions is $\emptyset$, then to execute all the basic actions for $a_2$. We use the notation $A_1 \triangleright a_2$, where $A_1$ is a set of actions, to denote that action $a_2$ is executed after all the actions in the set $A_1$ have no residual actions. Action $a_1 \| a_2$, instead, consists into *concurrently* executing all the basic actions forming both actions, until there are no more of residual actions to execute. Similarly, the execution of a set of actions $A$ consists in the concurrent execution of all its actions $a_k$ until the set of residual actions is empty. Semantically, the concurrent execution of basic actions is defined by function $up/2$ which is defined over an ERA dynamic program and an *set* of basic actions. The execution of $IF(C, a_1, a_2)$ is equal to the execution of $a_1$ if the system infers $C$, otherwise it is equal

to the execution of $a_2$. Given an ERA system $\mathcal{S} = (\mathcal{P}, E_P, E_i.E_F)$ with $\mathcal{P} : P_1 \dots P_n$, let $D(\mathcal{S})$ the set of all the action definitions $d$ such that, for some $j$, $d \in P_j$ or $d \in E_i$. The execution of action $a_{def}$, where $a_{def}$ is defined by one ore more action definitions, corresponds to the concurrent executions of all the actions $a_k$s such that $a_{def}$ **is** $a_k$ belongs to $D(\mathcal{S})$. Formally:

$$\frac{< \mathcal{S}, \{a_1, a_2\} > \mapsto^{G} < \mathcal{S}', A' >}{< \mathcal{S}, \{a_1 \| a_2\} > \mapsto^{G} < \mathcal{S}', A' >} \quad \frac{< \mathcal{S}, \{a_1\} > \mapsto^{G_1} < \mathcal{S}', A'_1 >}{< \mathcal{S}, \{a_1 \triangleright a_2\} > \mapsto^{G_1} < \mathcal{S}', \{A'_1 \triangleright a_2\} >}$$

$$\frac{< \mathcal{S}, A_1 > \mapsto^{G_1} < \mathcal{S}', A'_1 >}{< \mathcal{S}, \{A_1 \triangleright a_2\} > \mapsto^{G_1} < \mathcal{S}', \{A'_1 \triangleright a_2\} >} \quad \frac{< \mathcal{S}, \{a_2\} > \mapsto^{G_2} < \mathcal{S}'', A''_2 >\}}{< \mathcal{S}, \{\emptyset \triangleright a_2\} > \mapsto^{G_2} < \mathcal{S}'', A''_2 >}$$

$$\frac{\mathcal{S} \vdash_e C \wedge < \mathcal{S}, \{a_1\} > \mapsto^{G_1} < \mathcal{S}', A' >}{< \mathcal{S}, \{IF(C, a_1, a_2)\} > \mapsto^{G_1} < \mathcal{S}', A'_1 >} \quad \frac{\mathcal{S} \nvdash_e C \wedge < \mathcal{S}, \{a_2\} > \mapsto^{G_2} < \mathcal{S}'', A''_2 >}{< \mathcal{S}, \{IF(C, a_1, a_2)\} > \mapsto^{G_2} < \mathcal{S}'', A''_2 >}$$

$$\frac{A = \{a_1, \dots, a_n\} \; < (\mathcal{P}, E_P, E_i.E_{i+1}.E_F), \{a_k\} > \mapsto^{G_k} < (\mathcal{P}^k, E_P..E_i, E^k_{i+1}.E_F), A'_k\} >}{< (\mathcal{P}, E_P, E_i.E_{i+1}.E_F), A > \mapsto^{\bigcup G_k} < (up(\mathcal{P}, \bigcup G_k), E_P..E_i, \bigcup E^k_{i+1}.E_F), \bigcup A'_k >}$$

$$\frac{A = \{a_k : \; a_{def} \textbf{ is } a_k. \in D(\mathcal{S})\} \wedge \; < \mathcal{S}, A > \mapsto^{G} < \mathcal{S}', A' >}{< \mathcal{S}, \{a_{def}\} > \mapsto^{G} < \mathcal{S}', A' >}$$

## 5 Example of Usage: A monitoring system

We now present the ERA program $P_1$ (partially) formalizing example 1.

| | | |
|---|---|---|
| $(a)$ | **On** $alE(S)$ | **If** $flr(S, Fl), firepl(Pl), flr(Pl, Fl)$ **Do** $openA(Pl)$. |
| $(b)$ | **On** $openE(D)$ | **Do** $assert(open(D))$. |
| $(c)$ | **On** $closeE(D)$ | **Do** $assert(not\ open(D))$. |
| $(d)$ | $alert2E(S_1, S_2)$ | **is** $A(alE(S_1), alE(S_2), stop\_alertE) \bigtriangledown (alE(S_1) \bigtriangleup alE(S_2))$. |
| $(e)$ | $fire\_alarmA$ | **is** $(opendoorsA \| firecallA) \triangleright electricityA(off)$. |
| $(f)$ | **On** $alert2E(S_1, S_2)$ | **If** $not\ same\_room(S_1, S_2)$ **Do** $fire\_alarmA$. |
| $(g)$ | $same\_room(S_1, S_2)$ | $\leftarrow room(S_1, R_1), room(S_2, R_1)$. |
| $(j)$ | **On** $seminarE(S)$ | **If** $compose(S, M), sendto(S, E@)$ **Do** $sendA(M, E@)$. |
| $(k)$ | $sendto(S, E@)$ | $\leftarrow group(S, G), ingroup(Emp, G), mail(Emp, E@)$. |
| $(h)$ | $ingroup(Emp, G)$ | $\leftarrow ingroup(Emp, S), sub(S, G)$. |

We already discussed expressions $(a - e)$ in section 3. Reactive rule $(f)$ triggers action $fire\_alarmA$ when event $alert2E(S_1, S_2)$ occurs (ie, two sensors rise an alarm) under the condition that the senors are located in different rooms (this condition is inferred by rule $g$). When event $seminarE(S)$ occurs (a seminar $S$ is announced) reactive rule $(j)$ triggers action $sendA(M, E@)$ ie, a message $M$ is sent to the email address $E@$ of any employee $Emp$ in the working group $G$ to which seminar $S$ is dedicated (rules $(k-h)$).

An external update is done by an input program containing internal actions ($assert$, $retract$ and $define$) as facts. For instance, the behaviour of $P_1$ is updated by the following input program:

$$E_i : \quad assert(R_1). \quad assert(R_2). \quad assert(R_3). \quad define(d_1). \quad define(d_2).$$

where $R_1 - R_3$ and $d_1 - d_2$ are the following expressions.

$R_1$ : **When** $alE(S), room(S,R), not\ room(Pl,R)$ **Do** $not\ openA(Pl)$.
$R_2$ : **On** $redirectE(Emp,Num)$ **Do** $redirectA(Emp,Num)$.
$R_3$ : **On** $stop\_redirectE(Emp,Num)$ **Do** $stop\_redirectA(Emp)$.
$d_1 : redirectA(Emp,Num)$ **is** $assert(\tau_1) \triangleright assert(\tau_2)$.
$d_2 : stop\_redirectA(Emp,Num)$ **is** $retract(\tau_1) \| retract(\tau_2)$.

and $\tau_1$ and $\tau_2$ are the following rules:

$\tau_1$ : **When** $phonE(Call), dest(Call,Emp)$ **Do** $not\ forwX(Call,N)$.
$\tau_2$ : **On** $p\ honE(Call)$ **If** $dest(Call,Emp)$ **Do** $forwX(Call,Num)$.

We already discussed the effect of inhibition rule $R_1$. Reactive rules $R_2 - R_3$ encode the new behaviour of the system when an employee $Emp$ asks the system to start (resp. to stop) redirecting to the phone number $Num$ any phone call $Call$ to him. This is achieved by sequentially asserting (resp. retracting) rules $\tau_1, \tau_2$. The former is an inhibition rule that inhibits any previous rule reacting to a phone call for $Emp$ (ie, to the occurrence of event $phonE(Call)$) by forwarding the call to a number $N$. The latter is a reactive rule forwarding the call to number $Num$. Note that the two rules have to be asserted sequentially in order to prevent mutual conflicts. To revert to the previous behaviour it is sufficient to retract the two rules as done by action $stop\_redirectA$.

## 6 Conclusions and related work

We identified a set of desirable features for an ECA language, namely: a declarative semantics, the capability to express complex events and actions in a compositional way, and that of receiving external updates, and perform self updates to data, inference rules and reactive rules. For this purpose we defined the logic programming ECA language ERA, based on the concept of DyLPs and provided it with a declarative semantics based on the refined semantics of DyLPs (for inferring conclusions) and a transition system (for the execution of actions).

There exists several alternative proposals of ECA formalisms. However, most of these approaches are mainly procedural like, for instance, AMIT [31], RuleCore [1] and JEDI [19] or at least not fully declarative [33]. A declarative situation calculus-like characterizations of active database systems is given in [10], although the subject of complex actions is not treated there. Examples of Semantic Web-oriented ECA languages are XChange [13] and Active XML [3]. XChange has an LP-like semantics, allows to define reactive rules with complex actions and (unlike Active XML) complex events. However, neither of the two supports a construct similar to action definitions for defining actions. This possibility is allowed by the Agent-Oriented Language DALI [16]. On the other hand, DALI does not support complex events. The ideas and methodology for defining complex actions are inspired to works on process algebras like CCS [27], CSP [22] and TCC [30]. Rather then proposing high level ECA languages, these works design abstract models for defining programming languages for parallel execution of processes. Other related frameworks are Dynamic Prolog [12] and Transaction

Logic Programming (TLP) [11]. These works focus on the problem of updating a deductive database by performing transactions. In particular, TLP shares with ERA the possibilities to specify complex (trans)actions in terms of other, simpler, (trans)actions. However, TLP (and Dynamic Prolog) does not support complex events, nor does it cope with the possibility to receive external inputs during the computation of complex actions. None of the languages cited so far shows updates capabilities analogous to the ones evidenced in ERA. A class of proposals close in spirit to ERA, are LP update languages like EPI [18], Evolp [6], LUPS [8] and Kabul [23]. All these languages show the possibility to update rules. Within EPI and LUPS, it is possible to update derivation rules but not reactive rules. Evolution of reactive rules is a feature of both Kabul and Evolp. However, nor Evolp, Kabul, nor any of the cited LP update languages supports external nor complex actions or complex events.

The language ERA still requires a relevant amount of research. Preliminary investigations evidenced interesting properties of the operators of the action algebra like associativity, commutativity etc. Moreover, the transition system for ERA defined in section 4 is deterministic. Beside the opportunities opened by action definitions, the problem arises of restricting the class of allowed definitions in order to guarantee that transition relation $(4)$ is always defined. Our idea is to apply a criterium of *left guardedness* to action definitions (see [27]). In this work we opted for an inference system based on the refined semantics for DyLPs. With limited efforts, it would be possible to define an inference system on the basis of another semantics for DyLPs such as [7, 9, 14, 17, 24, 29, 34, 32]. In particular, we intend to develop a version of ERA based on the well founded semantics of DyLPs [9]. Well founded semantics [20] is a polynomial approximation of the answer set semantics that could be particulary suitable for applications involving large databases requiring the capability to quickly process vast amount of information. Finally, implementations of the language and its possible extensions are in order.

## References


1. Rulecore. *http://www.rulecore.com*.
2. Amazon web site. http//:www.amazon.com/, 2001.
3. S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active xml: Peer-to-peer data and web services integration. *citeseer.ist.psu.edu/article/abiteboul02active.html*, 2002.
4. Raman Adaikkalavan and Sharma Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. In *ADBIS*, pages 190–204, 2003.
5. J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. The refined extension principle for semantics of dynamic logic programming. *Studia Logica*, 79(1), 2005.
6. J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *JELIA'02*, LNAI, 2002.
7. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1–3):43–70, 2000.
8. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS: A language for updating logic programs. *Artificial Intelligence*, 132(1 & 2), 2002.
9. F. Banti, J. J. Alferes, and A. Brogi. The well founded semantics for dynamic logic programs. In Christian Lemaître, editor, *Proceedings of the 9th Ibero-American Conference on Artificial Intelligence (IBERAMIA-9)*, LNAI, 2004.

10. Chitta Baral and Jorge Lobo. Formal characterization of active databases. In *Logic in Databases*, pages 175–195, 1996.
11. A. J. Bonner and M. Kifer. Transaction logic programming. In David S. Warren, editor, *ICLP-93*, pages 257–279, Budapest, Hungary, 1993. The MIT Press.
12. Anthony J. Bonner. A logical semantics for hypothetical rulebases with deletion. *Journal of Logic Programming*, 32(2), 1997.
13. François Bry, Paula-Lavinia Patranjan, and Sebastian Schaffert. Xcerpt and xchange - logic programming languages for querying and evolution on the web. In *ICLP*, pages 450–451, 2004.
14. F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *ICLP-99*, Cambridge, November 1999. MIT Press.
15. Jan Carlson and Björn Lisper. An interval-based algebra for restricted event detection. In *FORMATS*, pages 121–133, 2003.
16. Stefania Costantini and Arianna Tocchio. The dali logic programming agent-oriented language. In *JELIA*, pages 685–688, 2004.
17. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of semantics based on causal rejection. *Theory and Practice of Logic Programming*, 2:711–767, 2002.
18. Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits. A framework for declarative update specifications in logic programs. In *IJCAI*, 2001.
19. G.Cugola, E. D. Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *20th International Conference on Software Enginieering*, 1998.
20. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
21. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *5th International Conference on Logic Programming*. MIT Press, 1988.
22. C.A.R. Hoare. *Communication and Concurrency*. Prentice-Hall, 1985.
23. J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.
24. J. A. Leite and L. M. Pereira. Generalizing updates: from models to programs. In *LPKR'97: workshop on Logic Programming and Knowledge Representation*, 1997.
25. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *KR-92*, 1992.
26. J. W. Lloyd. *Foundations of Logic Programming*. Springer–Verlag, 2 edition, 1987.
27. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
28. S.Abiteboul, C.Culet, L. Mignet, B.Amann, T.Milo, and A. Eyal. Active views for electronic commerce. In *25th Very Large Data Bases Coference Proceedings*, 1999.
29. C. Sakama and K. Inoue. Updating extended logic programs through abduction. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *LPNMR-99*, Berlin, 1999. Springer.
30. Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5/6), 1996.
31. Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Amit - the situation manager. *The International Journal on Very Large Data Bases archive*, 13, 2004.
32. J. Sefranek. A kripkean semantics for dynamic logic programming. In *Logic for Programming and Automated Reasoning(LPAR'2000)*. Springer Verlag, LNAI, 2000.
33. J. Widom and S. Ceri, editors. *Active Database Systems – Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, 1996.
34. Y. Zhang and N. Y. Foo. Updating logic programs. In Henri Prade, editor, *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI-98)*, 1998.