A procedure for an Event-Condition-Transaction language

Ana Sofia Gomes and José Júlio Alferes [⋆]

NOVA-LINCS - Dep. de Informática, Faculdade Ciências e Tecnologias
Universidade NOVA de Lisboa

Abstract Event-Condition-Action languages are the commonly accepted paradigm to express and model the behavior of reactive systems. While numerous Event-Condition-Action languages have been proposed in the literature, differing e.g. on the expressivity of the language and on its operational behavior, existing Event-Condition-Action languages do not generally support the action component to be formulated as a transaction. In this paper, sustaining that it is important to execute transactions in reactive languages, we propose an Event-Condition-Transaction language, based on an extension of Transaction Logic. This extension, called Transaction Logic with Events ($\mathcal{TR}^{ev}$), combines reasoning about the execution of transactions with the ability to detect complex events. An important characteristic of $\mathcal{TR}^{ev}$ is that it takes a choice function as a parameter of the theory, leaving open the behavioral decisions of the logic, and thereby allowing it to be suitable for a wide-spectrum of application scenarios like Semantic Web, multi-agent systems, databases, etc. We start by showing how $\mathcal{TR}^{ev}$ can be used as an Event-Condition-Action language where actions are considered as transactions, and how to differently instantiate this choice function to achieve different operational behaviors. Then, based on a particular operational instantiation of the logic, we present a procedure that is sound and complete w.r.t. the semantics and that is able to execute $\mathcal{TR}^{ev}$ programs.

1 Introduction and Motivation

Most of today's applications are highly dynamic, as they produce, or depend on data that is rapidly changing and evolving over time. This is especially the case of Web-based applications, which normally have to deal with large volumes of data, with new information being added and updated constantly. In fact, with the amount of online data increasing exponentially every year, it is estimated that the annual IP traffic will reach the zettabyte threshold in 2015[1].

This sheer amount of information forced us to dramatically change the way we store, access, and reason with data on the web, and led to the development of several research areas. Among them, the Semantic Web, which started about 15 years ago, aims to enrich the Web with machine-understandable information by promoting a collaborative movement where users publish data in one of the standard formats, RDF or OWL,

[⋆] This work was supported by project ERRO (PTDC/EIA-CCO/121823/2010).

[1] `http://blogs.cisco.com/news/the-dawn-of-the-zettabyte-era-infographic`

designed to give a precise semantic meaning to web data. On the other hand, research areas like Event Processing (EP) and Stream Reasoning deal with the problem of handling and processing a high volume of events (also called a *stream*), and reason with them to detect meaningful event patterns (also known as complex events). Although EP started in the 1990s within the database community, today we can find a number of EP solutions based on Semantic Web technologies like RDF, SPARQL and OWL [3,20,17]. These EP solutions deal with the challenge of detecting event patterns on a stream of atomic events. But detecting these patterns is only part of what one has to do to deal with the dynamics of data. In fact, detecting event patterns is only meaningful if we can act upon the knowledge of their occurrence.

Event-Condition-Action (ECA) languages solve this by explicitly defining what should be the reaction of a system, when a given event pattern is detected. For that, ECA-rules have the standard form: *on* **event** *if* **condition** *do* **action**, where whenever an event is known to be true, the condition is checked to hold in the current state and, if that is the case, the action is executed. Initially introduced to support reactivity in database systems, numerous ECA languages have been proposed e.g. in the context of the Semantic Web [6,10,19], multi-agent systems [18,12,15], conflict resolution [11]. Moreover, although all these languages share the same reactive paradigm, they often vary on the expressivity of the language and the connectives available, but also on their operational behavior, namely on the event consumption details, and on the response policy and scheduling. While ECA languages started in the database context, and many solutions exist supporting rich languages for defining complex actions, most ECA languages do not allow the action component to be defined as a transaction, or when they do, they either lack from a declarative semantics (e.g. [19]), or can only be applied in a database context since they only detect atomic events defined as primitive insertions/deletes on the database (e.g. [24,16]).

Originally proposed to make database operations reliable, transactions ensure several properties, like consistency or atomicity, over the execution of a set of actions. We sustain that in several situations ECA languages are indeed required to execute transactions in response to events, where either the whole transaction is executed or, if anything fails meanwhile, nothing is changed in the knowledge base (KB). As an application scenario, consider the case where the local government wants to control the city's air quality by restricting vehicles more than 15 years old to enter certain city areas. For that, the city needs to identify the plates of the cars crossing these areas, and issue fines for the unauthorized vehicles. In this case, whenever a vehicle enters the controled area (the event), we need to check if that vehicle has access to the area (the condition), and if not, issue a fine and notify the driver for the infraction. Clearly, some transactional properties regarding these actions must be ensured, as it can never be the case that a fine is issued and the driver is not notified, or vice-versa.

Transaction Logic ($\mathcal{TR}$) is a general logic proposed in [8] to deal with transactions, by allowing us to reason about their behaviors but also to *execute* them. For that, $\mathcal{TR}$ provides a general model theory that is parametric on a pair of oracles defining the semantics of states and updates of the KB (e.g. relational databases, action languages, description logics). With it, one can reason about the sequence of states (denoted as paths) where a transaction is executed, *independently* of the semantics of states and primitive actions of the KB. Additionally, $\mathcal{TR}$ also provides a proof-theory to *execute* a subclass of $\mathcal{TR}$ programs that can be formulated as Horn-like clauses. However, $\mathcal{TR}$ cannot simultaneously deal with complex events and transactions, and for that we have previously proposed $\mathcal{TR}^{ev}$ in [14]. $\mathcal{TR}^{ev}$ is an extension of $\mathcal{TR}$ that, like the original $\mathcal{TR}$, can reason about the execution of transactions, but also allows for the definition of complex events by combining atomic (or other complex) events. In $\mathcal{TR}^{ev}$, atomic events can either be external events, which are signalled to the KB, or the execution of primitive updates in the KB (similarly e.g. to the events "on insert" in databases). Moreover, as in active databases, transactions in $\mathcal{TR}^{ev}$ are *constrained* by the events that occur during their execution, as a transaction can only successfully commit when all events triggered during its execution are addressed. Importantly, $\mathcal{TR}^{ev}$ is parameterized with a pair of oracles as in the original $\mathcal{TR}$, but also with a $choice$ function abstracting the semantics of a reactive language from its response policies decisions. $\mathcal{TR}^{ev}$ is a first step to achieve a reactive language that combines the detection of events with the execution of transactions. However, it leaves open how the ECA paradigm can be encoded and achieved in $\mathcal{TR}^{ev}$, but also, how can one execute such reactive ECA rules.

In this paper we show how one can indeed use $\mathcal{TR}^{ev}$ as the basis for an Event-Condition-Transaction language, illustrating how different response policies can be achieved by correctly instantiating the choice functions. Then, to make the logic useful in practice, we provide a proof procedure sound and complete with the semantics, for a Horn-like subset of the logic, and for a particular choice function instantiation.

2 Background: $\mathcal{TR}^{ev}$

Transaction Logic [8], $\mathcal{TR}$, is a logic to execute and reason about general changes in KB, when these changes need to follow a transactional behavior. In a nutshell[2], The $\mathcal{TR}$ syntax extends that of first order logic with the operators $\otimes$ and $\Diamond$, where $\phi \otimes \psi$ denotes the action composed by an execution of ϕ followed by an execution of ψ, and $\Diamond\phi$ denotes the hypothetical execution of ϕ, i.e. a test to see whether ϕ can be executed but leaving the current state unchanged. Then, $\phi \wedge \psi$ denotes the simultaneous execution of ϕ and ψ; $\phi \vee \psi$ the execution of ϕ or ψ; and $\neg\phi$ an execution where ϕ is not executed.

In $\mathcal{TR}$ all formulas are read as transactions which are evaluated over sequences of KB states known as *paths*, and satisfaction of formulas means execution. I.e., a formula (or transaction) ϕ is true over a path π iff the transaction successfully executes over that sequence of states. $\mathcal{TR}$ makes no particular assumption on the representation of states, or on how states change. For that, $\mathcal{TR}$ requires the existence of two oracles: the data oracle $\mathcal{O}^d$ abstracting the representation of KB states and used to query them, and the transition oracle $\mathcal{O}^t$ abstracting the way states change. For example, a KB made of a relational database [8] can be modeled by having states represented as sets of ground atomic formulas, where the data oracle simply returns all these formulas, i.e., $\mathcal{O}^d(D) = D$. Moreover, for each predicate `p` in the KB, the transition oracle defines `p.ins` and `p.del`, representing the insertion and deletion of `p`, respectively (where

[2] For lack of space, and since $\mathcal{TR}^{ev}$ is an extension of $\mathcal{TR}$ (cf. [14]) we do not make a thorough overview of $\mathcal{TR}$ here. For complete details see e.g. [14,8]

$\texttt{p.ins} \in \mathcal{O}^t(D_1, D_2)$ iff $D_2 = D_1 \cup \{\texttt{p}\}$ and, $\texttt{p.del} \in \mathcal{O}^t(D_1, D_2)$ iff $D_2 = D_1 \backslash \{\texttt{p}\}$). We will use this relational database oracle definition in our example (Ex. 1).

The logic provides the concept of a *model* of a $\mathcal{TR}$ theory, which allows one to prove properties of transactions that hold for every possible path of execution; and the notion of executional entailment, in which a transaction ϕ is entailed by a theory given an initial state D_0 and a program P (denoted as $P, D_0 - \models \phi$), if there is a path $D_0, D_1 \ldots, D_n$ starting in D_0 on which the transaction, as a whole, succeeds. Given a transaction and an initial state, the executional entailment provides a means to determine what should be the evolution of states of the KB, to succeed the transaction in an atomic way. Non-deterministic transactions are possible, and in this case several successful paths exist. Finally, a proof procedure and corresponding implementation exist for a special class of $\mathcal{TR}$ theories, called serial-Horn programs, which extend definite logic programs with serial conjunction.

$\mathcal{TR}^{ev}$ extends $\mathcal{TR}$ in that, besides dealing with the execution of transaction, it is also able to raise and detect complex events. For that, $\mathcal{TR}^{ev}$ separates the evaluation of events from the evaluation of transactions. This is reflected in its syntax, and on the two different satisfaction relations – the event satisfaction $\models_{ev}$ and the transaction satisfaction $\models$. The alphabet of $\mathcal{TR}^{ev}$ contains an infinite number of constants $\mathcal{C}$, function symbols $\mathcal{F}$, variables $\mathcal{V}$ and predicate symbols $\mathcal{P}$. Furthermore, predicates in $\mathcal{TR}^{ev}$ are partitioned into transaction names ($\mathcal{P}_t$), event names ($\mathcal{P}_e$), and oracle primitives ($\mathcal{P}_{\mathcal{O}}$). Finally, formulas are also partitioned into transaction formulas and event formulas.

Event formulas are formulas meant to be *detected* and are either an event occurrence, or an expression defined inductively as $\neg\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, or $\phi \otimes \psi$, where ϕ and ψ are event formulas. We further assume $\phi;\psi$, which is syntactic sugar for $\phi \otimes \texttt{path} \otimes \psi$ (where $\texttt{path}$ is just any tautology, cf. [9]), with the meaning: "ϕ and then ψ, but where arbitrary events may be true between ϕ and ψ". An *event occurrence* is of the form $\mathbf{o}(\varphi)$ s.t. $\varphi \in \mathcal{P}_e$ or $\varphi \in \mathcal{P}_{\mathcal{O}}$ (the latter are events signaling changes in the KB, needed to allow reactive rules similar to e.g. "on insert" triggers in databases).

Transaction formulas are formulas that can be *executed*, and are either a transaction atom, or an expression defined inductively as $\neg\phi$, $\Diamond\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, or $\phi \otimes \psi$. A *transaction atom* is either a transaction name (in $\mathcal{P}_t$), an oracle defined primitive (in $\mathcal{P}_{\mathcal{O}}$), the response to an event (written $\mathbf{r}(\varphi)$ where $\varphi \in \mathcal{P}_{\mathcal{O}} \cup \mathcal{P}_e$), or an event name (in $\mathcal{P}_e$). The latter corresponds to the (trans)action of *explicitly* triggering/raising an event directly in a transaction. Finally, rules have the form $\varphi \leftarrow \psi$ and can be transaction or (complex) event rules. In a transaction rule φ is a transaction atom and ψ a transaction formula; in an event rule φ is an event occurrence and ψ is an event formula. A *program* is a set of transaction and event rules.

Example 1. Consider the example from the introduction, and a relational KB with information about license plates registration, drivers addresses, and fines. The transaction of processing an unauthorized access of a vehicle $\texttt{V}$, at a given time $\texttt{T}$, and city area $\texttt{A}$ (written $\texttt{processUnAccess}(\texttt{V}, \texttt{T}, \texttt{A})$) can be defined in $\mathcal{TR}^{ev}$ (and in $\mathcal{TR}$) as:[3]

[3] Without loss of generality (cf. [9]), we consider Herbrand instantiations of the language.

$\mathtt{processUnAccess(V,T,A)} \leftarrow$
$\quad \mathtt{fineCost(A,Cost)} \otimes \mathtt{unAccess(V,T,A,Cost).ins} \otimes \mathtt{notifyFine(V,T,A,Cost)}$
$\mathtt{notifyFine(V,T,A,Cost)} \leftarrow$
$\quad \mathtt{isDriver(D,V)} \otimes \mathtt{hasAddress(D,Addr)} \otimes \mathtt{sendLetter(D,A,V,T,A,Cost)}$

One can also define complex events. E.g., a vehicle is said to enter the city from a given entrance $\mathtt{E_1}$, if it is detected by the two sensors of that entrance, first by sensor 1 and then by sensor 2, with a time difference less than 0.5s:

$$\mathbf{o}(\mathtt{enterCityE_1(V,T_2)}) \leftarrow$$
$$((\mathbf{o}(\mathtt{sensor1E_1(V,T_1).ins}); \mathbf{o}(\mathtt{sensor2E_2(V,T_2).ins})) \wedge \mathtt{T_2} - \mathtt{T_1} < 0.5)$$

For simplicity, in this example we assume that sensors' data is directly inserted in the application's database, where the semantics of $\_.ins$ is as defined by the relational oracle above. However, other paradigms can be used to represent and reason about sensor's data. Namely, and given some recent works in the field of Sensor Networks [22,21], we could have alternatively assumed a Semantic Sensor Network to publish data in RDF, and have defined the oracle $\mathcal{O}^d$ to query this data using SPARQL. This would allow us to use $\mathcal{O}^d$ to integrate RDF data with the government's database.

Central to the theory of $\mathcal{TR}^{ev}$ is the correspondence between $\mathbf{o}(\varphi)$ and $\mathbf{r}(\varphi)$. As a transactional system, the occurrence of an event constrains the satisfaction path of the transaction where the event occurs, and a transaction can only "commit" if all the occurring events are answered. More precisely, a transaction is only satisfied on a path, if all the events occurring over that path are properly responded to. This behavior is achieved by evaluating event occurrences and transactions differently, and by imposing $\mathbf{r}(\varphi)$ to be true in the paths where $\mathbf{o}(\varphi)$ holds. For dealing with cases where more than one occurrence holds simultaneously, $\mathcal{TR}^{ev}$ takes as parameter, besides $\mathcal{TR}$'s data and transition oracles, a *choice* function selecting what event should be responded at a given time, in case of conflict. This function abstracts the operational decisions from the logic, and allows $\mathcal{TR}^{ev}$ to be useful for a wide spectrum of applications.

As in $\mathcal{TR}$, formulas of $\mathcal{TR}^{ev}$ are evaluated over paths (sequence of states), and the theory allows us to reason about *how* does the KB evolve in a transactional way, based on an initial KB state. In addition, a path in $\mathcal{TR}^{ev}$ is of the form $D_0 \xrightarrow{O_1} \dots \xrightarrow{O_n} D_n$, and where each O_i is a primitive event occurrence that holds in the state transition D_{i-1}, D_i. As a reactive system, $\mathcal{TR}^{ev}$ receives a series (or a stream) of external events which may cause the execution of transactions in response. This is defined as $P, D_0 - \models e_1 \otimes \dots \otimes e_k$, where D_0 is the initial KB state and $e_1 \otimes \dots \otimes e_k$ is the sequence of external events that arrived. Then, we want to know what is the path $D_0 \xrightarrow{O_1} \dots \xrightarrow{O_n} D_n$ encoding a KB evolution that responds to $e_1 \otimes \dots \otimes e_k$.

As usual, satisfaction of formulas is based on interpretations which define what atoms are true over what paths, by mapping every possible path to a set of atoms. If a transaction (resp. event) atom ϕ belongs to $M(\pi)$ then ϕ is said to execute (resp. occur) over path π given interpretation M. However, only the mappings compliant with the specified oracles are considered as interpretations:

Definition 1 (Interpretation). *An interpretation is a mapping M assigning a set of atoms (or $\top$[4]) to paths, with the restrictions (where D_is are states, and φ an atom):*

[4] For not having to consider partial mappings, besides formulas, interpretations can also return the special symbol $\top$. The interested reader is referred to [8] for details.

1. $\varphi \in M(\langle D \rangle)$ if $\varphi \in \mathcal{O}^d(D)$
2. $\{\varphi, \mathbf{o}(\varphi)\} \subseteq M(\langle D_1 \overset{\mathbf{o}(\varphi)}{\rightarrow} D_2 \rangle)$ if $\varphi \in \mathcal{O}^t(D_1, D_2)$
3. $\mathbf{o}(e) \in M(\langle D \overset{\mathbf{o}(e)}{\rightarrow} D \rangle)$

Satisfaction of formulas requires the definition of operations on paths. E.g., $\phi \otimes \psi$ is true on a path if ϕ is true up to some point in the path, and ψ is true from that onwards.

Definition 2 (Path Splits, Subpaths and Prefixes). *Let π be a k-path, i.e. a path of length k of the form $\langle D_1 \overset{O_1}{\rightarrow} \dots \overset{O_{k-1}}{\rightarrow} D_k \rangle$. A* split *of π is any pair of subpaths, π_1 and π_2, s.t. $\pi_1 = \langle D_1 \overset{O_1}{\rightarrow} \dots \overset{O_{i-1}}{\rightarrow} D_i \rangle$ and $\pi_2 = \langle D_i \overset{O_i}{\rightarrow} \dots \overset{O_{k-1}}{\rightarrow} D_k \rangle$ for some i $(1 \leq i \leq k)$. In this case, we write $\pi = \pi_1 \circ \pi_2$.*
A subpath π' of π is any subset of states of π where both the order of the states and their annotations is preserved. A prefix π_1 of π is any subpath of π sharing the initial state.

Satisfaction of complex formulas is different for event formulas and transaction formulas. While the satisfaction of event formulas concerns the *detection* of an event, the satisfaction of transaction formulas concerns the *execution* of actions in a transactional way. As such, when compared to the original $\mathcal{TR}$, transactions in $\mathcal{TR}^{ev}$ are further required to execute *all* the responses of the events occurring in the original execution path of that transaction. In other words, a transaction φ is satisfied over a path π, if φ is executed on a prefix π_1 of π (where $\pi = \pi_1 \circ \pi_2$), and all events occurring over π_1 are *responded* over π_2. This requires a non-monotonic behavior of the satisfaction relation of transaction formulas, making them dependent on the satisfaction of events.

Definition 3 (Satisfaction of Event Formulas). *Let M be an interpretation, π a path and ϕ a formula. If $M(\pi) = \top$ then $M, \pi \models_{ev} \phi$; else:*
1. ***Base Case:** $M, \pi \models_{ev} \phi$ iff $\phi \in M(\pi)$ for every event occurrence ϕ*
2. ***Negation:** $M, \pi \models_{ev} \neg\phi$ iff it is not the case that $M, \pi \models_{ev} \phi$*
3. ***Disjunction:** $M, \pi \models_{ev} \phi \vee \psi$ iff $M, \pi \models_{ev} \phi$ or $M, \pi \models_{ev} \psi$.*
4. ***Serial Conjunction:** $M, \pi \models_{ev} \phi \otimes \psi$ iff there is a split $\pi_1 \circ \pi_2$ of π s.t. $M, \pi_1 \models_{ev} \phi$ and $M, \pi_2 \models_{ev} \psi$*
5. ***Executional Possibility:** $M, \pi \models_{ev} \Diamond\phi$ iff π is a 1-path of the form $\langle D \rangle$ for some state D and $M, \pi' \models_{ev} \phi$ for some path π' that begins at D.*

Definition 4 (Satisfaction of Transaction Formulas). *Let M be an interpretation, π a path, ϕ transaction formula. If $M(\pi) = \top$ then $M, \pi \models \phi$; else:*
1. ***Base Case:** $M, \pi \models p$ iff there is a prefix π' of π s.t. $p \in M(\pi')$ and π is an expansion of path π' w.r.t. M, for every transaction atom p s.t. $p \notin \mathcal{P}_e$.*
2. ***Event Case:** $M, \pi \models e$ iff $e \in \mathcal{P}_e$ and there is a prefix π' of π s.t. $M, \pi' \models_{ev} \mathbf{o}(e)$ and π is an expansion of path π' w.r.t. M.*
3. ***Negation:** $M, \pi \models \neg\phi$ iff it is not the case that $M, \pi \models \phi$*
4. ***Disjunction:** $M, \pi \models \phi \vee \psi$ iff $M, \pi \models \phi$ or $M, \pi \models \psi$.*
5. ***Serial Conjunction:** $M, \pi \models \phi \otimes \psi$ iff there is a prefix π' of π and a split $\pi_1 \circ \pi_2$ of π' s.t. $M, \pi_1 \models \phi$ and $M, \pi_2 \models \psi$ and π is an expansion of path π' w.r.t. M.*
6. ***Executional Possibility:** $M, \pi \models \Diamond\phi$ iff π is a 1-path of the form $\langle D \rangle$ for some state D and $M, \pi' \models \phi$ for some path π' that begins at D.*

The latter definition depends on the notion of expansion of a path. An *expansion* of a path π_1 w.r.t. to an interpretation M is an operation that returns a new path π_2 where all events occurring over π_1 (and also over π_2) are completely answered. Formalizing this expansion requires the prior definition of what it means to answer an event:

Definition 5 (Path response). *For a path π_1 and an interpretation M we say that π is a response of π_1 iff $choice(M, \pi_1) = e$ and we can split π into $\pi_1 \circ \pi_2$ s.t. $M, \pi_2 \models \mathbf{r}(e)$.*

The choice of what unanswered event should be picked at each moment is given by an event function $choice$. This function has the role to decide what events are unanswered over a path π w.r.t. an interpretation M and, based on a given criteria, select what event among them should be responded to first. Just like $\mathcal{TR}$ is parametric to a pair of oracles ($\mathcal{O}^d$ and $\mathcal{O}^t$), $\mathcal{TR}^{ev}$ takes the $choice$ function as an additional parameter. For now, we leave the definition and role of this function open until Section 3. Nevertheless, and importantly, if all events that occur on a path π are answered on π w.r.t. M, then $choice(M, \pi) = \epsilon$. We can now define what is an expansion of a path.

Definition 6 (Expansion of a path). *A path π is completely answered w.r.t. to an interpretation M iff $choice(M, \pi) = \epsilon$. π is an* expansion *of the path π_1 w.r.t. M iff:*
- *π is completely answered w.r.t. M, and*
- *either $\pi = \pi_1$; or there is a sequence of paths $\pi_1, \ldots, \pi$, starting in π_1 and ending in π, s.t. each π_i in the sequence is a response of π_{i-1} w.r.t. M.*

The latter definition specifies how to expand a path π_1 in order to obtain another path π where all events satisfied over subpaths of π are also answered within π. This must perforce have some procedural nature: it must start by detecting which are the unanswered events; pick one of them, according to some criteria given by a $choice$ function, that for now is seen as a parameter; then expand the path with the response of the chosen event. Each path π_i of the sequence $\pi_1, \pi_2, \ldots, \pi$ is a prefix of the path π_{i+1}, and where at least one of the events unanswered on π_i is now answered on π'; otherwise, if all events occurring over π_i are answered, then $\pi_i = \pi$, and the expansion is complete. We can now define the notion of *model* of formulas and programs.

Definition 7 (Models and Minimal Models). *An interpretation M is a* model *of a transaction (resp. event) formula ϕ iff for every path π, $M, \pi \models \phi$ (resp. $M, \pi \models_{ev} \phi$). M is a model of a program P (denoted $M \models P$) iff it is a model of every rule in P. We say that a model is minimal if it is a $\subseteq$-minimal model.*

This notion of models can be used to reason about properties of transaction and event formulas that hold for *every* possible path of execution. However, to know if a formula succeeds on a particular path, we need only to consider the event occurrences *supported* by that path, either because they appear as occurrences in the transition of states, or because they are a necessary consequence of the program's rules given that path. Because of this, executional entailment in $\mathcal{TR}^{ev}$ is defined w.r.t. minimal models.

Definition 8 ($\mathcal{TR}^{ev}$ Executional Entailment). *Let P be a program, ϕ a transaction formula and $D_1 \xrightarrow{O_0} \ldots \xrightarrow{O_n} D_n$ a path. Then $P, (D_1 \xrightarrow{O_0} \ldots \xrightarrow{O_n} D_n) \models \phi$ ($\star$) iff for every minimal model M of P, $M, \langle D_1 \xrightarrow{O_0} \ldots \xrightarrow{O_n} D_n \rangle \models \phi$. $P, D_1 - \models \phi$ is said to be true, if there is a path $D_1 \xrightarrow{O_0} \ldots \xrightarrow{O_n} D_n$ that makes ($\star$) true.*

3 $\mathcal{TR}^{ev}$ as an Event-Condition-Transaction and the $choice$ function

In the previous section, we provided the logical background for a reactive language that can express and reason about both transactions and complex events. Next we show how

$\mathcal{TR}^{ev}$ can indeed be used as an Event-Condition-Transaction language, and how several ECA behaviors can be embedded in the semantics by providing the right translation into $\mathcal{TR}^{ev}$, and the right instantiations of the $choice$ function.

As mentioned, an ECA language follows the basic paradigm **on** *event* **if** *condition* **do** *action*, defining that, whenever the *event* is learned to occur, the *condition* is tested, and, if it holds, the *action* is executed. As such, an ECA rule is said to be active whenever the event holds and, to satisfy it, either the condition does not hold before the execution of the action, or the action is issued for execution. Moreover, in an Event-Condition-Transaction language, this action needs not only to be executed, but to be executed as a transaction. This means that we need to guarantee that either the whole of the transaction is executed or, if anything fails meanwhile, the KB is left unchanged.

Since $\mathcal{TR}^{ev}$ forces every formula $\mathbf{r}(ev)$ to be true whenever $\mathbf{o}(ev)$ is learnt to be true, this behavior can be simply encoded as:

$$\begin{array}{l} \mathbf{r}(ev) \leftarrow \Diamond cond \otimes action \\ \mathbf{r}(ev) \leftarrow \neg\Diamond cond \end{array} \tag{1}$$

where $\Diamond cond$ is a test (and which necessarily does not cause changes in the KB) to determine if the condition $cond$ holds, and if this is the case, the $action$ is executed in the KB. Moreover, if one wants to define the event ev as a complex event, then one should add a rule stating the event pattern definition: $\mathbf{o}(ev) \leftarrow body$.

Example 2. Recall the example in the introduction and the transaction defined in Ex. 1 for processing unauthorized accesses. Then event-condition-transaction rule triggering that transaction can be written as:

$$\begin{array}{ll} \mathbf{o}(\mathtt{enterCity(V,T,e1)}) & \leftarrow \mathbf{o}(\mathtt{enterCityE_1(V,T)}) \\ \mathbf{r}(\mathtt{enterCity(V,\_,A)}) & \leftarrow \mathtt{authorized(V,A)} \\ \mathbf{r}(\mathtt{enterCity(V,T,A)}) & \leftarrow \mathtt{unauthorized(V,A)} \otimes \mathtt{processUnAccess(V,T,A)} \end{array}$$

where, since `authorized` and `unauthorized` are queries to the KB (i.e., they cause no change in the database), we can drop the $\Diamond$ constructor.

Moreover, the definition of an ECA language requires the specification of an operational behaviors, which in turn, involves two majors decisions: 1) in which order should events be responded when more than one event is detected simultaneously; and 2) how should an event be responded to. In order to make $\mathcal{TR}^{ev}$ as flexible as possible, its model theory was abstracted from these decisions, encapsulating them in a $choice$ function. This function is required as a parameter of the theory (similarly to the oracles $\mathcal{O}^t$ and $\mathcal{O}^d$) and precisely defines what is the next event that still needs to be responded.

Definition 9 **($choice$ function).** *Let M be an interpretation and π be a path. Then:* $choice(M,\pi) = \mathit{firstUnans}(M,\pi,order(M,\pi))$.

Matching these two major decisions, our definition of the $choice$ function is partitioned in two functions: the $order$ function specifying the sorting criteria of events, and a $\mathit{firstUnans}$ function which checks what events are unanswered and returns the first one based on the previous order. The former decision defines the handling order of events, i.e. given a set of occurring events, what should be responded first. This ordering can be defined e.g. based on when they have occurred (temporal order), on a priority list, or any other criteria. This decision defines the response policy of an ECA-language, i.e. how should an event be responded. We start by illustrating the $order$ function:

Example 3 (Ordering-Functions). Let $\langle e_1, \ldots, e_n \rangle$ be a sequence of events, π a path, and M an interpretation.

Temporal Ending Order $order(M, \pi) = \langle e_1, \ldots, e_n \rangle$ iff $\forall e_i$ s.t. $1 \leq i \leq n$ then $\exists \pi_i$ subpath of π where $M, \pi_i \models_{ev} \mathbf{o}(e_i)$ and $\forall e_j$ s.t. $i < j$ and then e_j occurs after e_i w.r.t. π.

Temporal Starting Order $order(M, \pi) = \langle e_1, \ldots, e_n \rangle$ iff $\forall e_i$ s.t. $1 \leq i \leq n$ then $\exists \pi_i$ subpath of π where $M, \pi_i \models_{ev} \mathbf{o}(e_i)$ and $\forall e_j$ s.t. $i < j$ then e_j starts before e_i w.r.t. π.

Priority List Order Let L be a priority list where events are linked with numbers starting in 1, where 1 is the event with higher priority. $order_L(M, \pi) = \langle e_1, \ldots, e_n \rangle$ iff $\forall e_i$ $\exists \pi_i$ subpath of π s.t. $M, \pi_i \models_{ev} \mathbf{o}(e_i)$ and $\forall e_j$ where $1 \leq i < j \leq n$, π_j is subpath of π and $M, \pi_j \models_{ev} \mathbf{o}(e_j)$ then $L(e_i) \leq L(e_j)$.

All these examples require the notion of event ordering, which can be defined as:

Definition 10 (Ordering of Events). *Let e_1, e_2 be events, π a path, and M an interpretation. e_2* occurs after *e_1 w.r.t. π and M iff $\exists \pi_1, \pi_2$ subpaths of π s.t. $\pi_1 = \langle D_i \xrightarrow{O_i} \ldots \xrightarrow{O_{j-1}} D_j \rangle$, $\pi_2 = \langle D_n \xrightarrow{O_n} \ldots \xrightarrow{O_{m-1}} D_m \rangle$, $M, \pi_1 \models_{ev} \mathbf{o}(e_1)$, $M, \pi_2 \models_{ev} \mathbf{o}(e_2)$ and $D_j \leq D_m$ w.r.t. the ordering in π. e_1* starts before *e_2 w.r.t. π if $D_i \leq D_n$*

Choosing the appropriate event ordering obviously depends on the application in mind. For instance, in system monitoring applications there may exist alarms with higher priority over others that need to be addressed immediately, while in a webstore context it may be more important to treat events in the temporal orders in which they are detected.

It remains to be defined the response policy, i.e., what requisites should be imposed w.r.t. the response executions. This is done by appropriately instantiating the $firstUnans$ function. Here we illustrate two alternative instantiations. In the first, encoded in *Relaxed Response*, the function simply retrieves the first event e such that its response is not satisfied in a path after the occurrence. With it, if an event occurs more than once, it is sufficient to respond to it once. Alternative definitions are possible, e.g. where responses are issued explicitly for each event. This is encoded in the *Explicit Response*, where we verify whether $\mathbf{r}(e_i)$ is satisfied but always w.r.t. its correct order.

Example 4 (Answering Choices). Let π be a path, M an interpretation, and $\langle e_1, \ldots, e_n \rangle$ a sequence of events.

Relaxed Response $firstUnans(M, \pi, \langle e_1, \ldots, e_n \rangle) = e_i$ if e_i is the first event in $\langle e_1, \ldots, e_n \rangle$ s.t. $\exists \pi'$ subpath of π where $M, \pi' \models_{ev} \mathbf{o}(e)$ and $\neg \exists \pi''$ s.t. π'' is also a subpath of π, π'' is after π' and $M, \pi'' \models \mathbf{r}(e)$.

Explicit Response $firstUnans(M, \pi, \langle e_1, \ldots, e_n \rangle) = e_i$ if e_i is the first event in $\langle e_1, \ldots, e_n \rangle$ s.t. $\exists \pi'$ subpath of π where $M, \pi' \models_{ev} \mathbf{o}(e)$ and if $\exists \pi''$ subpath of π that is after π' where $M, \pi'' \models \mathbf{r}(e_i)$ then $\exists \pi_1, \pi_2$ subpaths of π and π_2 is after π_1 where $M, \pi_1 \models_{ev} \mathbf{o}(e_j)$, $M, \pi_2 \models \mathbf{r}(e_j)$, $j < i$ and π'' starts before the ending of π_2

4 Procedure for serial-$\mathcal{TR}^{ev}$

To make $\mathcal{TR}^{ev}$ useful in practice, we propose a proof procedure for executing Event-Condition-Transaction rules that is sound and complete w.r.t. $\mathcal{TR}^{ev}$'s executional entailment. Given a $\mathcal{TR}^{ev}$ program and a KB state, the procedure takes a *stream* of events that are known to occur, and finds a KB evolution where all the (possibly complex) events resulting from the direct and indirect occurrence of this stream, are responded to as a transaction. More precisely, the procedure finds solutions for statements of the form $P, D_0 - \models e_1 \otimes \ldots \otimes e_k$, by finding paths (which encode a KB evolution) where the formula $P, D_0 \overset{O_0}{\hookrightarrow} \ldots \overset{O_{n-1}}{\hookrightarrow} D_n \models e_1 \otimes \ldots \otimes e_k$ holds. Regarding the (procedural) choices discussed in the previous section, this procedures fixes an event ordering based on a priority list, and assumes Explicit Response (cf. Ex. 4).

The procedure is partitioned into two major parts: the execution of actions (based on a top-down computation), and the detection of event patterns (based on a bottom-up computation). The detection of event patterns is inspired by the ETALIS detection algorithm [4], where event rules are first pre-processed using a *binarization of events*. In other words, event rules are first transformed so that all their bodies have at most two atoms. If we have a body with more than 2 atoms, e.g. $\mathbf{o}(e) \leftarrow \mathbf{o}(a)\ \mathtt{OP}\ \mathbf{o}(b)\ \mathtt{OP}\ \mathbf{o}(c)$ and if we assume a left-associative operator, the binarization of this rule leads to replacing it by $ie_1 \leftarrow \mathbf{o}(a)\ \mathtt{OP}\ \mathbf{o}(b)$ and $\mathbf{o}(e) \leftarrow ie_1\ \mathtt{OP}\ \mathbf{o}(c)$, and where $\mathtt{OP}$ can be any binary $\mathcal{TR}^{ev}$ operator. This is done without loss of generality since:

Proposition 1 (Program equivalence). *Let P_1 and P_2 be programs, π be a path and ϕ a formula defined in both P_1 and P_2 alphabet. We say that $P_1 \equiv P_2$ if:*

$$M_1, \pi \models \phi \quad \textit{iff} \quad M_2, \pi \models \phi$$

where M_1 (resp. M_2) is the set of minimal models of P_1 (resp. P_2).

Let P be a program with rule: $e \leftarrow e_1\ \mathtt{OP}\ e_2\ \mathtt{OP}\ e_3$ for any events e_1-e_3 and operator $\mathtt{OP}$, and let P' be obtained from P by removing that rule and adding $ie_1 \leftarrow e_1\ \mathtt{OP}\ e_2$ and $e \leftarrow ie_1\ \mathtt{OP}\ e_3$. Then $P \equiv P'$.

Besides binarization, and as it is usual in EP systems, we restrict the use of negation in the procedure. The problem with negation is that it is hard to detect the non-presence of an event pattern in an unbounded interval. Due to this, EP systems like [1,4] always define negation bounded to two other events as in $\mathbf{not}(e_3)[e_1, e_2]$. This holds if e_3 does not occur in the interval defined by the occurrence of e_1 and e_2. This is captured in $\mathcal{TR}^{ev}$ by: $e_1 \otimes \neg(\mathtt{path} \otimes e_3 \otimes \mathtt{path}) \otimes e_2$, and we restrict negation to this pattern.

The execution of actions is based on $\mathcal{TR}$ proof-theory [8,13] which is only defined for a subclass of programs where transactions can be expressed as serial-Horn goals. As such, the execution of transactions in this procedure is defined for this fragment, which resembles Horn-clauses of logic programming. A serial goal is a transaction formula of the form $a_1 \otimes a_2 \otimes \ldots \otimes a_n$ where each a_i is an atom and $n \geq 0$. When $n = 0$ we write $()$ which denotes the empty goal. Finally, a serial-Horn rule has the form $b \leftarrow a_1 \otimes \ldots \otimes a_n$, where the body $a_1 \otimes \ldots \otimes a_n$ is a serial goal and b is an atom.

The procedure starts with a program P, an initial state D, a serial goal $e_1 \otimes \ldots \otimes e_k$ and iteratively manipulates *resolvents*. At each step, the procedure non-deterministically applies a series of rules of Def. 11 to the current resolvent until it either reaches the empty goal and succeeds, or no more rules are applicable and the derivation fails. Moreover, if the procedure succeeds, it also returns a path in which the goal succeeds. To cater for this last requirement, resolvents contain the path obtained so far. A resolvent is of the form $\pi, ESet \Vdash^{id}_{P'} \phi$, where ϕ is the current transaction goal to be executed, π

is the path obtained by the procedure so far, $ESet$ is the set of events that were previously triggered and still need to be addressed, and id is an auxiliar state count identifier (whose usage is made clear below). Finally, P' is the current program, containing all the rules from the original program P plus temporary event rules to help deal with the detection of event patterns. A successful derivation for $P, D\text{–} \models e_1 \otimes \ldots \otimes e_k$ starts with the resolvent $\langle D \rangle, \emptyset \Vdash^{1}_{P} e_1 \otimes \ldots \otimes e_k$ and ends in the resolvent $\pi, \emptyset \Vdash^{id}_{P'} ()$. If such a derivation exists, then we write $P, \pi \vdash e_1 \otimes \ldots \otimes e_k$. Most derivation rules have a direct correspondence with $\mathcal{TR}$'s proof theory [8], but now incorporating the notion of path expansion and event detection. Rule 1 replaces a transaction atom L by the $Body$ of a program rule whose head is L; rule 2 deals with a query to the oracle deleting it from the set of goals whenever this query is true in the current state (i.e., the last state of π); rule 3 executes actions according to the transition oracle definition (i.e., if an action A can be executed in the last state D_1 of path π, reaching the state D_2, then we add the path $D_1 \xrightarrow{\mathbf{o}(A)} D_2$ to our current path, but also the path *expansion*, cf. Def. 12, resulting from answering the events that have directly or indirectly become true because of $\mathbf{o}(A)$); finally, rule 4 deals with the triggering of explicit events – if an event e is explicitly triggered and D_1 is the last state of the current path π, then we add the information that $\mathbf{o}(e)$ occurred in state D_1 to our path, and expand it with the KB evolution needed to answer all events that are now true because of e's occurrence.

Definition 11 (Execution). *A derivation for a serial goal ϕ in a program P and state D is a sequence of resolvents starting with $\langle D \rangle, \emptyset \Vdash^{1}_{P} \phi$, and obtained by non-deterministically applying the following rules.*

Let $\pi, ESet \Vdash^{id}_{P_1} L_1 \otimes L_2 \otimes \ldots \otimes L_k$ be a resolvent. The next resolvent is:

1. ***Unfolding of Rule**:*
   $\pi, ESet \Vdash^{id}_{P_1} Body \otimes L_2 \otimes \ldots \otimes L_k$ *if* $L_1 \leftarrow Body \in P$
2. ***Query**:*
   $\pi, ESet \Vdash^{id}_{P_1} L_2 \otimes \ldots \otimes L_k$ *if* $\mathtt{last}(\pi) = D_1$ *and* $\mathcal{O}^d(D_1) \models L_1$
3. ***Update primitive**:*
   $\pi \circ \langle D_1 \xrightarrow{\mathbf{o}(L_1)} D_2 \xrightarrow{O_2} \ldots \xrightarrow{O_{j-1}} D_j \rangle, ESet' \Vdash^{id'}_{P_2} L_2 \otimes \ldots \otimes L_k$ *if:*
   - $\mathtt{last}(\pi) = D_1$ *and* $\mathcal{O}^t(D_1, D_2) \models L_1$
   - *ExpandPath*$(P_1, L_1, \pi, ESet, id) = (P_2, \langle D_2 \xrightarrow{O_2} \ldots \xrightarrow{O_{j-1}} D_j \rangle, ESet', id')$
4. ***Explicit event request**:*
   $\pi \circ \langle D_1 \xrightarrow{\mathbf{o}(L_1)} D_1 \xrightarrow{O_1} \ldots \xrightarrow{O_{j-1}} D_j \rangle, ESet' \Vdash^{id'}_{P_2} L_2 \otimes \ldots \otimes L_k$ *if:*
   - $\mathtt{last}(\pi) = D_1$ *and* $L_1 \in \mathcal{P}_e$
   - *ExpandPath*$(P_1, L_1, \pi, ESet, id) = (P_2, \pi \circ \langle D_1 \xrightarrow{O_1} \ldots \xrightarrow{O_{j-1}} D_j \rangle, ESet', id')$

An execution for ϕ in program P, and state D is successful *if it ends in a resolvent of the form $\pi, \emptyset \Vdash^{id'}_{P'} ()$. In this case we write $P, \pi \vdash \phi$.*

The $ExpandPath$ function is called in the procedure above whenever an event A (either an explicit event or a the execution of a primitive action) occurs, and is responsible for expanding the current path with the responses of the events made true. Moreover, given the event rules in the program, it may be the case that other event occurrences become true, and these other events need also to be responded by this function. The set of events that become true due to the occurrence of A are computed by the $Closure$ function (Def.14), and the process is iterated until there are no more unanswered events:

Definition 12 (Expand Path).
Input: program P, primitive A, path π, event set $ESet$, id
Output: program P', path π', event set $ESet'$, id'

Define $ESet' := ESet \cup \{\mathbf{o}(A)\{id, id+1\}\}$, $\pi' := \pi$, $id' := id + 1$, $P' := P$
while $\texttt{needResponse}(ESet', id, id') \neq \emptyset$ {

1. *Let* $(ESet_{temp}, P_{temp}) = \mathit{Closure}(ESet', P', id')$
2. *Let* $\mathbf{o}(e) = FirstInOrder(\texttt{needResponse}(ESet_{temp}, id, id'))$
3. *Let D be the final state of π', and consider a derivation starting in* $\langle D\rangle, \emptyset \Vdash^{id'}_{P_{temp}} \mathbf{r}(e)$ *and ending in* $\pi_f, ESet_f \Vdash^{id_f}_{P_f} ()$*; if no such derivation exists* ***return*** $failure$
4. *Define* $\pi' := \pi' \circ \pi_f$, $id' := id_f$, $P' := P_f$, $ESet' := (ESet_{temp} \cup ESet_f) \setminus \{\mathbf{o}(e)\}$ }

If $\texttt{needResponse}(ESet', id, id') = \emptyset$ *then the computation is said to be successful.*
In this case ***return*** $(P', \pi', ESet', id')$*; otherwise* ***return*** $failure$.

The $ESet'$ variable in the latter definition contains, at each moment, the set of events that have happened during the execution. Each event in this set is associated to a pair $\{id_i, id_f\}$ specifying the exact interval where the event happened. One can then recast the path of occurrence based on these ids. If $e\{id_i, id_f\} \in ESet$, and $\pi = \langle D_1 \xrightarrow{O_1} \ldots \xrightarrow{O_{k-1}} D_k\rangle$ is the current path, then event e is said to occur in $\pi_{<id_i, id_f>}$, where $\pi_{<id_i, id_f>}$ is the path obtained from π by trimming it from state D_{id_i} to state D_{id_f}: $\langle D_{id_i} \xrightarrow{O_{id_i}} \ldots \xrightarrow{O_{id_f-1}} D_{id_f}\rangle$.

At each iteration $ExpandPath$ collects all the events in $ESet'$ which need to be responded to w.r.t. that iteration. I.e., the events whose occurrence holds on a path starting after the initial state of the function call:

Definition 13 **($\texttt{needResponse}(ESet, id_i, id_j)$).** *Let id_1, id_2, id_i be identifiers and $ESet$ a set of events of the form $e\{id_1, id_2\}$, where id_1 and id_2 define the starting and ending of e, respectively.* $\texttt{needResponse}(ESet, id_i, id_j)$ *is the subset of $ESet$ s.t.* $id_i \leq id_1 \leq id_2 \leq id_j$.

$FirstInOrder$ function simply sorts the events w.r.t. the chosen order function, according to the semantics (cf. Ex. 3), and returns the first event in that order.

Finally, the $Closure$ computation, crucial for this procedure, is responsible for detecting event patterns. Given a pre-processed program where all event-rules are binary, $Closure$ matches events to bodies of event rules, produces new temporary rules containing information about what events still need to occur to trigger an event pattern, and returns a new set of (complex) events that are true. During this procedure, the event component of the program is partitioned between *permanent rules* and *temporary rules*. Permanent rules have the form $\mathbf{o}(e) \leftarrow body$ and come from pre-processing the original program. They can never expire and are always available for activation. Temporary rules have the form $\mathbf{o}(e) \;{}_{\texttt{OP}}\overset{id_1}{\underset{id_2}{\leftarrow}}\; body$ and arise from partially satisfying a permanent rule. They are valid only for some particular iterations (unless as we shall see, if their ids are not open). Then, they are either deleted (in case they are expired without being satisfied) or transformed (if they are partially satisfied). We say these rules are expired if the difference between the current global id and the rule's ending id is greater than 1. Moreover, temporary rules also have the information about the operation $\texttt{OP}$ which defines the constraints needed to satisfy an event pattern.

Definition 14 (Closure).

```
Input: $ESet, P$
Output: $ESet', P'$
repeat {
Define $ESet' := ESet$, $P' := P$
For each $\mathbf{o}(e)\{id_i, id_f\} \in ESet$:
  Base Cases:
    1. If $h \leftarrow \mathbf{o}(e) \in P'$ then:
         $ESet := ESet \cup \{h\{id_i, id_f\}\}$
    2. If $h \otimes\overset{id_1}{\underset{id_2}{\leftarrow}} \mathbf{o}(e) \in P'$ and $id_i = id_2$ then:
         $ESet := ESet \cup \{h\{id_1, id_f\}\}$ and
         $P' := P' \setminus \{h \otimes\overset{id_1}{\underset{id_2}{\leftarrow}} \mathbf{o}(e)\}$
    3. If $h \wedge\overset{id_1}{\underset{id_2}{\leftarrow}} \mathbf{o}(e) \in P'$, $id_1 = id_i, id_2 = id_f$ then:
         $ESet := ESet \cup \{h\{id_i, id_f\}\}$ and
         $P' := P' \setminus \{h \wedge\overset{id_1}{\underset{id_2}{\leftarrow}} \mathbf{o}(e)\}$
    4. If $h \otimes\overset{*}{\underset{*}{\leftarrow}} \mathbf{o}(e) \in P'$ then:
         $ESet := ESet \cup \{h\{{}^*id_i, id_f\}\}$ and
         $P' := P' \setminus \{h \otimes\overset{*}{\underset{*}{\leftarrow}} \mathbf{o}(e)\}$
  Negation Case:
    1. If $h \leftarrow \mathbf{not}(\mathbf{o}(e_3))[\mathbf{o}(e), \mathbf{o}(e_2)] \in P'$ then:
         $P' := P' \cup \{h \otimes\overset{id_i}{\underset{id_f^*}{\leftarrow}} \mathbf{o}(e_2), (h \otimes\overset{id_i}{\underset{id_f^*}{\leftarrow}} \mathbf{o}(e_2)) \neg\overset{id_i}{\underset{id_f^*}{\leftarrow}} \mathbf{o}(e_3)\}$
  Operations Cases:
    1. If $h \leftarrow \mathbf{o}(e) \otimes \mathbf{o}(e_1) \in P'$ then:
         $P' := P' \cup \{h \otimes\overset{id_i}{\underset{id_f}{\leftarrow}} \mathbf{o}(e_1)\}$
    2. If $h \leftarrow \mathbf{o}(e) \wedge \mathbf{o}(e_1) \in P'$ then:
         $P' := P' \cup \{h \wedge\overset{id_i}{\underset{id_f}{\leftarrow}} \mathbf{o}(e_1)\}$
    3. If $h \leftarrow \mathbf{o}(e)\ ;\ \mathbf{o}(e_1) \in P'$ then:
         $P' := P' \cup \{h \otimes\overset{id_i}{\underset{id_f^*}{\leftarrow}} \mathbf{o}(e_1)\}$
  Path Cases:
    1. If $h \leftarrow \mathtt{path} \otimes \mathbf{o}(e_1) \in P'$ then:
         $P' := P' \cup \{h \otimes\overset{*}{\underset{*}{\leftarrow}} \mathbf{o}(e_1)\}$
    2. If $h \otimes\overset{id_1}{\underset{id_2}{\leftarrow}} \mathtt{path} \in P'$ then:
         $ESet := ESet \cup \{h\{id_1, id_2^*\}\}$ and
         $P' := P' \setminus \{h \otimes\overset{id_1}{\underset{id_2}{\leftarrow}} \mathtt{path}\}$
} until $ESet = ESet'$
For each: $rule_j \neg\overset{id_1}{\underset{id_2*}{\leftarrow}} \mathbf{o}(e) \in P'$ and $\mathbf{o}(e)\{id_i, id_f\} \in ESet$ do:
    $P' := P' \setminus \{rule_j \neg\overset{id_1}{\underset{id_2*}{\leftarrow}} \mathbf{o}(e), rule_j\}$
Return $ESet', P'$
```

Note that, the expression `path` is used to expand the interval where an event pattern holds, and is instrumental to define complex event operators as shown in [14]. E.g., $\mathbf{o}(e) \leftarrow \mathbf{o}(a.ins) \otimes \mathtt{path}$ makes $\mathbf{o}(e)$ true in all paths $D_1 \xrightarrow{a.ins} D_2$ where $\mathbf{o}(a.ins)$ holds, but also in all paths obtained by expanding $D_1 \xrightarrow{a.ins} D_2$ to the right. Conversely, $\mathbf{o}(e) \leftarrow \mathtt{path} \otimes \mathbf{o}(a.ins)$ makes $\mathbf{o}(e)$ true in all paths obtained by expanding $D_1 \xrightarrow{a.ins} D_2$ to the left. To cope with this, the procedure deals with the *open ids*: ${}^*id$, id^* and $*$, where $*$ states that the right or left interval of an event pattern occurrence is unknown and ${}^*id$ (resp. id^*) states that the starting (resp. ending) of an event is any point before (resp. after) or equal to id. Since these ids can propagate to several events, we also say that $\forall id.\ id = *$, $id = {}^*id_1$ if $id \leq {}^*id_1$ and finally, $id = id_1^*$ if $id \geq id_1$.

Also, recall that the negation $\mathbf{not}(\mathbf{o}(e_3))[\mathbf{o}(e_1), \mathbf{o}(e_2)]$ can appear in the body of an event pattern rule as syntactic sugar for $\mathbf{o}(e_1) \otimes \neg(\mathtt{path} \otimes \mathbf{o}(e_3) \otimes \mathtt{path}) \otimes \mathbf{o}(e_2)$. Such an event starts when $\mathbf{o}(e_1)\{id_i, id_f\}$ is added to $ESet$, and then we add two rules to the program: $h \otimes\overset{id_i}{\underset{id_f^*}{\leftarrow}} \mathbf{o}(e_2)$ and $(h \otimes\overset{id_i}{\underset{id_f^*}{\leftarrow}} \mathbf{o}(e_2)) \neg\overset{id_i}{\underset{id_f^*}{\leftarrow}} \mathbf{o}(e_3)$. The former rule says that the not-event becomes true when $\mathbf{o}(e_2)$ appears in the $ESet$. The latter rule checks if

$\mathbf{o}(e_3)$ appears in the $ESet$ (before $\mathbf{o}(e_2)$), and in that case, the first temporary rule is removed, so that a later occurrence of e_2 does not make the not-event true. Importantly, the removal of temporary rules that arise from such negation patterns is performed after the fixed point is achieved, separating the monotonic construction of the $ESet$, from the non-monotonic behavior of the rule $_\neg\leftarrow$.

Theorem 1 (Soundness and Completeness of the Procedure). *Let P be a program, π a path, and ϕ a transaction formula. $P, \pi \models \phi$ iff $P, \pi \vdash \phi$*

5 Discussion and Final Remarks

Our work can be compared to solutions that deal with the detection of events, and with the execution of (trans)actions. Event Processing (EP) systems, e.g. [1,23,4], offer very expressive event algebras and corresponding procedures to efficiently detect complex event patterns over large streams of events. As shown in [14], $\mathcal{TR}^{ev}$ can express most event patterns of SNOOP [1] algebra, failing only to translate the expressions requiring the explicit specification of time. Our procedure is inspired on the one from ETALIS [4] algebra, namely the idea of rule binarization, and program transformation. ETALIS has some roots in $\mathcal{TR}$, sharing some of its syntax and connectives, and a similar translation result can be achieved for this algebra (but omitted for lack of space). However ETALIS, as all EP systems, does not deal with the execution of (trans)actions. As such, one can see the procedure presented herein as an extension of ETALIS algorithm with the ability to execute transactions in reaction to the events detected. Moreover, EP-SPARQL [3] is an interesting stream reasoning solution, based on ETALIS, that provides a means to integrate RDF data with event streams for the Semantic Web. We believe that with the correct oracles instantiations, a similar behavior could be achieved in $\mathcal{TR}^{ev}$, and leave this as a future direction.

Several solutions based on action theories exist to model very expressively the effects of transactions that react to events, as [5,7]. However, these are based on active databases, and events are restricted to simple actions like "on insert/delete", thereby failing to encode complex events as defined in EP algebras and $\mathcal{TR}^{ev}$. The work of [11] proposes a policy description language where policies are formulated as sets of ECA rules, and conflicts between policies are captured by logic programs. It ensures transaction-like actions if the user provides the correct specification for conflict rules. Yet, only a relaxed model of transactions can be achieved and it requires a complete low-level specification of the transaction conflicts by the user. In multi-agent systems, [15,12] propose logic programming languages that react and execute actions in response to complex events. Unfortunately, actions fail to follow any kind of transaction model.

Several ECA languages have been proposed in the literature like [2,10,11] with very expressive event and action algebras. However, ECA languages normally do not allow the action to be defined as a transaction, and when they do, they lack from a declarative semantics as [19]; or they are based on active databases and can only detect atomic events defined as insertions/deletes [24,16]. As shown in this paper, with the appropriate instantiations of the $choice$ function, $\mathcal{TR}^{ev}$ can be used as an ECA language where the action is guaranteed to execute as a transaction, and offer different operational behaviors depending on the application needs. Moreover, the procedure presented herein gives an important contribution to implement such an Event-Condition-Transaction language, and closing the existing gap to use it in real scenarios.

References

1. R. Adaikkalavan and S. Chakravarthy. Snoopib: Interval-based event specification and detection for active databases. *Data Knowl. Eng.*, 59(1):139–165, 2006.
2. J. J. Alferes, F. Banti, and A. Brogi. Evolving reactive logic programs. *Intelligenza Artificiale*, 5(1):77–81, 2011.
3. D. Anicic, P. Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *WWW 2011*, pages 635–644, 2011.
4. D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic. Stream reasoning and complex event processing in etalis. *Semantic Web*, 3(4):397–407, 2012.
5. C. Baral, J. Lobo, and G. Trajcevski. Formal characterizations of active databases: Part ii. In *DOOD*, volume 1341 of *LNCS*, pages 247–264. Springer, 1997.
6. E. Behrends, O. Fritzen, W. May, and F. Schenk. Embedding event algebras and process for eca rules for the semantic web. *Fundam. Inform.*, 82(3):237–263, 2008.
7. L. E. Bertossi, J. Pinto, and R. Valdivia. Specifying active databases in the situation calculus. In *SCCC*, pages 32–39. IEEE Computer Society, 1998.
8. A. J. Bonner and M. Kifer. Transaction logic programming. In *ICLP*, pages 257–279, 1993.
9. A. J. Bonner and M. Kifer. Results on reasoning about updates in transaction logic. In *Transactions and Change in Logic Databases*, pages 166–196, 1998.
10. F. Bry, M. Eckert, and P.-L. Patranjan. Reactivity on the web: Paradigms and applications of the language xchange. *J. Web Eng.*, 5(1):3–24, 2006.
11. J. Chomicki, J. Lobo, and S. A. Naqvi. Conflict resolution using logic programming. *IEEE Trans. Knowl. Data Eng.*, 15(1):244–249, 2003.
12. S. Costantini and G. D. Gasperis. Complex reactivity with preferences in rule-based agents. In *RuleML*, pages 167–181, 2012.
13. P. Fodor and M. Kifer. Tabling for transaction logic. In *ACMPPDP*, pages 199–208, 2010.
14. A. S. Gomes and J. J. Alferes. Transaction Logic with (complex) events. *Theory and Practice of Logic Programming, On-line Supplement*, 2014.
15. R. A. Kowalski and F. Sadri. A logic-based framework for reactive systems. In *RuleML*, pages 1–15, 2012.
16. G. Lausen, B. Ludäscher, and W. May. On active deductive databases: The statelog approach. In *Transactions and Change in Logic Databases*, pages 69–106, 1998.
17. A. Margara, J. Urbani, F. van Harmelen, and H. E. Bal. Streaming the web: Reasoning over dynamic data. *J. Web Sem.*, 25:24–44, 2014.
18. R. Müller, U. Greiner, and E. Rahm. AgentWork: a workflow system supporting rule-based workflow adaptation. *Data Knowl. Eng.*, 51(2):223–256, 2004.
19. G. Papamarkos, A. Poulovassilis, and P. T. Wood. Event-condition-action rules on RDF metadata in P2P environments. *Comp. Networks*, 50(10):1513–1532, 2006.
20. Y. Ren and J. Z. Pan. Optimising ontology stream reasoning with truth maintenance system. In *ACM CIKM*, pages 831–836, 2011.
21. M. Rinne, S. Törmä, and E. Nuutila. Sparql-based applications for rdf-encoded sensor data. In *SSN*, pages 81–96, 2012.
22. A. P. Sheth, C. A. Henson, and S. S. Sahoo. Semantic sensor web. *IEEE Internet Computing*, 12(4):78–83, 2008.
23. E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD Conference*, pages 407–418. ACM, 2006.
24. C. Zaniolo. Active database rules with transaction-conscious stable-model semantics. In *DOOD*, pages 55–72, 1995.