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 possibility to express complex actions and events, and a declarative semantics. In this paper, we introduce ERA, an ECA language based on, and extending 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 **On** $Event$ **If** $Condition$ **Do** $Action$ 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 [26, 6], Semantic Web applications [21, 24], distributed systems [13], Real-Time Enterprize and Business Activity Management and agents [11].

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 from the occurrence of several basic ones. A widely used way for defining complex events is to rely on some event algebra [10, 1], i.e. to introduce operators that define complex events as the result of compositions of more basic ones that occur at the same or at different instants. Actions that are triggered by reactive rules may also 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 (in terms of sub-events and sub-actions), permits a simpler and

* This work has been partly funded by the European Commission under project Rewerse (http://rewerse.net). Thanks are due to Wolfgang May for his comments on previous versions.

M. Fisher et al. (Eds.): JELIA 2006, LNAI 4160, pp. 29–42, 2006.

more elegant programming style by breaking complex definitions into simpler ones and by allowing to use the definition of the same entity in different fragments of code.

An ECA language would also benefit from a declarative semantics taking advantage of the simplicity of its the basic concepts. 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 both relational 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 updates of 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 (for a detailed discussion see section 5). In particular, none provides the evolving capability, nor it is immediately clear how to incorporate such capability to these languages. 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 effort has been deployed 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) [2, 17, 5, 12], and update languages [3, 12, 17, 4] that 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 do they provide mechanism for specifying complex events or actions.

To overcome the limitations of both ECA and LP update languages, we present here an ECA language, defined by starting from DyLPs, called ERA (after Evolving Reactive Algebraic programs). This language builds on previous work on the update language Evolp [3], inheriting from it the evolving capabilities, and extending it with the possibility of defining and dealing with complex events and actions, and also considering external actions. The semantics of ERA is defined by means of an *inference system* (specifying what conclusions are derived by a program) and of an *operational semantics* (specifying the effects of actions). The former is derived from the refined semantics for DyLPs [2]. The latter is defined by a transition system inspired by existing work on process algebras. [22, 15].

The rest of the paper is structured as follows: we start in section 2 with an informal introduction to the language introducing its constructs and highlighting its main features. In section 3 we briefly introduce the syntax and semantics of DyLPs, and establish general notation. Section 4 is dedicated to the definition of the syntax and semantics of ERA. The main goals of the paper are the motivation for the language and its formal definition. A study of its properties and formal relation to other systems, cannot be presented here for lack of space. We nevertheless present some comparisons with related work in section 5, where we also draw conclusions and sketch future work.

2 Outline of the Language

Before the formal definition of ERA, which is given in section 4, we start here by informally introducing the various constructs of the language. As stated in the introduction, we aim at defining a language exhibiting both the advantages of ECA languages (with reactive rules, complex events and actions) and of LP updates (with inference rules, possibility of declaratively specifying self-updates). As such, 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*).

Reactive rules are as usual in ECA languages, and have the form (1), where: $Event$ is a basic or a complex event expressed in an algebra similar to the Snoop algebra [1]; $Condition$ is a conjunction of (positive or negative) literals and $Action$ is a basic or a complex action. *Inference rules* are LP rules with default negation, where default negated heads are allowed [19]. Finally, ERA also includes *inhibition rules* of the form:

When B **Do** $not\ Action$

where B is a conjunction of literals and events. Such an expression intuitively means: when B is true, do not execute $Action$. Inhibition rules are useful for updating the behaviour of reactive rules. If the inhibition rule above is asserted all the rules with $Action$ in the head are updated with the extra condition that B must *not* be satisfied in order to execute $Action$.

ERA allows to combine basic events to obtain complex ones by an event algebra. The operators we use are: $\triangle \mid \bigtriangledown \mid A \mid not$. Intuitively, $e_1 \triangle e_2$ occurs at an instant i iff both e_1 *and* e_2 occur at i; $e_1 \bigtriangledown e_2$ occurs at instant i iff either e_1 *or* e_2 occur at instant i; $not\ e$ occurs at instant i iff e *does not* occur i. $A(e_1, e_2, e_3)$ occurs at the same instant of e_3, in case e_1 occurred before, and e_2 in the middle. This operator is very important since it allows to combine (and reason with) events occurring at different time points.

Actions can also be basic or complex, and they may affect both the stored knowledge (internal actions) or the external environment. Basic external actions are related to the specific application of the language. Basic internal actions are for adding or retracting facts and *rules* (inference, reactive or inhibition rules), of the form $assert(\tau)$ and $retract(\tau)$ respectively, for raising basic events, of the form $raise(e)$. There is also an internal action $define(d)$ for adding new definitions of actions and events (see more on these definitions below). Complex actions are obtained by applying algebraic operators on basic actions. Such operators are: $\triangleright \mid \parallel \mid IF$, the first for executing actions *sequentially*, and the second for executing them *concurrently*. Executing $IF(C, a_1, a_2)$ amounts to execute a_1 in case C is true, or to execute a_2 otherwise.

For allowing for modularity on the definition of both complex actions and events, ERA allows for *event* and *action definition* expressions. These are of the form, respectively, e_{def} **is** e and a_{def} **is** a where e_{def} (resp. a_{def}) is an atom representing a new event and e (resp. a) is an event (resp. an action) obtained by the event (resp. action) algebra above. It is also possible to use defined events (resp. actions) in the definition of other events (resp. actions).

To better motivate and illustrate these various constructs of the language ERA, including how they concur with the features mentioned in the introduction, we present now an example from the domain of monitoring systems.

Example 1. Consider an (ECA) system for managing electronic devices in a building, viz. the phone lines and the fire security system. The system receives inputs such as signals of sensors and messages from employees and administrators, and can activate devices like electric doors or fireplugs, redirect phone calls and send emails. Sensors alert the system whenever an abnormal quantity of smoke is found. If a (basic) event $(alE(S))$[1], signaling a warning from sensor S occurs, the system opens all the fireplugs Pl in the floor Fl where S is located. This behaviour is encoded by the reactive rule

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

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 the complex action $fire\_alarmA$, which applies a security protocol: All the doors are unlocked (by the basic action $opendoorsA$) to allow people to leave the building; At the same time, a phone call is sent to a firemen station (by the action $firecallA$); Then the system cuts the electricity in the building (by action $turnA(elect, off)$). $opendoorsA$ and $firecallA$ can be executed simultaneously, but $turnA(elect, off)$ has to be executed after the electric doors have been opened. This behaviour is encoded by following definitions and rules

$$\begin{array}{l} alert2E(S_1, S_2) \ \textbf{ is } A(alE(S_1), alE(S_2), stop\_alertE) \bigtriangledown (alE(S_1) \bigtriangleup alE(S_2)). \\ fire\_alarmA \quad \textbf{ is } (opendoorsA \triangleright turnA(elect, off)) || firecallA. \\ \textbf{On } alert2E(S_1, S_2) \ \textbf{If } not\ same\_room(S_1, S_2) \textbf{ Do } fire\_alarmA. \\ same\_room(S_1, S_2) \leftarrow room(S_1, R_1), room(S_2, R_1). \end{array}$$

The last rule is already a simple example of an inference rule. For another example, suppose that we want to allow the system to be able to notify (by email) all members of a working group in some particular situation. Moreover suppose that working groups are hierarchically defined. Representing in ERA that if an employee belongs to a subgroup she also belongs to its supergroups, can be done by the inference rule[2]:

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

We provide now an example of *evolution*. Suppose the administrators decide to update the behaviour of the system such that from then onwards, when a sensor S raises an

[1] In the sequel, we use names of atoms ending in E to represent events, and ending in A to represent actions.

[2] The rules above uses recursion, on the predicate $ingroup/2$, a feature that is beyond the capabilities of many ECA commercial systems, like e.g. SQL-triggers [26].

alarm, only the fireplugs in the room R where S is located is opened. Moreover, each employee can from then onwards command the system to start redirecting phone calls to him (and to stop the previous behaviour of the systems regarding indirections, whatever they were. This behaviour is obtained by updating the system, asserting the following rules and definitions:

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

where τ_1 and τ_2 are the following rules:

τ_1 : **When** $phonE(Call), dest(Call,Emp)$ **Do** $not\ forwA(Call,N)$.
τ_2 : **On** $p\ honE(Call)$ **If** $dest(Call,Emp)$ **Do** $forwA(Call,Num)$.

The formal details of how to update an ERA system are given in section 4.2. Here, when $R1$ is asserted, if $alE(S)$ occurs in room R, any fire plug Pl which is not in R is not opened, even if Pl and S are on the same floor. Reactive rules R_2-R_3 encode the new behaviour of the system when an employee Emp commands 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 τ_1, τ_2. The former is an inhibition rule that inhibits any previous rule reacting to a phone call for Emp (i.e. 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 τ_1, τ_2 have to be asserted sequentially in order to prevent mutual conflicts. To revert to the previous behaviour it is sufficient to retract τ_1, τ_2 as done by action $stop\_redirectA$.

Such (evolution) changes could alternatively be done by handily modifying the previous rules ie, by *retracting* them and then *asserting* new rules. As with LP updates, also ERA offers the possibility to update reactive rules instead of rewriting. This possibility offered by ERA can be very useful in large systems developed and modified by several programmers and administrators, especially if updates are performed by users that are not aware of the existing rules governing the system, as in the previous example.

Having informally introduced the language, it is now time to start formalizing it. Before that some background on LP updates and notation is required.

3 Background and Notation

In what follows, we use the standard LP notation and, for the knowledge base, *generalized logic programs* (GLP) [19]. Arguments of predicates (here also called atoms) are enclosed within parentheses 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 its rules. Two rules τ and η are *conflicting* (denoted by $\tau \bowtie \eta$) iff the head of τ is the atom A and the head of η is $not\ A$, or vice versa.

A Dynamic Logic Program $\mathcal{P}$ over an alphabet $\mathcal{L}$ is a sequence $P_1, \ldots, P_m$ where the P_is 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 [2], 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 or generalized program [19]). The rationale for the definition of a refined model M of a DyLP is made according with the *causal rejection principle* [12, 17]: 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 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 $\mathcal{P}'..(P_i \cup E_i)$.

4 Formal Definition of ERA

4.1 Syntax of ERA Programs

We start the formal presentation of ERA by defining the syntax introduced in section 2.

Definition 2. *Let $\mathcal{L}$, $\mathcal{E}_B$, $\mathcal{E}_{def}$, $\mathcal{A}_X$ and $\mathcal{A}_{def}$ be sets of atoms respectively called:* condition alphabet*, set of* basic events*, of* event names*, of* external actions*, and of* action names*. 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:*

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

where τ (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 *$\mathcal{L}^{ERA}$ over $\mathcal{L}$, $\mathcal{E}_B$, $\mathcal{E}_{def}$, $\mathcal{A}_X$ and $\mathcal{A}_{def}$ is the triple $\mathcal{E}, \mathcal{L}, \mathcal{A}$. 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 $Cond$ 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** $Cond$ **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 expressions over $\mathcal{L}^{ERA}$.*

As in DyLPs, ERA considers sequences of programs, each representing an update (with asserted rules or definitions) of the previous ones. Such a sequence is called an *ERA dynamic program*, and determines, at each instant, the behaviour of the system. For this reason the semantics of ERA is given by ERA dynamic programs.

4.2 ERA Systems

The defined syntax allows to program reactive systems, hereafter called *ERA systems*. An ERA system has, at each moment, an ERA dynamic program describing and determining its behaviour, receives input (called *input program*) from the outside, and acts. The actions determine both the evolution of the system (by e.g. adding a new ERA program to the running sequence) and the execution in the external environment. Formally, an input program E_i, over an alphabet $\mathcal{L}^{ERA}$, is any set of 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[3] E_i. The sequence of programs $E_1, \ldots E_n$ denotes the sequence of input programs received at instants $1, \ldots, n$. A basic event e_b *occurs* at instant i iff the fact e_b belongs to E_i. We further assume that every input program contains event $truE$. This allows for defining reactive rules that are always triggered (reacting on event $truE$), or for expressing commands of updates to ERA systems, by having in the input program reactive rules reacting to $truE$ and with empty $true$ condition. For instance, updating the system of example 1 with rule R_1 is done by adding to the input program **On** $truE$ **If** $true$ **Do** $assert(R_1)$.

Since a complex event is obtained by composing basic events that occurred in distinct time instants (viz. when using operator A), for detecting the occurrence of complex events 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 4.3 and 4.4, 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 $raise$) of autonomously *raising* events in the future, future input programs are included as "passive" elements that are modified as effects of actions (see rule (2)).

The semantics of an ERA system specifies, at each instant, which conclusions are derived, which actions are 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 4.3.

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 $\langle \mathcal{S}, A\rangle \mapsto^G \langle \mathcal{S}', A'\rangle$ 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 a 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 4.4. At each instant an ERA system receives an input program, derives a new set of actions A_N and

[3] 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.

starts to execute these actions together with the residual actions not yet executed. As a result, the system evolves according to the transition relation [4] $\rightarrow$. Formally:

$$\frac{A_N = \{a_k \in \mathcal{A} : \ \mathcal{S} \ \vdash_e a_k\} \ \wedge \ \langle \mathcal{S}, (A \cup A_N)\rangle \mapsto^G \langle \mathcal{S}', A'\rangle}{\langle \mathcal{S}, A\rangle \rightarrow^G \langle \mathcal{S}', A'\rangle} \quad (1)$$

4.3 Inferring Conclusions

The inference mechanism of ERA is derived from the inference mechanism for DyLPs. In section 3, 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 basis of the refined semantics. From the 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 $\mathcal{L}^{ERA} : \ (\mathcal{E}, \mathcal{L}, \mathcal{A})$, 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, hence, are 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 $e_{def} \leftarrow e$.

Basically 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 $\bigtriangledown$ stands for the logic operators $\wedge$ and $\vee$. This is encoded by the following set of rules

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

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[5]: $AR(\mathcal{S}) =$

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

[4] Transition relation $\mapsto$ defines the effect of the execution of a set of actions, while $\rightarrow$ defines the global evolution of the system.

[5] 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 (i.e. the basic step of the recursion).

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[6]. Formally:

Definition 3. *Let $\vdash$ be an inference relation defined as in Section 3, and $\mathcal{S}$, $\mathcal{P}^R$, 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:*

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

We specified no rules for operator not . These rules are not needed since event (literal) $not\ e_p$ is inferred by default negation whenever there is no proof for e_p. The following theorem formalizes the intuitive meanings the various operators provided in section 4.1.

Proposition 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:*

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

4.4 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 (1) and the definition of $\vdash_e$, the semantics of ERA. As mentioned above, these transitions have the form: $\langle \mathcal{S}, A\rangle \mapsto^G \langle \mathcal{S}', A'\rangle$.

The effects of basic actions on the current ERA program are defined by the *updating function* $up/2$. Let $\mathcal{P}$ be an 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:

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

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

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

1st and 3rd rules below). The only exception involves action $raise(e_b)$ 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:

$$\begin{array}{rl} \langle(\mathcal{P}, E_P, E_i.E_F), \emptyset\rangle & \mapsto^{\emptyset} \ \langle(\mathcal{P}, E_P..E_i, E_F), \emptyset\rangle \\ \langle(\mathcal{P}, E_P, E_i.E_{i+1}.E_S), \{raise(e_b)\}\rangle & \mapsto^{\emptyset} \ \langle(\mathcal{P}, E_P..E_i, (E_{i+1} \cup \{e_b\}).E_F), \emptyset\rangle \\ \langle(\mathcal{P}, E_P, E_i.E_F), \{a_i\}\rangle & \mapsto^{\{a_i\}} \langle(up(\mathcal{P}, \{a_i\}), E_P..E_i, E_F), \emptyset\rangle \end{array}$$

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 $a_1 \triangleright a_2$, 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 $A = \{a_1, \ldots, a_n\}$ consists in the concurrent execution of all its actions a_k until the set of residual actions is empty.

The execution of $IF(C, a_1, a_2)$ amounts to the execution of a_1 if the system infers C, or to the execution of e_2 otherwise. Given an ERA system $\mathcal{S} = (\mathcal{P}, E_P, E_i.E_F)$ with $\mathcal{P} : P_1 \ldots P_n$, let $D(\mathcal{S})$ be 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 or more action definitions, corresponds to the concurrent executions of all the actions a_ks such that a_{def} **is** a_k belongs to $D(\mathcal{S})$. Formally:

$$\frac{\langle \mathcal{S}, \{a_1, a_2\}\rangle \mapsto^{G} \langle \mathcal{S}', A'\rangle}{\langle \mathcal{S}, \{a_1 \| a_2\}\rangle \mapsto^{G} \langle \mathcal{S}', A'\rangle} \quad \frac{\langle \mathcal{S}, \{a_1\}\rangle \mapsto^{G_1} \langle \mathcal{S}', A_1'\rangle}{\langle \mathcal{S}, \{a_1 \triangleright a_2\}\rangle \mapsto^{G_1} \langle \mathcal{S}', \{A_1' \triangleright a_2\}\rangle}$$

$$\frac{\langle \mathcal{S}, A_1\rangle \mapsto^{G_1} \langle \mathcal{S}', A_1'\rangle}{\langle \mathcal{S}, \{A_1 \triangleright a_2\}\rangle \mapsto^{G_1} \langle \mathcal{S}', \{A_1' \triangleright a_2\}\rangle} \quad \frac{\langle \mathcal{S}, \{a_2\}\rangle \mapsto^{G_2} \langle \mathcal{S}'', A_2''\rangle}{\langle \mathcal{S}, \{\emptyset \triangleright a_2\}\rangle \mapsto^{G_2} \langle \mathcal{S}'', A_2''\rangle}$$

$$\frac{\mathcal{S} \vdash_e C \ \wedge \langle \mathcal{S}, \{a_1\}\rangle \mapsto^{G_1} \langle \mathcal{S}', A'\rangle}{\langle \mathcal{S}, \{IF(C, a_1, a_2)\}\rangle \mapsto^{G_1} \langle \mathcal{S}', A_1'\rangle} \quad \frac{\mathcal{S} \nvdash_e C \ \wedge \langle \mathcal{S}, \{a_2\}\rangle \mapsto^{G_2} \langle \mathcal{S}'', A_2''\rangle}{\langle \mathcal{S}, \{IF(C, a_1, a_2)\}\rangle \mapsto^{G_2} \langle \mathcal{S}'', A_2''\rangle}$$

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

$$\frac{A = \{a_1, .., a_n\} \ \langle(\mathcal{P}, E_P, E_i.E_{i+1}.E_F), \{a_k\}\rangle \mapsto^{G_k} \langle(\mathcal{P}^k, E_P..E_i, E_{i+1}^k.E_F), A_k'\}\rangle}{\langle(\mathcal{P}, E_P, E_i.E_{i+1}.E_F), A\rangle \mapsto^{\cup G_k} \langle(up(\mathcal{P}, \bigcup G_k), E_P..E_i, \bigcup E_{i+1}^k.E_F), \bigcup A_k'\rangle}$$

As it is clear from this last rule, the definition of concurrent execution of actions in ERA does not rely on any concept of *serialization*. Actions may have, three different effects. Namely: to update the system, to rise new events, or to modify the external environment (by external actions). Semantically, internal updates are defined by function $up/2$ (see section 4.4) which is defined over an ERA dynamic program and a *set* of basic actions, while the raised events are added to the next input program and are then processed concurrently. No serialization is then needed for this kind of actions. Finally, the description and execution of external actions do not belong to the semantics of

ERA, since the meaning and effects of these actions depend on the application domains. Singular applications may require some notion of serialization for external actions (for instance, messages sent over the same communication channel are sent one by one.)

5 Conclusions and Related Work

We identified 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 performing self updates to data, inference rules and reactive rules. For this purpose we defined the logic programming ECA language ERA, 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). This new language is close in spirit to LP update languages like EPI [12], LUPS [4], Kabul [17] and, most significantly, Evolp [3]. All these languages have the possibility to update rules (though in EPI and LUPS only derivation rules can be updated). However, none of these supports external nor complex actions or complex events. In [16] Evolp has been extended to consider simple external actions, in the context of an agent architecture. The ERA language goes much beyond in the definition of complex actions and events. A formal comparison with Evolp, clearly showing how ERA is a proper extension of it, cannot be shown here for lack of space.

There exist several alternative proposals of ECA formalisms. Most of these approaches are mainly procedural like, for instance, AMIT [25] and JEDI [13] or at least not fully declarative [26]. A declarative situation calculus-like characterizations of active database systems is given in [6], although the subject of complex actions is not treated there. An example of a Semantic Web-oriented ECA languages is XChange [9], which also has a LP-like semantics, and allows to define reactive rules with complex actions and events. However, it does not support a construct similar to action definitions for defining actions, nor does it consider updates of rules. Updates of rules are also not part of the general framework for reactivity on the semantic web defined in [21]. Defining actions is a possibility allowed by the Agent-Oriented Language DALI [11], which in turn does not support complex events. Another related work is [23] which applies DyLPs to the agent language 3APL. Since 3APL is a language and architecture for programming BDI agents, this work is not directly relatable to ECA paradigms, although future comparisons with ERA could be interesting given the similarity of the semantics for KR. The ideas and methodology for defining complex actions are inspired by works on process algebras like CCS [22] and CSP [15] 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 [8] and Transaction Logic Programming (TLP) [7]. These 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, ones. However, TLP (and Dynamic Prolog) does not support complex events, nor does it cope with the possibility of receiving external inputs during the computation of complex actions. Finally, none of these ECA languages show update capabilities analogous to the ones of LP update languages, and that are also in ERA. As such, it is not

obvious how to provide a meaning to inhibition rules or exceptions to rules in those ECA languages.

The language ERA still deserves a significant amount of research. Preliminary investigations evidenced interesting properties of the operators of the action algebra like associativity, commutativity etc, and deserve further study. In this paper 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 [17, 5, 12]. In particular, we intend to develop a version of ERA based on the well founded semantics of DyLPs [5]. Well founded semantics [14] is a polynomial approximation to the answer set semantics that and is suitable for applications requiring the capability to quickly process vast amount of information. Implementations of the language are subject of ongoing research, where intend to take advantage of existing event-detection algorithms. For simplicity, here we presented a minimal set of operators for the event and action algebras. Specific application domains and confrontations with related languages may suggest eventual extensions of the language. For instance, the language GOLOG [18] presents an operator $\vee$, representing non deterministic choice between two actions which is not expressible in the current definition of ERA. We also plan to provide the possibility to execute ACID transactions in ERA and explore possible relations with Statelog [20].

References

1. Raman Adaikkalavan and Sharma Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. In *ADBIS*, pages 190–204, 2003.
2. 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.
3. 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.
4. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS: A language for updating logic programs. *Artificial Intelligence*, 132(1 & 2), 2002.
5. F. Banti, J. J. Alferes, and A. Brogi. The well founded semantics for dynamic logic programs. In Christian Lemaître, editor, *IBERAMIA-9*, LNAI, 2004.
6. Chitta Baral and Jorge Lobo. Formal characterization of active databases. In *Logic in Databases*, pages 175–195, 1996.
7. 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.
8. Anthony J. Bonner. A logical semantics for hypothetical rulebases with deletion. *Journal of Logic Programming*, 32(2), 1997.
9. F. Bry, P. Patranjan, and S. Schaffert. Xcerpt and xchange - logic programming languages for querying and evolution on the web. In *ICLP*, pages 450–451, 2004.
10. Jan Carlson and Björn Lisper. An interval-based algebra for restricted event detection. In *FORMATS*, pages 121–133, 2003.
11. Stefania Costantini and Arianna Tocchio. The DALI logic programming agent-oriented language. In *JELIA*, pages 685–688, 2004.
12. 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.
13. G.Cugola, E. D. Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *20th Int. Conf. on Software Enginieering*, 1998.

14. 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.
15. C.A.R. Hoare. *Communication and Concurrency*. Prentice-Hall, 1985.
16. J. Leite and L. Soares. Enhancing a multi-agent system with evolving logic programs. In K. Satoh K. Inoue and F. Toni, editors, *CLIMA-VII*, 2006.
17. J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.
18. Hector J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 1997.
19. 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.
20. B. Ludäscher, W. May, and G. Lausen. Nested transactions in a logical language for active rules. In D. Pedreschi and C. Zaniolo, editors, *Logic in Databases*, pages 197–222, 1996.
21. W. May, J. Alferes, and R. Amador. Active rules in the Semantic Web: Dealing with language heterogeneity. In *RuleML*, pages 30–44. Springer, 2005.
22. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
23. V. Nigam and J. Leite. Incorporating knowledge updates in 3APL - preliminary report. In R. Bordini, M. Dastani, J. Dix, and A. El F. Seghrouchni, editors, *ProMAS'06*, 2006.
24. 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.
25. Vijay A. Saraswat, Radha Jagadeesan, and Vineet Gupta. Amit - the situation manager. *The International Journal on Very Large Data Bases archive*, 13, 2004.
26. J. Widom and S. Ceri, editors. *Active Database Systems – Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, 1996.