TOWARDS CREATION OF LOGICAL FRAMEWORK FOR EVENT-DRIVEN INFORMATION SYSTEMS

Darko Anicic and Nenead Stojanovic
Forschung Zentrum Informatik (FZI), University of Karlsruhe, Germany

Keywords: Event-Condition-Action Rules, Reactive Systems, Rule-based Reasoning, Logic Programming.

Abstract: Event-Condition-Action (ECA) rules offer extensible and flexible approach to realizing active Enterprise Information Systems. Such systems are enabled to actively respond on events or state changes. Hence their behavior is programable by means of ECA rules. We propose an implementation of ECA rules in a completely logical framework, using *Transaction Datalog*¬ as an underlying logic. In this way, we extend the current ECA framework by means of powerful and declarative semantics, which also have an appropriate procedural interpretation. We show how a logical calculus of *Transaction Datalog*¬ can be exploited for realizing composite events, conditions, and actions; justifying the use of declarative semantics for solving some of the existing issues in reactive systems.

1 INTRODUCTION

Event-driven architecture (EDA) represents a new hype in enterprise information systems, that complements the service-oriented architecture. Event-driven applications trigger actions as a response to the detection of events. The event may signify a problem or an impending problem, an opportunity, a threshold, or a deviation. Upon generation, the event is immediately disseminated to all interested parties (humans or machines). The interested parties evaluate the event, and optionally take action. The event-driven action may include the invocation of a service, the triggering of a business process, and/or further information publication/syndication. EDA is the architecture of choice for implementing straight-through multistage business processes that deliver goods, services and information with minimum delay and maximal flexibility. It is a style of application architecture centered on asynchronous push-based communication leading to the so-called active enterprise information systems, that are able to react autonomously on various internal and external events. However, despite its enormous importance, this kind of systems is still missing a comprehensive mechanism for (formal) representation of event-action causality (usually coded in Event-Condition-Activity rules). One challenge is, for example, to formally represent two set of actions, triggered by the same event, that have to be synchronised with each other (e.g., one action from an ECA rule has to be accomplished in order to start an action from another ECA rule). Such and similar formal descriptions could than serve for executing different activities, and reasoning about behavioral aspect of reactive information systems.

In order to formalise reactive systems, we are proposing *Transaction Datalog*¬ ($\mathcal{TD}\neg$) (Bonner, 1998) to be used as an underlying logic. In fact, the main contribution is the extension of the standard ECA framework with the formal semantics of $\mathcal{TD}\neg$. We use deductive rules to define new implicit events or complex conditions and actions. Moreover we argue that the interaction between events and actions is possible and achievable by means of logic. As pointed in (Bry and Eckert, 2007b), use of rules for describing, rule based, ”virtual” events is highly desirable due to a number of reasons: rules serve as an abstraction mechanism and offer a higher-level event description. Rules allow for an easy extraction of different views of the same reactive system. Rules are suitable to mediate between the same events differently represented in various interacting reactive systems. Finally, rules can be used for reasoning about causal relationship between events. Similar argumentation for the rule-based events also applies for the condition and action part. Moreover integration of events, conditions, and actions by means of logic brings new possibilities in utilising behavioral aspects of event-

Anicic D. and Stojanovic N. (2008).
TOWARDS CREATION OF LOGICAL FRAMEWORK FOR EVENT-DRIVEN INFORMATION SYSTEMS.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - ISAS*, pages 394-401
DOI: 10.5220/0001716703940401
Copyright © SciTePress

driven enterprise information systems.

Our goal is to enable information systems to capture internal and external (relevant) changes. A system needs to properly react on changes in an automated manner. Such reaction can be seen as an act of change propagation. In general, an action changes the state of the system or triggers a new event. Therefore it is a question how to effectively control the whole reactive system, i.e. how machines can keep executing certain activities, ensuring at the same time the system consistency. In order to achieve this goal, we combine a reactive system (i.e., ECA system) with deductive capabilities in a Logic Programming style. First, our extension is motivated by the aim to synthesize *active behavior* (from ECA rules) with *procedural and deductive semantics* (from Transaction Datalog¬). Second, the extended framework integrates *reactive and continuous* behavior appropriately.

The paper is organised in the following manner: In the second section we give a short introduction into the ECA rules, whereas the third section introduces $TransactionDatalog\neg$. Our logic-based realisation of ECA rules is described in setcion four. Section five describes related work, whereas section six contains concluding remarks.

2 ECA FRAMEWORK

Event-Condition-Action rules have recently gained significant attention in information systems where reactive behavior is required, i.e., systems capable to detect events and respond to them automatically. The context, in which events are triggered, is also taken into account. The general form of ECA rules is: ”ON *Event* IF *Condition* DO *Action*”. Interpretation of a single ECA rule is to execute *Action* when *Event* occurs, provided that *Condition* holds. When a set of rules is considered, the interpretation is getting more complicated (Berstel et al., 2007). This, particularly holds, when taking into account the execution model of rules (i.e., behavior of rules at *run-time*). First, the rule set may contain conflicting rules. Second, there is an issue how to execute rules with respect to: ordering, priorities, granularity; such that, they still capture the intended meaning. For those reasons we use the semantics of ($\mathcal{TD}$), trying to create an ECA framework that is more robust and formal. In the following, we briefly describe the basic elements of an ECA rule.

Event. In general, reactive systems are recursive systems, where event is a central notion for driving the execution. Hence the role of events is, first, to identify situations in which the system is supposed to react, and second, to start the execution of ECA programs in an appropriate moment.

Condition. The context, in which an ECA rule fires, is described by the condition part. The condition part is usually represented as a query to a persistent knowledgebase. More importantly, the condition acts as a glue between different parts of a rule (i.e., event and action). In Section 4, we will introduce a more complex condition form, where the condition part may also be represented as any Datalog-like rule.

Action. The action part changes the state of a system. While events are triggered as a consequence of state changes, the actual state changes are caused by actions. Hence the reactive behavior of ECA systems is realized through the execution of actions. Typical examples of actions are: updating persistent data, calling a web service, triggering new events, committing a database transaction, or the rule base modification. In some cases, more simple (atomic) actions may be combined to form a *complex action*. An example of a complex action is a *sequence* of atomic actions. Further on, a complex action may be defined as a specification of *alternative* actions (i.e., if one action fails, the other one will start executing). In Section 4, we propose use of Transaction Datalog¬, in order to specify complex actions. Use of Transaction Datalog in an ECA framework, does not allow us only to create more complex actions, but also to enable coordination and cooperation between them. Communication, synchronization and concurrency between running actions are also supported.

3 OVERVIEW OF TRANSACTION DATALOG¬

In this section we give a short overview of Transaction Datalog, highlighting the language’s most important properties with respect to this paper.

Transaction Datalog. Transaction Datalog ($\mathcal{TD}$) is a subset of Transaction Logic ($\mathcal{TR}$) (Bonner and Kifer, 1995), and Concurrent Transaction Logic ($\mathcal{CTR}$) (Bonner and Kifer, 1996). Concurrent Transaction Logic is a general logic designed specifically for the rule-based paradigm, and intended to provide a declarative account for state-changing actions. Our motivation to use this particular logic, in an ECA framework, lies in a set of special logical and temporal connectives provided by $\mathcal{TD}$. For instance, a sequence of actions may be executed sequentially or in parallel. Further, actions may be synchronized and communicate among themselves (e.g., one action can read what another action writes). On the other side, an action may also be executed in isolation from the other actions. $\mathcal{TD}$ allows users to specify properties of state-changing programs, and to reason about them (Bonner and Kifer, 1995).

Transaction Logic has a ”Horn” fragment, with both, a procedural and declarative semantics (Bonner and Kifer, 1995). $\mathcal{TD}$ is derived from this Horn fragment, just like, classical Datalog is derived from classical Horn logic (Bonner, 1998). Compared with the full logic, $\mathcal{TD}$ has simpler semantics, hence gives a possibility for more effective implementation. At the same time, we will show (in Sections 4.1 and 4.3) that $\mathcal{TD}$ is still expressive enough to fulfil requirements for an expressive ECA framework.

Syntax. The syntax of Transaction Datalog is the same as the classical Datalog syntax (Ullman, 1990), extended with three additional predicate symbols: *p.empty*, *ins.p* and *del.p*. Intuitively, *p.empty* means ”*Is relation p empty*”, *ins.p* means ”*Insert atom p in the knowledgebase*”, and *del.p* means ”*Delete atom p from the knowledgebase*”. $\mathcal{TD}$, further, includes: *serial conjunction*, *concurrent conjunction*, and *modality of isolation*, respectively denoted $\otimes$, $\mid$, $\odot$. For example, if r, q, t, and w are atomic actions, than p is defined as $p(X) \leftarrow del.q(X) \otimes \odot(t(X) \mid w(X)) \otimes ins.r(X)$, is a complex action (represented as a Transaction Datalog rule). Executing the action p, intuitively, we perform three successive sub-actions: delete an atom $q(X)$, execute in parallel sub-actions, t and w, and finally, insert an atom $r(X)$ to the knowledge base. If any of the sub-action fails, the whole action will fail. In that situation, the system retracts to its initial state (i.e., a state before the execution started). This mechanism keeps the system always in a consistent state. The isolation operator is used to constrain communication between concurrently running sub-actions. For instance, consider a complex action $\varphi_1 \mid (\odot\varphi_2) \mid \varphi_3$. During the execution of such an action, sub-actions φ_1 and φ_3 may communicate between themselves. However φ_2 will be executed in isolation of them.

The following example from (Bonner, 1998), demonstrates another important feature of $\mathcal{TD}$, that is *synchronization*.

Example 3.
$actionA \leftarrow actionB \mid actionC.$
$actionB \leftarrow taskB_1 \otimes ins.startC_2 \otimes taskB_2 \otimes startB_3 \otimes taskB_3.$
$actionC \leftarrow taskC_1 \otimes startC_2 \otimes taskC_2 \otimes ins.startB_3 \otimes taskC_3.$

The first rule defines a complex *actionA*, which consists of two sub-actions: *actionB* and *actionC*. The two sub-actions execute concurrently, but not independently. In particular, each action performs three tasks, where $taskC_2$ (from *actionC*) cannot start until $taskB_1$ (from *actionB*) is finished. Similarly, $taskB_3$ cannot start until $taskC_2$ is completed (i.e., specified with $ins.startB_3$ from *actionC*, and an atom $startB_3$ from *actionB*). Therefore we see that *actionB* communicates with *actionC*. Moreover the two sub-actions, *actionB* and *actionC*, are synchronized between themselves.

Negation. Unlike (Bonner, 1998), we use Transaction Datalog embellished with *negation-as-failure* (Lloyd, 1989), thus extending $\mathcal{TD}$ to $\mathcal{TD}\neg$ (the same way as classical Datalog is extended to Datalog$\neg$). In Section 4, we will define complex ECA rules that possible include *negated* events and actions. When negation is allowed, there might not be a least fixed point (i.e., unique solution for a given set of rules), but several minimal fixed points (Ullman, 1990). In such situations, there is a problem to determine what is the actual meaning of the rules. We permit the *stratified negation* (Ullman, 1990), as with that form of negation, one can still choose an intuitive meaning in case of the non-unique solution. In addition, we should also mention that rules in $\mathcal{TD}$ may be recursive as in classical Datalog.

Execution Paths. Transaction Datalog provides a declarative account for *state-changing* actions. A *state* is defined as a finite set of ground atomic formulas with EDB predicates[1]. To model complex, concurrent actions, $\mathcal{TD}$ has a notion of *execution paths*, that records the execution history of the complex actions. Intuitively, the path represents periods of continuous execution, separated by periods of suspended execution (during which other action may execute) (Bonner and Kifer, 1996). In Section 4, we further discuss importance of the *execution paths* with respect to our ECA framework. Transaction Datalog has a *model theory* inherited from $\mathcal{TR}$, and an operational semantics based on a proof procedure with unification. For detailed analysis about syntax, semantics, and an inference system of $\mathcal{TD}$, we refer the reader to (Bonner, 1998; Bonner and Kifer, 1996).

4 IMPLEMENTING ECA RULES WITH $\mathcal{TD}\neg$

In this section we review the role of basic elements of an ECA rule (i.e., event, condition, action), putting them in a logical framework, and implementing them with Transaction Datalog$\neg$.

[1]As in classical Datalog, there are two sorts of predicates: base or EDB, and derived or IDB predicates.

4.1 Event

The execution in reactive systems is driven by events. We distinguish between an *internal* and an *external* event. The internal event occurs as a consequence of a state-change in the system (e.g., a fact has been inserted in the knowledgebase, a transaction committed, an exception occurred in the system etc.), while the external event is raised by a happening outside the system (e.g., caused by an external procedure, a sensor or an application)[2]. In either case, an event needs to be registered, such that, an ECA system can recognize the event and behave accordingly. For this purpose we use an *event ontology* (e.g., complex events are build out of atomic events by means of class relationship etc.).

Composite Event. Composite (or complex) event consists of atomic events that satisfy some predefined patterns. For instance, a pattern may be defined as a conjunction, disjunction, or negation of atomic events, followed by some temporal constraints (e.g., one event happened 10 min after another one). We use Transaction Datalog¬ to formally specify these patterns, and later on, to identify complex events by capturing the state-changes in the knowledgebase.

We assume a discrete time model, where ***time*** is an ordered set of time points. In this paper points are represented as integers, but other time models for time and data representation are possible without restrictions. The notion of an ***atomic event*** is defined as a *relevant state change*[3] in a system, characterized by the time. Formally, an event is $e(T_1,T_2,X_1,X_2,...,X_n), n \geq 0$, where e is an event name (i.e., a predicate symbol), and $T_1,T_2,X_1,X_2,...,X_n$ is a list of arguments. $X_1,X_2,...,X_n$ represent a set of data terms. Events contain data relevant for a reactive system. The data of event is a data term that may be either a variable, a constant, or a function symbol. T_1,T_2 defines a time interval during which the event has occurred. Following the argumentation from (Paschke et al., 2007), interval-based events are suitable for Complex Event Processing (CEP). For an atomic event $e_1(T_1,T_2,X_1,X_2,...,X_n)$, it appears that T_1 is equal to T_2. However consider a complex event e that is a sequence of events e_1, e_2, and e_3 in the following order: e_1 *before* (e_2 *before* e_3). If an event was not defined over a time interval (i.e., the detection time of the terminating event is used as occurrence time of the complex event), an inconsistency would occur due to the possibility to detect e as a sequence: e_1 *before* (e_2 *before* e_3) as well as for the sequence: e_2 *before* (e_1 *before* e_3). In order to prevent such unintended semantics, a complex event $e(T_1,T_2,X_1,X_2,...,X_n)$ that, for instance, consist of events $e_1(T_3,T_4,X_1,X_2,...,X_n)$ and $e_2(T_5,T_6,X_1,X_2,...,X_n)$ is defined over an interval $[T_1,T_2]$ where $T_1 = \min\{T_3,T_5\}$ and $T_2 = \max\{T_4,T_6\}$.

[2]Note that there is no strict difference between an (explicit) event and a state change (i.e., an implicit event).

[3]What is a relevant change depends on an application.

Definition 4.1 A complex event is a formula of the following form:

- an atomic event;
- $(event_1 \wedge event_2 \wedge ... \wedge event_n)$, where $n \geq 0$ and each $event_i$ is an event (Conjunctive composition);
- $(event_1 \vee event_2 \vee ... \vee event_n)$, where $n \geq 0$ and each $event_i$ is an event (Disjunctive composition);
- $(event_1 \otimes event_2 \otimes ... \otimes event_n)$, where $n \geq 0$ and each $event_i$ is an event (Sequential composition);
- $\neg event$, where *event* is an event (negation).
- $\odot event$, where *event* is an event (isolation).

A rule is a formula of the form $eventA \leftarrow eventB$, where *eventA* is an atomic event, and *eventB* is either an atomic or a complex event ■ In the above definition, every $event_i$ is defined over a time interval $[T_1,T_2]$ with possible set of data terms that are omitted due to space reasons.

In following examples we demonstrate the power of Transaction Datalog¬ language, and give justification for its use in our ECA framework.

Example 4.1 defines a complex event, *checkStatus*, which happens ”*if a priceChange event is followed with a stockBuy event*”. Further on, the two events have happened within a certain time frame (i.e., $t < 5$).

Example 4.1

$checkStatus(T_1,T_4,X,Y,Z,W) \leftarrow priceChange(T_1,T_2,X,Y) \otimes stockBuy(T_3,T_4,Z,Y,W) \wedge (T_4 - T_1 < 5)$.

In our system, we have an event ontology where the following has been defined[4]:

- $priceChange(T_i,T_j,X,Y)$ is an event, that describes the change in the stock price X (e.g., ±5%) of a company Y;
- $stockBuy(T_i,T_j,Z,W,Y)$ defines a transaction, in which, a buyer Z has bought W amount of stocks from a company Y.

[4]Note that, apart from events defined in the event ontology, we use a number of built-in predicates with predefined meaning (e.g., ”$-$” and ”$<$” represent ”subtraction” and ”less then” arithmetic operations).

In some cases a user may be interested in analyzing past events. For this purpose, we need a formalism that allow us, not only to create complex events, but also to query them. We have an *event system log*, which serves to record each event occurrence in the system. In the following example we ask for all events where the change in stock price was bigger than 10%.

Example 4.2
$? - priceChange(T_i, T_j, X, Y) \wedge X > 10$.

As all events are accumulated in the event system log, we can also describe situations where negated events are used. For instance, Example 4.3 represents a $notFulfilledOrder$ event, that triggers when a customer has made a purchase, but the purchase has not been delivered within a certain time. Therefore, we see, that the system is also capable to support *non-monotonic* features (i.e., the existence of an event which is defined in absence of other events, may be *retracted*, if one of the absent event occurs later). Note that, since the event stream is infinite, one should always define a time interval as a scope of a query, or a rule. In the Example 4.3 the interval in which we check whether an item has been delivered is $[T_3, T_4]$.

Example 4.3
$notFulfilledOrder(T_1, T_4, X) \leftarrow purchased(T_1, T_2, X) \otimes \neg \mathrm{delivered}(T_3, T_4, X) \wedge (T_4 - T_1 > 3)$.

Example 4.4
$complexEvent(T_1, T_4, X) \leftarrow eventA(T_1, T_2, X) \otimes eventB(T_3, T_4, X)$

In Example 4.4, we demonstrate use of the sequential composition operator (from Definition 4.1), i.e., a *complexEvent* will occur if *eventA* is followed by *eventB*. If *eventA* has happened, the system needs to "remember" it, and to raise *complexEvent* once *eventB* happens. Detection of complex events is done by following the state-change (transition) path. In general case, executing a Transaction Datalog rule, the system may change the state from S_1 to S_n (i.e., going through states: $S_1, S_2, ..., S_n$). If the rule describes a complex event, this state-transition may be seen as the progress towards detection of a complex event. In this way, if *eventA* has occurred but *eventB* has not, the system will *wait*, and raise *complexEvent* once *eventB* is triggered. Note, if we replace $eventB(T_3, T_4, X)$ with $eventA(T_3, T_4, X)$ from the example 4.1, we can define a repeated event (e.g., double click).

Finally, the modality of isolation operator $\odot$ is used for defining a composite event with additional constraints. Usually a composite event consists of (atomic) events that satisfy some pre-defined pattern. In this respect, a composite event is not dependant on all events monitored in the system, but those that constitutes that particular event. However the modality of isolation operator allow us to construct a composite event that is, apart from its (atomic) events, also constrained with other events from the system. For instance, a composite event e, defined as $e(T_1, T_6, X, Y) \leftarrow e_1(T_1, T_2, X) \otimes e_2(T_3, T_4, X, Y) \otimes e_3(T_5, T_6, X, Y)$, will be triggered if e_1, e_2, and e_3 happen next to each other with no other events in-between. Of course, a time interval for such composite events should be clearly defined as a scope over which events are monitored (i.e., $[T_1, T_6]$ in this case).

4.2 Condition

The condition part determines whether an ECA rule (triggered by a certain event) will be executed or not. The condition part is usually represented as a query, since it depends on the current state of the system. In our framework, the condition of an ECA rule may be significantly more complex.

Definition 4.2 The condition part of an ECA rule is any Datalog¬ rule, where the rule head is the condition name, and the rule body is the condition definition ■

The following $\mathcal{TD}$ formula represents an action, *act*, that will be executed iff the condition part, *cond*, is fulfilled:

$$\odot[cond \otimes act]$$

In this way we express a $\mathcal{TD}$ formula, that is at the same time, a Condition-Action rule. Hence the action, *act*, will be executed only if the condition, *cond* is fulfilled. Further on, the condition part (*cond*) may be extensively dependant on a number of other sub-conditions, as defined by Definition 4.2, and shown below:

$$cond \leftarrow cond_1 \wedge (cond_2 \vee cond_3) \wedge \neg cond_4.$$
$$cond_3 \leftarrow cond_1 \wedge cond_5.$$

4.3 Action

In general case, the purpose of the action part is to change the state of the system. An example of a state change is a single update in the knowledgebase. However atomic actions, such as data update, are too limiting in practise. More often we need to combine atomic actions into *complex actions*. We extend the standard ECA framework with deductive capabilities, such that the action part can be formally described with Transaction Datalog¬. First, our extension is motivated by the aim to integrate *active behavior* (from ECA rules) with *deductive capabilities* (from Transaction Datalog¬). Second, the extended framework integrate *reactive* and *continuous* behavior appropriately.

In the following, we give legal possibilities for creating a complex action out of atomic ones.

Definition 4.3 An action is a formula of the following form:

- an atomic action;
- ($action_1 \wedge action_2 \wedge \ldots \wedge action_n$), where $n \geq 0$ and each $action_i$ is an action (Conjunctive composition);
- ($action_1 \vee action_2 \vee \ldots \vee action_n$), where $n \geq 0$ and each $action_i$ is an action (Disjunctive composition);
- ($action_1 \otimes action_2 \otimes \ldots \otimes action_n$), where $n \geq 0$ and each $action_i$ is an action (Sequential composition);
- ($action_1 \mid action_2 \mid \ldots \mid action_n$), where $n \geq 0$ and each $action_i$ is an action (Concurrent composition);
- $\neg action$, where *action* is an action (negation).
- $\odot action$, where *action* is an action (isolation).

A rule is a formula of the form $actionA \leftarrow actionB$, where *actionA* is an atomic action, and *actionB* is either an atomic or a complex action ■

A rule may be seen as an action procedure, where the rule head is a complex action name, and the rule body is the action definition. Likewise events, actions are not just propositions, but contain data terms. Each action $a(X_1, X_2, \ldots, X_n)$ may be of arity n, $n \geq 0$, where $X_1, X_2, \ldots, X_n$ is a list of variables or constants, representing parameters of the action procedure.

Utilising $\mathcal{TR}$ operators, complex action may create complex processes that, at the end, may form a workflow (Bonner, 1999). As we can see, from Section 3 and Definition 4.3, actions may run in parallel possibly having non-serializable access to shared resources, or for instance, they can communicate and synchronise themselves.

As mentioned in Section 3, $\mathcal{TD}$ has a notion of *execution paths*. Execution paths show the way complex actions are executed, and record their execution history. Formally, an execution path of a complex action is represented as a finite sequence of pairs: $S_1S_2, S_3S_4, \ldots, S_{n-1}S_n$, where each state-change (i.e., S_iS_{i+1}) represents a period of an atomic action execution. For instance, imagine an event e_1 has occurred, and caused a corresponding complex action a_1 to start executing. The event e_1 has occurred when the knowledgebase was in the state S_1, and after the execution of the action a_1, the system will be brought to the state S_n. Since the a_1 is a complex action, the system will go through a set of states $S_1S_2, S_3S_4, \ldots, S_{n-1}S_n$. Every state transition, S_iS_{i+1}, corresponds to execution of an atomic action. Now suppose that during the execution of a_1, an event e_2 has occurred. The event e_2 has caused an action a_2 to happen, which is a simple action, and hence, will be completed before a_1. Note that a_2 will change the state of the whole system, while a_1 is still executing. Thanks to an *interleaving semantics* of $\mathcal{TD}$, after the execution of both actions, our whole ECA system will still remain in a consistent state. The interleaving semantics assumes a single execution path although a number of concurrent complex actions may be running at the same time. Every complex action consists of a sequence of sub-actions, where each sub-action changes a state of the system. However by interleaving these sequences, we obtain a new sequence of state changes, which is an execution path.

However some complex actions need to be executed continuously (i.e., without interruption by other sub-actions, or suspension). In this situation we use a *modality of isolation*. For example, consider a complex *actionA* which consists of three sub-actions:

$$actionA \leftarrow \odot\, [taskA_1 \otimes taskA_2 \otimes taskA_3]$$

An execution path of the above action is a single pair of states, S_1, S_2. Although *actionA* is a complex action, it has been modeled to change the state only from S_1 to S_2 (not as a sequence of changes: S_1S_2, S_2S_3, S_3S_4). Therefore if some other actions are executing at the same time, their execution paths will not be interleaved with the one of *actionA*. Therefore assuming that *actionA* started to execute first, the other actions will be committed after the *actionA*.

In the Example 3 (section 3) we have explained how a complex action can be implemented with two concurrently running, synchronised, actions. In case we want to synchronise the two actions with some (external) events, we may replace constructs for the synchronisation (i.e., $ins.startC_2$ and $ins.startB_3$) with some events $startC_2$, $startB_3$. In this way, our ECA framework features tighter integration between events and actions. Moreover this integration has been achieved at the logical level, which allows reasoning about actions and events. For instance a pattern such as ”notify me if an *actionA* happened before an *eventE*, and the *eventE* happened before an *actionB*” would be easy extractable by the reasoner.

4.4 Executing ECA Rules

In this section we give more details about basic concepts underlying our ECA framework with respect to the evaluation of ECA rules. We adhere to the general syntax of an ECA rule:

ON *Event* IF *Condition* DO *Action*

However our approach differs from existing frameworks in defining the *Event*, *Condition*, and *Action* part. As given by Definitions 4.1, 4.2, and 4.3, each component of an ECA rule may be defined as a set of either $\mathcal{TD}\neg$ or Datalog¬ rules.

ECA-based systems, with complex events, conditions, and actions; have been extensively studied elsewhere in (Paton and Díaz, 1999; Bry and Eckert, 2007a). One of the common issue in these systems is a specification of an *execution semantics* when a *set* of ECA rules is considered (rather than a single a rule). The execution semantics gives an answer how rules are treated at *run-time*. Imagine a situation where several events occur at the same time triggering several actions to happen (i.e., several ECA rules). The system does not know in which order to apply those rules. Different choices may lead to different executions. For these, and similar issues, various execution semantics have been proposed, e.g., selecting only one rule from the conflict set, executing all rules, rejecting the execution of any rule and triggering an exception etc. (Paton and Díaz, 1999)

Our approach is accomplished in a completely declarative framework (i.e., using the formal semantics of $\mathcal{TD}\neg$ language). We write ECA rules as a set of logical equations. These equations are specified declaratively (i.e., specifying *what* we want to achieve, rather than *how* we want to achieve that). Now, let us assume the same issue as earlier in this text (i.e., several ECA rules have been triggered at the same time). In our case, we deploy an inference engine to find an execution path. Precisely, the goal of the inference engine is to find a possible execution, for a given set of rules following the procedural semantics of $\mathcal{TD}\neg$ (Bonner and Kifer, 1996).

In conclusion, following the semantics of $\mathcal{TD}\neg$, the reasoner executes a set of ECA rules, executing at the same time, a set of (possibly) complex actions. These actions may run in parallel, or in a sequence (or combined). More importantly is that our framework has two-fold benefit of using semantics. The first advantage is deployment of an inference engine for finding a *legal execution path* among all possible choices of executions. Usually, the execution path depends on run-time properties of a logic program. Hence searching for an executable path (in run-time) is rather a reasoning task. Using declarative semantics, our framework is capable to accomplish this task in an automated manner. We see this approach, as more flexible, and in some real-world scenarios, as the only one. The second advantage is: executing a path (found by the reasoner), the system will never run into an *inconsistent* state.

ECA rules are considered as an appropriate form of *reactive rules* for a *distributed environment* (e.g., Web). However their use in a distributed environment may be very unpredictable, with respect to their intended semantics (Kifer et al., 2006). In general case, execution of an event may trigger other events, and these events may trigger even more events. There is neither guarantee that, such a chain of events will stop, nor that states (through which a distributed system passes) are valid. We see semantics as a means to establish some sort of a *consistency check* mechanism in an ECA framework. The purpose of this mechanism is to control state-changing actions, keeping the system always in a consistent state. By executing a set of complex ECA rules, our system changes its states. In this transition, every state in which the system enters, needs to be a legal state (with respect to the ECA rules and the semantics provided by $\mathcal{TD}$). However if the inference engine, searching for a possible execution path, enters to an illegal state (w.r.t the semantics of given rules), such a state-transition will be rolled back. In this way, our framework is an attempt to implement ECA rules in a completely logical framework, trying to achieve better run-time properties of the entire ECA system.

5 RELATED WORK

This section briefly overview current approaches in realizing an ECA framework, with the declarative semantics, and logical rules. Work on modeling *behavioral* aspect of an application (using various forms of reactive rules) has started in the Active Database community. Different aspects have been studied extensively, ranging from modeling and execution of rules, to architectural issues (Paton and Díaz, 1999). However, what is clearly missing in this work, is a clean integration of active behavior with *deductive* and *tem-*

poral capabilities. This is exactly a goal of our approach. Going in that direction, (Behrends et al., 2006) is an attempt which combines ECA rules with Process Algebra. The idea is to enrich the *action* part, with the declarative semantics of Process Algebra, particularly CCS algebra (Milner, 1983). Use of Process Algebra specification aims to enable the *reasoning* functionality (e.g., model checking) in such an ECA system. Recently, the event part of a rule has also been put in a logical framework (Bry and Eckert, 2007a). There, an event may be defined using reactive, but also, deductive rules. In (Paschke et al., 2007) a homogenous reaction rule language was given. The approach combines different paradigms such as reactive rules, declarative rules and integrity constraints. In conclusion, all previously mentioned studies, are motivated to use more formal semantics. Our approach may also be seen as an attempt towards that goal, though followed by a pure Logic Programming style.

6 CONCLUSIONS AND FUTURE WORK

We propose an expressive ECA framework that uses Transaction Datalog¬ as an underlying formalism. The framework clearly extends capabilities of Active Databases with declarative semantics, and power of rule-based reasoning. Further on, Active Databases usually combine two or more formalisms (e.g., SQL as a declarative language for querying, and Java, or some other high-level language, for procedural programming). Similarly, in (Behrends et al., 2006), Process Algebra has been chosen as a formalism for the complex action specification and execution. The event and condition part may possibly be specified by other languages (e.g., XPath/XQuery, Datalog, SPARQL, F-Logic etc.). Using Transaction Datalog¬, we provide a unified framework that is also clean and simple. Transaction Datalog does not make a sharp distinction between declarative and procedural programming. Therefore, our framework provides a seamless integration of these two programming styles, and allows specification of more complex events, conditions, and actions (e.g, events can be combined as a sequence of atomic events or actions can talk to each other or be nested and executed in parallel). We believe this approach is more pragmatic from the implementation and optimization point of view. For the next steps we will continue to formalise our ECA framework. Currently, we are working on practical reasoning procedures for $\mathcal{TD}\neg$. Later on, we plan to work on efficient algorithms for CEP with $\mathcal{TD}\neg$, as well as on the prototype implementation.

REFERENCES

Behrends, E., Fritzen, O., May, W., and Schenk, F. (2006). Combining eca rules with process algebras for the semantic web. In *RuleML*.

Berstel, B., Bonnard, P., Bry, F., Eckert, M., and Patranjan, P. L. (2007). Reactive rules on the web. In *Reasoning Web*. Springer.

Bonner, A. J. (1998). Transaction datalog: A compositional language for transaction programming. In *Database Programming Languages*. Springer.

Bonner, A. J. (1999). Workflow, transactions and datalog. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM.

Bonner, A. J. and Kifer, M. (1995). Transaction logic programming (or, a logic of procedural and declarative knowledge. In *Technical Report CSRI-270*.

Bonner, A. J. and Kifer, M. (1996). Concurrency and communication in transaction logic. In *Joint International Conference and Symposium on Logic Programming*. MIT Press.

Bry, F. and Eckert, M. (2007a). Rule-based composite event queries: The language xchangeeq and its semantics. In *RR*. Springer.

Bry, F. and Eckert, M. (2007b). Towards formal foundations of event queries and rules. In *Second Int. Workshop on Event-Driven Architecture, Processing and Systems EDA-PS*.

Kifer, M., Bernstein, A., and Lewis, P. (2006). *Database Systems - An Application-Oriented Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

Lloyd, J. W. (1989). *Foundations of Logic Programming*. Computer Science Press.

Milner, R. (1983). Calculi for synchrony and asynchrony. In *Theor. Comput. Sci.*

Paschke, A., Kozlenkov, A., and Boley, H. (2007). A homogenous reaction rules language for complex event processing. In *International Workshop on Event Drive Architecture for Complex Event Process*. ACM.

Paton, N. W. and Díaz, O. (1999). Active database systems. In *ACM Comput. Surv.* ACM.

Ullman, J. D. (1990). *Principles of Database and Knowledge-Base Systems, Volume I*. W. H. Freeman & Co., New York, NY, USA, 2nd edition.