Expressive Logical Framework for Reasoning about Complex Events and Situations

Darko Anicic
FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe
roland.stuehmer@fzi.de

Nenad Stojanovic
FZI Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe
ljiljana.stojanovic@fzi.de

Abstract

Business and enterprise management processes become more and more *event-driven*. Being event-driven means that processes rely on receiving events to monitor the execution progress, and issuing events to initiate its next stages. What is more important, being event-driven enables these processes to become more active and flexible. There is a paradigm shift today from passive processes to *active* ones (based on push of information rather than on pull of information and knowledge). Also the management of business processes needs to be *flexible*, i.e, to adapt to ad hoc changes (events) during operations. All these requirements suggest use of Complex Event Processing (i.e., event patterns, event pattern rules, and event constraints) for realising event-driven business processes. We implement these concepts in a completely logical framework using Concurrent Transaction Logic ($\mathcal{CTR}$). Particularly, $\mathcal{CTR}$ is used for specifying, reasoning, and executing event-driven activities.

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. An event may signify a problem or an impending problem, an opportunity, a threshold, or a deviation. Upon generation, an event is immediately disseminated to all interested parties (humans or machines). The interested parties evaluate the event, and optionally take an 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 basic concepts of Complex Event Processing (CEP), i.e. *event patterns*, *contexts* in which events are to be detected, *event-action causality* (usually coded as Event-Condition-Action rules) and *event constraints*. It is challenging to represent these basic concepts in an appropriate form, and to give them formal semantics (such that reasoning about behavioral aspect of event-driven information systems is possible). An event-driven system is a reactive system which is hard to control. Many issues with respect to termination of the system may arise (e.g., it is not clear whether the system will ever terminate). Also the order of triggered actions is important (e.g., it is not clear in which order two or more actions triggered by the same event pattern will execute, and how that order may influence the whole system). For these and similar reasons formal semantics is necessary as a control mechanism in an event-driven reactive system.

In order to formalise event-driven systems, we propose Concurrent Transaction Logic ($\mathcal{CTR}$) (Bonner and Kifer 1996) to be used as an underlying logic. We implement basic concepts of Complex Event Processing (CEP), i.e., event patterns, ECA rules (including event pattern rules), and event constraints with $\mathcal{CTR}$. We use deductive rules to define new implicit event patterns or complex contexts (conditions) and actions, and 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" event patterns 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. As for the rule-based event patterns, similar argumentation applies for the condition and action part of ECA rules. Moreover integration of event patterns and constraints as well as contexts, and actions by means of logic brings new possibilities in utilising behavioral aspects of event-driven enterprise information systems.

The paper is organised in the following manner: In Section *CTR Overview*, we introduce $\mathcal{CTR}$ as an underlying

formalism for logic-based event processing. We implement basic concepts of CEP, namely event patterns in Section *Event Pattern*; reactive rules in Section *Implementing ECA Rules with* $\mathcal{CTR}$; and finally event pattern constraints in Section *Event Pattern Constraints*. Section *Related Work* describes related work, whereas Section *Conclusions and Future Work* contains concluding remarks.

CTR Overview

Concurrent Transaction Logic ($\mathcal{CTR}$) (Bonner and Kifer 1996) is a general logic designed specifically for the rule-based paradigm, and intended to provide a declarative account for state-changing actions. A model and proof theory of $\mathcal{CTR}$ has been defined in (Bonner and Kifer 1996). It is an extension of Transaction Logic ($\mathcal{TR}$) (Bonner and Kifer 1995), with its "Horn" fragment defined. In this section we will informally introduced syntax and semantics of $\mathcal{CTR}$. For further considerations related to the semantics of $\mathcal{CTR}$ the reader is referred to (Bonner and Kifer 1995).

Syntax. The atomic formulas of $\mathcal{CTR}$ has the form $p(t_1, ..., t_n)$, where p is a predicate symbol and the t_i's are function terms. More complex formulas are built using connectives and quantifiers.

The same as in classical logic, $\mathcal{CTR}$ has $\wedge, \vee, \neg, \forall$, and $\exists$. Unlike in classical logic, $\mathcal{CTR}$ has two connectives, $\otimes$ (serial conjunction) and $|$ (concurrent conjunction); operators: $\diamond$ (executional possibility), $\Box$ (executional necessity) and $\odot$ (modality of isolation, i.e., isolated execution).

Informal Semantics. In short, $\mathcal{CTR}$ is a logic for state-changing actions. The truth of $\mathcal{CTR}$ formulas is determined over *paths*. A path is a finite sequence of states. If a formula, ψ, is true over a path $< s_1, ..., s_n >$ it means that ψ can be executed[1] starting with state s_1. During the execution, ψ will change the current state to $s_2, s_3, ...$ and finally terminate at the state s_n. Having this in mind, the intended semantics of $\mathcal{CTR}$ connectives and modal operators can be summarised as follows:

- $\phi \otimes \psi$ means: execute ϕ, then execute ψ;
- $\phi \mid \psi$ means: execute ϕ and ψ concurrently;
- $\phi \wedge \psi$ means: ϕ and ψ must both be executed along the same path;
- $\phi \vee \psi$ means: execute ϕ or ψ nondeterministically;
- $\neg\, \phi$ means: execute in any way, provided that this will not be a valid execution of ϕ;
- $\odot\, \phi$ means: execute ϕ in isolation of other possible concurrently running activities;
- $\diamond\, \phi$ means: check whether it is possible to execute ϕ at the current state;
- $\Box\, \phi$ means: check necessity to execute ϕ at the current state.

[1] $\mathcal{CTR}$ has a notion of *executional entailment* over that *execution* path, i.e., a logical account of transaction formulas execution (Bonner and Kifer 1995).

The logic has notions of *data* and *transition* oracles. The *data* oracle, $\mathcal{O}^d(\mathbf{D})$, is used to solve queries related to a particular state $\mathbf{D}$. Likewise, the *transition* oracle, $\mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$, is used to specify an update (transition). If $a \in \mathcal{O}^t(\mathbf{D}_1, \mathbf{D}_2)$, then a is an elementary update that changes state $\mathbf{D}_1$ into state $\mathbf{D}_2$. For example, the state change can happen if an atom or a Horn rule is inserted or deleted from $\mathbf{D}$.

For further details about $\mathcal{CTR}$ semantics, its "Horn" fragment etc., the reader is referred to (Bonner and Kifer 1996), (Bonner and Kifer 1995), (Roman and Kifer 2007).

Event Pattern

The execution in CEP applications is driven by events. In order to react on an event, we need first to specify event patterns which we want our application to react on. An *event pattern* is a template which matches certain events. For example, an event pattern matches all orders from customers in response to a discount announcement event. Sometimes the discount announcement event is called an *atomic* event, which is used to build a *complex* event. In general, an event pattern is a pattern which is built out of (atomic and complex) events satisfying certain relational operators, contexts, and data parameters. Relational operators typically depend on a particular language used for CEP. For example they can be a conjunction, disjunction, causality dependencies, or negation of events. A relational operator is further followed by a *context*. A context of an event pattern is a constraint. It can be a temporal constraint (e.g., one event happened 10 min after another one), or more generally the state of a reactive system (see Section *Context*). Data parameters are terms that specify data or variables carried by an event pattern. We use $\mathcal{CTR}$ to formally specify these patterns, and later on, to implement other fundamental concepts for CEP.

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. Since $\mathcal{CTR}$ is a state-changing logic (see Section *CTR Overview*), the notion of an event pattern is defined as a *relevant state change*[2] in an event-driven 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 name of an event pattern (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 parameters (terms). Event patterns contain data relevant for a reactive system. The data of an event pattern 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, Kozlenkov, and Boley 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 an event pattern 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

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

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, an event pattern $e(T_1, T_2, X_1, X_2, ..., X_n)$ 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$\{T_4, T_6\}$.

Definition (Event Pattern). An event pattern is a formula of the following form:

- an atomic event;
- $(event_1 \vee event_2 \vee ... \vee event_n)$, where $n \geq 0$ and each $event_i$ is an event pattern (Disjunctive composition);
- $(event_1 \otimes event_2 \otimes ... \otimes event_n)$, where $n \geq 0$ and each $event_i$ is an event pattern (Sequential composition);
- $(event_1 | event_2 | ... | event_n)$, where $n \geq 0$ and each $event_i$ is an event pattern (Concurrent composition);
- $\neg event$, where $event$ is an event pattern (negation).
- $\odot event$, where $event$ is an event pattern (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 pattern ■

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 $\mathcal{CTR}$, and give justification for its use in processing event patterns.

Example 3.1 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 3.1.
$checkStatus(T_1, T_4, X, Y, Z, W) \leftarrow priceChange(T_1, T_2, X, Y) \otimes stockBuy(T_3, T_4, Z, Y, W) \otimes (T_4 - T_1 < 5)$.
assuming we have defined the following event patterns[3]:

- $priceChange(T_i, T_j, X, Y)$ is an event pattern, that describes the change in the stock price X (e.g., $\pm 5\%$) 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.

In some cases a user may be interested in analyzing past events. For this purpose, we need a possibility, not only to create patterns, but also to query them. In the following example we ask for all events from past where the change in stock price was bigger than 10%.

[3]Note that for creating event patterns, we also use a number of built-in predicates with predefined meaning (e.g., "$-$" represents "subtraction" and "$<$" denotes the comparison operations "less then").

Example 3.2.
$? - priceChange(T_i, T_j, X, Y) \otimes X > 10$.

It is possible also to describe negated events. For instance, Example 3.3 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 3.3 the interval in which we check whether an item has been delivered is $[T_3, T_4]$.

Example 3.3.
$notFulfilledOrder(T_1, T_4, X) \leftarrow purchased(T_1, T_2, X) \otimes \neg$delivered$(T_3, T_4, X) \otimes (T_4 - T_1 > 3)$.

Example 3.4.
$complexEvent(T_1, T_4, X) \leftarrow eventA(T_1, T_2, X) \otimes eventB(T_3, T_4, X)$

Example 3.4, demonstrates (as well as previous examples) use of the sequential composition operator (from Definition *Event Pattern*), 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 event patterns is done by following the state-change (transition) path. In general case, executing a $\mathcal{CTR}$ 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 an event pattern, this state-transition may be seen as the progress towards detection of that event pattern. In this way, if $eventA$ has occurred but $eventB$ has not, the system will *wait*, and raise $complexEvent$ once $eventB$ is triggered.

Finally, the modality of isolation operator $\odot$ is used for defining event patterns with additional constraints. Usually a complex event consists of (atomic or complex) events that satisfy some pre-defined pattern. In this respect, that complex event is not dependent on all events (monitored in an event-driven system or an event cloud), 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 constituting events, also constrained with other events from the system. For instance, a composite event e, defined as $e(T_1, T_6, X, Y) \leftarrow \odot(e_1(T_1, T_2, X) \otimes e_2(T_3, T_4, X, Y) \otimes e_3(T_5, T_6, X, Y))$, 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).

Context

Context is represented by the *state* of a reactive system. It can contain any number of Boolean conditions; or a database containing all facts that are true in that state. Moreover a

context can be represented with implicit information stated in form of rules. This in turn allows an inference engine to reason about the current context, prior to initiating an action for execution or raising an event. Process of discovering (reasoning about) the current context is necessary since reaction (i.e., another event pattern or an action) is valid only in a certain context. One particular event pattern can be interpreted in different ways, and hence can trigger different actions or events, when occurred in different situation. We say that one event in a particular context consists one particular *situation*.

In the context of ECA rules (see Section *Implementing ECA Rules with $\mathcal{CTR}$*), context is related to the condition part of an ECA rule. It determines whether the rule (triggered by a certain event pattern) will execute or not. The condition part is usually represented as a query. In our case, the context (and hence the condition of an ECA rule) may be more complex. It is an inference rule, therefore it includes implicit information too.

Definition (Context). The context is defined by any Datalog$\neg$ rule, where the rule head is the context name, and the rule body is the context definition ■

In the following example we show a complex context which is dependent on a few other sub-conditions and tests.

$ctx(T, T_1, T_2) \leftarrow ctx_1(T_1) \wedge (cond_2(T_2) \vee cond_3(T_2)) \wedge \neg cond_4(T, T_1, T_2) \wedge (T_2 - T_1 < 10sec.$
$ctx_3(T) \leftarrow ctx_1(T) \wedge ctx_5(T).$

Processing of complex contexts may affect performance of event processing engines. Many of "state of the art" event pattern rule languages cannot express complex contexts (providing only the condition part for querying the data). The approach based on $\mathcal{CTR}$[4] supports not only specification of complex constraints, but also their evaluation (since it is based on logic) and reasoning about event patterns in particular contexts (i.e., situations).

Implementing ECA Rules with $\mathcal{CTR}$

Event-Condition-Action rules are used in event-driven information systems where reactive behavior is required, i.e., systems capable to detect events and respond to them automatically (Paton, Schneider, and Gries 1999). 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.

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 $\mathcal{CTR}$.

Event and Condition

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; second, to start the execution of ECA programs in an appropriate moment.

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).

The event and condition parts of an ECA rule are implemented in $\mathcal{CTR}$ as shown in the previous section. The implementation of the event part corresponds to the implementation of the event pattern. Similarly the condition part corresponds to the context.

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 general case, the purpose of the action part is to change the state of the system. An example of the 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 $\mathcal{CTR}$. First, our extension is motivated by the aim to integrate *active behavior* (from ECA rules) with *deductive capabilities* (from $\mathcal{CTR}$). Second, the extended framework integrate *reactive* and *continuous* behavior appropriately.

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

Definition (Action). An action is a formula of the following form:

- an atomic action;
- $(action_1 \vee action_2 \vee ... \vee action_n)$, where $n \geq 0$ and each $action_i$ is an action (Disjunctive composition);
- $(action_1 \otimes action_2 \otimes ... \otimes action_n)$, where $n \geq 0$ and each $action_i$ is an action (Sequential composition);
- $(action_1 \mid action_2 \mid ... \mid action_n)$, 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 $a(X_1, X_2, ..., X_n)$

[4]Datalog$\neg$ is a subset of $\mathcal{CTR}$, and Datalog$\neg$ like rules (used to express the context) are called database rules in $\mathcal{CTR}$.

may be of arity n, $n \geq 0$, where $X_1, X_2, ..., X_n$ is a list of variables or constants, representing parameters of the action procedure.

Utilising $\mathcal{CTR}$ operators, complex actions may create complex processes that, at the end, may form a workflow (Bonner 1999). As we can see, from Section *CTR Overview* and Definition *Action*, 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 *CTR Overview*, $\mathcal{CTR}$ 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: $S_1S_2, S_3S_4, ..., S_{n-1}S_n$, 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 $S_1S_2, S_3S_4, ..., S_{n-1}S_n$. 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 $\mathcal{CTR}$, after the execution of both actions, the whole ECA system will still remain in the 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:

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

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*.

The following example from (Bonner and Kifer 1995), demonstrates another important feature of $\mathcal{CTR}$, that is *synchronization*.

Example (Synchronization).
actionA ← *actionB* | *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.

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 $startC_2$, $startB_3$, and hence achieve 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 some *actionB*" would be easy extractable by the reasoner.

Executing ECA Rules

Once the consisting elements of an ECA rule are defined, we can glue them together demonstrating how ECA rules are implemented in $\mathcal{CTR}$. Let us observe first the condition-action (C-A) pattern of an ECA rule. To implement this pattern one can utilise an implementation of **if-then-else** statement in declarative way, as shown in (Bonner, Kifer, and Consens 1994):

$$\begin{aligned} if\_a\_b\_c &\leftarrow (\diamond a) \otimes b \\ if\_a\_b\_c &\leftarrow (\Box\neg a) \otimes c \end{aligned} \tag{1}$$

The above rules executes b if a condition a is satisfied, else c will execute[5]. Further on, only **if-then** statement can be represented with the following:

$$\begin{aligned} if\_a\_b\_c &\leftarrow (\diamond a) \otimes b \\ if\_a\_b\_c &\leftarrow \Box\neg a \end{aligned} \tag{2}$$

In the same way we can write a C-A rule:

$$\begin{aligned} c\_a\_rule &\leftarrow (\diamond cond) \otimes act \\ c\_a\_rule &\leftarrow \Box\neg cond \end{aligned} \tag{3}$$

We shell write **if cond then act** as an abbreviation for the proposition c_a_rule. So the complete ECA rule is then:

$$\begin{aligned} event(X) &\leftarrow c\_a\_rule \\ c\_a\_rule &\leftarrow \textbf{if}\ cond\ \textbf{then}\ act \end{aligned} \tag{4}$$

If there are n actions to be triggered on $event(X)$ we write:

$$event(X) \leftarrow c\_\_a\_rule_1 | c\_\_a\_rule_2 | ... | c\_\_a\_rule_n \tag{5}$$

[5] $if\_a\_b\_c$ is a name of this particular **if-then-else** statement. The name should not be used as a head of any other rule.

providing for each action a condition:

$$\begin{aligned} c\_a\_rule_1 &\leftarrow \textbf{if } cond_1 \textbf{ then } act_1 \\ c\_a\_rule_2 &\leftarrow \textbf{if } cond_2 \textbf{ then } act_2 \\ &\ldots \\ c\_a\_rule_n &\leftarrow \textbf{if } cond_n \textbf{ then } act_n \end{aligned} \tag{6}$$

The rule in (5) executes n actions in parallel. This means that there is no a preferred order over the execution of actions. However in some situation it is desirable to trigger actions in a certain order (by implementing a special policy of an application). For example, actions need to execute in reverse order (from the rule in (5)). In that situation, $\mathcal{CTR}$ offers an easy way to implement such a conflict resolution strategy:

$$event(X) \leftarrow c\_a\_rule_n \otimes c\_\_a\_rule_{n-1} \otimes \ldots \otimes c\_\_a\_rule_1 \tag{7}$$

Rules from (4) and (7) demonstrate how ECA rules can be completely implemented in a declarative way. What is a benefit of implementing them declaratively? ECA rules are considered as an appropriate form of *reactive rules*. However their use in practice very often can be unpredictable, with respect to their intended semantics (Kifer, Bernstein, and Lewis 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 reactive system passes) are valid. We see semantics as a means to establish some sort of a *consistency check* mechanism in an ECA reactive system. The purpose of this mechanism is to control state-changing actions, keeping the system always in the consistent state. By executing a set of complex ECA rules, a reactive system implemented with $\mathcal{CTR}$ 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 $\mathcal{CTR}$). 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 sense ECA rules implemented in a logical framework, allow for an easier control over the entire event-driven system during the run-time.

Event Pattern Rules

An event pattern rule is a reactive rule triggered whenever an event pattern is matched. In that situation an action specified by the rule is executed, and as an acknowledgment of that act a new event is triggered (Luckham 2001). Hence an event pattern rule establishes a causal relationship between two events: an event that triggered the rule, and an event that is created when the rule executes its action. It is obvious that an event pattern rule is a special case of an ECA rule. Therefore an implementation of event pattern rules amounts to the implementation of ECA rules in $\mathcal{CTR}$ (already discussed in this section), provided that an action is always to trigger another event.

Event Pattern Constraints

An event pattern constraint expresses a condition which must be satisfied by the events observed in an event-driven system. According to (Luckham 2001), a constraint is used to test certain patterns of events that *never happen* in the system. For example, with a constraint we express a policy that an order from a customer must never be first confirmed and later denied. We can take a broader definition of a constraint. Apart from expressing events that must never happen, constraints can also guarantee which events *must happen*. So constraints serve to force an event-driven system to behave in a way specified by constraints (e.g., a particular event pattern must happen, otherwise the current execution is not be possible).

Implementation of a powerful event constraint algebra with $\mathcal{CTR}$ have already been specified in (Roman and Kifer 2007; 2008). Therefore we will not go into details of $\mathcal{CTR}$ event constraints here. Rather for the sake of completeness we show that $\mathcal{CTR}$ is a unifying logic formalism suitable for CEP, including event constraint handling too. Example *Constraints* illustrates a couple of event constraints that can be expressed in $\mathcal{CTR}$. For the further details, a reader is referred to (Roman and Kifer 2007; 2008). Since the truth values of $\mathcal{CTR}$ formulas are defined on paths (see Section *CTR Overview*), let us denote an event constraint e (which needs to occur somewhere on an execution $path$) with $path \otimes e \otimes path$ (abbreviated ∇e). Then the following formulas are all legal event constraints implemented in $\mathcal{CTR}$.

Example (Constraints).
$\nabla e_1 \vee \neg\nabla e_2$ - If event pattern e_1 occurs, then e_2 must also occur (before or after e_1);
$\neg\nabla e_1 \vee (\nabla e_1 \otimes \nabla e_2$ - If event pattern e_1 occurs, then e_2 must occur after e_1;
$\neg\nabla e_1 \vee \odot(e_1 \otimes e_2)$ - If event pattern e_1 occurs, then e_2 must occur right after e_1 with no event in-between.

Related Work

In order to capture relevant changes in a system and respond to those changes adequately, a number of logic-based reactive frameworks have been proposed. This section briefly overview some approaches to processing events and reactive 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 *temporal* 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. Further on, IBM has been developed an event processing tool available in IBM Tivoli Enterprise Console. The engine is capable to process event patterns and executes ECA rules. The approach uses Prolog programming language to implement a formalism for specifying event patterns, event

filters, and actions. Although the formalism itself is very expressive, the whole approach is rather complex. Recently, event patterns have been put in a logical framework in (Bry and Eckert 2007a). There, an event may be defined using reactive, but also, deductive rules. In (Paschke, Kozlenkov, and Boley 2007) a homogenous reaction rule language was proposed. 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. $\mathcal{CTR}$ approach may also be seen as an attempt towards that goal, though followed by a pure Logic Programming style.

Conclusions and Future Work

We propose an expressive approach for Complex Event Processing based on a logic, particularly Concurrent Transaction Logic. The approach 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. Event patterns and contexts are than specified by other languages (e.g., XPath/XQuery, Datalog, SPARQL, F-Logic etc.). Using $\mathcal{CTR}$, it is possible to implement a unified framework that is also clean and simple. $\mathcal{CTR}$ does not make a sharp distinction between declarative and procedural programming. Therefore, the proposed framework provides a seamless integration of these two programming styles, and allows specification of more complex event patterns, contexts (conditions), actions, and event constraints. We believe this approach is more pragmatic from the implementation and optimization point of view. For the next steps we will continue to study advantages and drawbacks of Complex Event Processing implemented in a logical framework, and work on an implementation for $\mathcal{CTR}$-based event processing.

References

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

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.

Bonner, A. J.; Kifer, M.; and Consens, M. 1994. Database programming in transaction logic. In *DBLP-4: Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages*. Springer-Verlag.

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.

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*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd edition.

Luckham, D. C. 2001. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc.

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.

Paton, N. W.; Schneider, F.; and Gries, D. 1999. *Active Rules in Database Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1st edition.

Roman, D., and Kifer, M. 2007. Reasoning about the behavior of semantic web services with concurrent transaction logic. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowmentg.

Roman, D., and Kifer, M. 2008. Semantic web service choreography: Contracting and enactment. In *International Semantic Web Conference*. Springer.