How to combine event stream reasoning with transactions for the Semantic Web

Ana Sofia Gomes and José Júlio Alferes*

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

Abstract Semantic Sensor Web is a new trend of research integrating Semantic Web technologies with sensor networks. It uses Semantic Web standards to describe both the data produced by the sensors, but also the sensors and their networks, which enables interoperability of sensor networks, and provides a way to formally analyze and reason about these networks. Since sensors produce data at a very high rate, they require solutions to reason efficiently about what complex events occur based on the data captured. Nevertheless, besides detecting complex events, sensor based applications also need to execute actions in response to these events, and in some cases, to execute these actions in a transactional way. In this paper we propose $\mathcal{TR}^{ev}$ as a solution to combine the detection of complex events with the execution of transactions for these domains. $\mathcal{TR}^{ev}$ is an abstract logic to model and execute reactive transactions. The logic is parametric on a pair of oracles defining the basic primitives of the domain, which makes it suitable for a wide range of applications. In this paper we provide oracle instantiations combining RDF/OWL and relational database semantics for $\mathcal{TR}^{ev}$. Afterwards, based on these oracles, we illustrate how $\mathcal{TR}^{ev}$ can be useful for these domains.

1 Introduction and Motivation

The future of the Internet-of-Things is filled with sensors, and with it, sensor data and sensor networks. Nodes in these sensor networks use the internet to interact and communicate with each other, but also with other services and applications, helping with the detection of changes in the environment, and making daily decisions based on these changes. Today, sensor networks are successfully used in detecting emergency situations, monitoring agriculture conditions and animal farming, industrial control, home automation, patient health surveillance, etc.

With the popularity increase of the Internet-of-Things and the widespread of sensor networks, one important problem is how to deploy sensors' data so it can be accessible by a larger number of different applications and services. Sensors produce data at an extremely high rate, with extremely heterogeneous schemas, vocabularies and data formats, making it very hard to discover and reuse. Based on the premise that it is a waste of resources to use sensors' data for just one single application, an important research effort has been made recently in Semantic

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

Sensor Web (SSW) [20,8] with the goal to enable interoperability between sensors, and reuse of sensors' data. By using Semantic Web technologies, SSW solves this problem by providing a way to semantically describe sensors capabilities, sensors measurements and observations, deployments, etc. With interoperability as one of the main flagships of the Semantic Web, Semantic Web technologies allow one to integrate and reason about knowledge published across different sources, by using RDF as a data model in combination with ontology languages like OWL [16], and to use this knowledge to execute actions in several application domains. In the this context, the Semantic Sensor Network Incubator Group[1] defines an ontology for sensors based on OWL and RDFS, and publishes sensors' data using RDF statements, enabling users to reuse and integrate data from multiple sensors, but also to reason about such data in a powerful way.

In scenarios like sensor networks, where the production of data is high, the fields of Event Processing (EP) and Stream Reasoning provide important solutions to efficiently handle large volumes of data, and to detect complex changes based on these. In these areas, an *event* is a first-class citizen, encoding some change that may be relevant to the system, like e.g. a new sensor's observation. Then, based on the occurrence of a set of events (also called as a *stream*), EP solutions handle the detection of meaningful event patterns (also known as complex events), based on expressive operators and temporal relationship of events. Stream Reasoning exceeds EP by combining streams with domain application knowledge, allowing one to use this knowledge to reason about the events that become true over time. While traditionally EP and Stream Reasoning solutions were designed mostly for databases, the increase of popularity of the Semantic Web and its technologies led to the development of several solutions that can successfully handle and reason about RDF statements and RDFS/OWL models [1,19,15]. However, and by design, EP solutions are incomplete, as they do not deal with the problem of acting upon the event patterns they detect. Detecting these patterns is only meaningful if we can act upon this knowledge, and thus, in general, we need more complete solutions that allow us to define what to do when an event occurs.

In another context, Event-Condition-Action (ECA) languages solve this by explicitly defining how should a system *react* whenever a given pattern is detected. These languages support rules of the form: *on* **event** *if* **condition** *do* **action**, where whenever an event occurs, the condition is checked to hold in the current state and, if that is the case, the action is executed. Today a number of ECA languages exist, providing a semantics for reactive systems in the context of the Semantic Web [3,6,18], multi-agent systems [17,9,13], conflict resolution [7], etc. However, even though ECA languages started in the database context, and many solutions exist with rich languages for defining complex actions, most ECA languages do not allow the action component to be defined as a transaction. Moreover, when they do, they either lack from a declarative semantics (e.g. [18]), or are only suitable for databases since they only detect atomic events defined as primitive insertions/deletes on the database (e.g. [21,14]).

[1] `http://www.w3.org/2005/Incubator/ssn/`

In this case we sustain that in many applications, and especially applications dependent on sensor networks, it is important to guarantee transactional properties, like consistency or atomicity, over the execution of a set of actions issued in response to events. As an application scenario, consider the case where the police wants to monitor and detect traffic violations based on a sensor network deployed in some road. A sensor in such a network can identify plates of vehicles and distinguish between types of vehicles. Then, based on the information that the sensor publishes in RDF, the application must reason about what vehicles are indulging in traffic violations and, in these cases, issue fines for these violations and notify the corresponding drivers. Clearly, some transactional behavior 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.

$\mathcal{TR}$ [4] is a general purpose logic to model and reason about the executional behavior of transactions. It provides a general model theory that is parametric on a pair of oracles defining the semantics of states and updates of the knowledge base (KB) (e.g. relational databases, action languages, description logics, etc.). With it, one can reason about the sequence of states (also denoted as a *path*) 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 the Horn-like clauses of logic programming. However, $\mathcal{TR}$ fails to deal simultaneously with complex events and transactions, and for that we have previously proposed $\mathcal{TR}^{ev}$ in [11]. $\mathcal{TR}^{ev}$ is an extension of $\mathcal{TR}$ that can reason about the execution of transactions, but also about the complex events that become true in this transaction execution. Just like in EP algebras, with $\mathcal{TR}^{ev}$ one is able to define complex events by combining atomic (or other complex) events with temporal operators. In $\mathcal{TR}^{ev}$, atomic events can either be external events, which are signalled to the KB, primitive updates in the KB (similarly e.g. to the events "on insert" in databases), or events that the oracle defines to occur in state transitions. Moreover, as in active databases, transactions in $\mathcal{TR}^{ev}$ are *constrained* by the events that occur during their execution: a transaction can only successfully commit when all events triggered during its execution are addressed. $\mathcal{TR}^{ev}$ is parameterized with a pair of oracles as in the original $\mathcal{TR}$, but also takes an additional *choice* function, which abstracts the semantics of a reactive language from its response policies decisions. Of course, to be put to work in specific domains, $\mathcal{TR}^{ev}$ (and $\mathcal{TR}$) require the instantiation of such oracle.

In this paper we propose $\mathcal{TR}^{ev}$ as a solution to combine heterogeneous event stream reasoning with the execution of transactions for sensor networks that use Semantic Web technologies. This is done by providing an appropriate oracle instantiation to reason about RDF/OWL semantics. With it, one can decide what events become true in a given path, based on the occurrence of atomic events and the knowledge inferred from the sensors' ontology. After defining such an oracle, we provide an elaborated example to illustrate what kind of event stream reasoning can be done using $\mathcal{TR}^{ev}$, and how to combine more than one oracle instantiation for a sensor based application that uses RDF/OWL to

describe and reason about events together with a relational database to perform transactions.

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

Transaction Logic [4], $\mathcal{TR}$, is a logic to execute and reason about general changes in a KB, when these changes need to follow a transactional behavior. In a nutshell[2], $\mathcal{TR}$ 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. Moreover, $\phi \wedge \psi$ 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. A key feature of $\mathcal{TR}$ is the separation of primitive operations from the logic of combining them. $\mathcal{TR}$'s theory is parametric on two different oracles allowing the incorporation of a wide variety of KB semantics, from classical to non-monotonic to various other non-standard logics. These oracles abstract the representation of KB states and how to query them (by including the data oracle $\mathcal{O}^d$), and abstract the way states change (defined by the transition oracle $\mathcal{O}^t$). Consequently, the language of primitive queries and actions is not fixed, and neither is the definition of what is a state. To distinguish between states, $\mathcal{TR}$ works with a set of state identifiers to uniquely identify a state. With this, the *data oracle* $\mathcal{O}^d$ is a mapping from state identifiers to sets of formulas where, given a state identifier i, $\mathcal{O}^d(i)$ returns the set of formulas true in state i. The *state transition oracle* $\mathcal{O}^t(i_1, i_2)$ is a function that maps pairs of KB states into sets of ground atoms called elementary transitions, where given two state identifiers i_1 and i_2, $\mathcal{O}^t(i_1, i_2)$ returns the set of elementary transitions that are true when the KB changes from state i_1 into i_2.

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 written $P, D_0 \models \phi$, if there is a path $D_0, D_1, \ldots, D_n$, which starts in that state D_0, and 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 a atomic way. Non-deterministic transactions are possible, and in this case several successful paths exist. For a special class of $\mathcal{TR}$ theories (known as serial-Horn programs) there is a proof procedure and corresponding implementation [4,10].

[2] For lack of space, and since $\mathcal{TR}^{ev}$ is an extension of $\mathcal{TR}$ (cf. [11]) we do not make a thorough overview of $\mathcal{TR}$ here. For complete details see e.g. [11,4]

$\mathcal{TR}^{ev}$ 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 $\models$. $\mathcal{TR}^{ev}$'s alphabet 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 ($\mathcal{P}_\mathcal{O}$). Importantly, to support event stream reasoning in the oracle side, for this paper, we also consider the case where oracles primitives are partitioned into oracle actions $\mathcal{P}_{\mathcal{O}_a}$ and oracle events $\mathcal{P}_{\mathcal{O}_e}$. Finally, formulas in $\mathcal{TR}^{ev}$ are partitioned into transaction formulas and event formulas, and are evaluated differently: event formulas are meant to be detected w.r.t. a path encoding the history of execution; while transaction formulas are meant to be executed. One of the goals of $\mathcal{TR}^{ev}$'s theory is to find the paths where a given *reactive* transaction formula ϕ successfully executes.

Event formulas, i.e. formulas that can be *detected*, 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 `path` is just any tautology, cf. [5]), with the meaning: "ϕ followed by 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 signalling changes in the KB, needed to allow reactive rules similar to e.g. "on insert" triggers in databases). *Transaction formulas*, i.e. formulas that can be *executed*, as in $\mathcal{TR}$ are either a transaction atom, or an expression defined inductively as $\neg\phi$, $\Diamond\phi$, $\phi \wedge \psi$, $\phi \vee \psi$, or $\phi \otimes \psi$. In $\mathcal{TR}^{ev}$, 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 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.

Central to $\mathcal{TR}^{ev}$'s theory 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 in a path, if all the events occurring in that path are 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, also a *choice* function defining what event should be selected for being 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 in a wide spectrum of applications.

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. As in $\mathcal{TR}$, $\mathcal{TR}^{ev}$'s formulas are also 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. This is defined as $P, D_0^- \models e_1 \otimes \ldots \otimes e_k$, where D_0 is the initial KB state and $e_1 \otimes \ldots \otimes e_k$ is the sequence of events that arrived. A path $D_0 \overset{O_1}{\rightarrow} \ldots \overset{O_n}{\rightarrow} D_n$ that make $P, D_0^- \models e_1 \otimes \ldots \otimes e_k$ true, represents a KB evolution responding to $e_1 \otimes \ldots \otimes e_k$

As usual, satisfaction of formulas is based on interpretations which define what atoms are true over what paths, by mapping paths to sets 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:

Definition 1 (Interpretation). *An interpretation M is a mapping assigning a set of atoms (or $\top$[3]) to every possible path, with the restrictions (where D_is are states, and φ an atom):*

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* $\varphi \in \mathcal{O}^t(D_1, D_2) \wedge \varphi \in \mathcal{P}_{\mathcal{O}_a}$
3. $\mathbf{o}(\varphi) \in M(\langle D_1 \overset{\mathbf{o}(\varphi)}{\rightarrow} D_2 \rangle)$	*if* $\mathbf{o}(\varphi) \in \mathcal{O}^t(D_1, D_2) \wedge \mathbf{o}(\varphi) \in \mathcal{P}_{\mathcal{O}_e}$
4. $\mathbf{o}(e) \in M(\langle D \overset{\mathbf{o}(e)}{\rightarrow} D \rangle)$	

Understanding this notion of interpretation, and its restrictions, is important for understanding $\mathcal{TR}^{ev}$'s semantics. The first three points above, force all interpretations to satisfy primitive formulas on the paths where the oracles satisfy them, i.e., only the mappings that comply with the specified oracles are considered as interpretations. The second point also states that, whenever a primitive action φ (e.g. the insertion of a fact in the KB) is made true by the oracle, the occurrence associated with the primitive action $\mathbf{o}(\varphi)$ (e.g. "on insert" of that fact) is also made true in every M, and in this case, the path is annotated with φ's occurrence. As such, this restriction guarantees compliance with the oracles, viz. whenever the oracle satisfies a primitive action in a transition, all Ms also satisfy both the primitive action, and the primitive occurrence in that same transition. Similarly, the third point makes the correspondence between the primitive events defined by the oracle, and the primitive events made true by Ms. This allows the oracle to define primitive events different from primitive actions, and make interpretations satisfy these events in these transitions.

Finally, the fourth point guarantees that, whenever an event is observed to occur in a transition, then all interpretations necessarily satisfy this occurrence. This point is an important technical detail to satisfy the action of explicitly triggering an event. By forcing M to satisfy $\mathbf{o}(e)$ whenever it appears explicitly in the history of the path, we impose compliance between the history of occurrences on a path and the set of formulas that interpretations make true on that same path. Note that making the occurrence of an event explicitly true does not change the KB state *per se* and thus, these transitions only take place on paths

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

where the current state does not evolve. However, as we shall see, $\mathcal{TR}^{ev}$ theory imposes that, whenever $\mathbf{o}(e)$ is true in some part of a path (or subpath), then for a transaction to be satisfied, $\mathbf{r}(e)$ must also be true. Thus naturally, some actions may need to be executed to satisfy $\mathbf{r}(e)$ as an implicit result of making this occurrence true, which in turn, may cause changes in the KB.

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 point onwards.

Definition 2 (Path Splits, Subpaths and Prefixes). *Let π be a k-path, i.e. a path of length k of the form $\langle D_1 \stackrel{O_1}{\rightarrow} \ldots \stackrel{O_{k-1}}{\rightarrow} D_k\rangle$. A* split *of π is any pair of subpaths, π_1 and π_2, s.t. $\pi_1 = \langle D_1 \stackrel{O_1}{\rightarrow} \ldots \stackrel{O_{i-1}}{\rightarrow} D_i\rangle$ and $\pi_2 = \langle D_i \stackrel{O_i}{\rightarrow} \ldots \stackrel{O_{k-1}}{\rightarrow} D_k\rangle$ 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 the order of the states is preserved. A prefix π_1 of π is any subpath of π sharing the initial state.

As mentioned above, satisfaction of complex formulas is different for event formulas and transaction formulas. While the former concerns the *detection* of an event, the latter 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 in a prefix π_1 of π (i.e. where $\pi = \pi_1 \circ \pi_2$), and all events occurring over π_1 are *responded to* in 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* function picks, at each moment, the next event unanswered event to respond to. First it has to decide what events are unanswered in a path π w.r.t. an interpretation M and, based on a given criteria, selects 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. Before defining this *choice* function, we first define what is an expansion of a path. Nevertheless, an important notion here is that, if all events that occur on a path π are answered on π w.r.t. M, then $choice(M, \pi) = \epsilon$.

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; and finally, 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 unanswered events 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. Note that, since complex events are possible, in general nothing prevents π_{i+1} to have more unanswered events than π_i. In fact, it may be impossible to address all events in a finite path, and in that case, such a sequence of paths does not exists. In fact, non-termination is a known issue of reactive rules, and is an undecidable problem in the general case [2].

These definitions leave open the *choice* function, that is taken as a further parameter of $\mathcal{TR}^{ev}$, and specifies how to choose the next unanswered event to respond to. For its instantiation one needs to decide: 1) in which order should events be responded and 2) how should an event be responded. The former defines the handling order of events in case of conflict, e.g. based on when events have occurred (temporal order), on a priority list, or any other criteria. The

latter defines the response policy of an ECA-language, i.e. when is an event considered to be responded. E.g., if an event occurs more than once before the system can respond to it, this specifies if such response should be issued only once or equally to the amount of occurrences. Choosing the appropriate operational semantics depends on the application in mind. For this paper, we fix an instantiation of *choice* function, where events are responded in the (temporal) order in which they occurred, and events for which there was already a response are not responded to again:

Definition 7 (Temporal *choice*). *Let M be an interpretation and π a path. The temporal function is $choice(M, \pi) = firstUnans(M, \pi, order(M, \pi))$ where:*

- *$order(M, \pi) = \langle e_1, \ldots, e_n \rangle$ iff $\forall e_i$ $1 \leq i \leq n$, $\exists \pi_i$ subpath of π where $M, \pi \models_{ev} \mathbf{o}(e_i)$ and $\forall e_j$ s.t. $i < j$ then e_j occurs after e_i*
- *e_2 occurs after e_1 w.r.t. π and M iff there exists π_1, π_2 subpaths of π such that $\pi_1 = \langle D_i \xrightarrow{O_i} \ldots \xrightarrow{O_{j-1}} D_j \rangle$, $\pi_2 = \langle D_n \xrightarrow{O_n} \ldots \xrightarrow{O_{m-1}} D_m \rangle$, $M, \pi_1 \models_{ev} \mathbf{o}(e_1)$, $M, \pi_2 \models_{ev} \mathbf{o}(e_2)$ and $D_j \leq D_m$ w.r.t. the ordering in π.*
- *$firstUnans(M, \pi, \langle e_1, \ldots, e_n \rangle) = e_i$ iff e_i is the first event in $\langle e_1, \ldots, e_n \rangle$ where given π' subpath of π and $M, \pi' \models_{ev} \mathbf{o}(e)$ then $\neg \exists \pi''$ s.t. π'' is also a subpath of π, π'' is after π' and $M, \pi'' \models \mathbf{r}(e)$.*

Afterwards, we define the notion of *model* of formulas and programs.

Definition 8 (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.*
Let M_1, M_2 be interpretations, $M_1 \leq M_2$ if $\forall \pi$: $M_2(\pi) = \top \vee M_1(\pi) \subseteq M_2(\pi)$.
Let ϕ be a formula, and P a program. M is a minimal model *of ϕ (resp. P) if M is a model of ϕ (resp. P) and $M \leq M'$ for every model M' of ϕ (resp. P).*

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 whether a formula succeeds in 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 9 ($\mathcal{TR}^{ev}$ 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 true, if there is a path $D_1 \xrightarrow{O_0} \ldots \xrightarrow{O_n} D_n$ that makes ($\star$) true.

3 Oracles for stream reasoning

$\mathcal{TR}^{ev}$ provides a powerful theory to talk about executional properties of abstract reactive transactions. With it, one is able to say what properties (or fluents) hold

for every possible path of execution, or express relations between transactions and events, e.g. to say "event ψ occurs whenever transaction ϕ succeeds". In addition, with $\mathcal{TR}^{ev}$'s proof theory, one can also talk about a particular execution path, and say exactly *how* an abstract reactive transaction succeeds.

Of course, to use $\mathcal{TR}^{ev}$ in applications one needs to instantiate the appropriate oracles $\mathcal{O}^d$ and $\mathcal{O}^t$, on which $\mathcal{TR}^{ev}$ is parametric, that describe the behavior of the KBs in the domain at hands. As illustration of how this can be done, consider the relational oracle proposed in [4]:

Definition 10 (Relational Oracle). *In a relational oracle, states can be represented by sets of ground atomic formulas. The data oracle simply returns all these formulas, i.e., $\mathcal{O}^d(D) = D$. Moreover, for each predicate symbol p in D, the transition oracle defines two new predicates, $p.ins$ and $p.del$ representing the insertion and deletion atoms, respectively. Formally, $p.ins \in \mathcal{O}^t(D_1, D_2)$ iff $D_2 = D_1 \cup \{p\}$ and, $p.del \in \mathcal{O}^t(D_1, D_2)$ iff $D_2 = D_1 \backslash \{p\}$.*

Example 1 (Financial Transactions - adapted from [4]). Consider a bank's KB defined by the relational database of Definition 10 and where the balance of a bank account is given by the relation $\texttt{balance}(\texttt{Acnt}, \texttt{Amt})$. Using just $\_.ins$ and $\_.del$ as primitive actions, we define the transactions: $\texttt{withdraw}(\texttt{Amt}, \texttt{Acnt})$ to withdraw an amount from an account; $\texttt{deposit}(\texttt{Amt}, \texttt{Acnt})$ to deposit an amount into an account; $\texttt{changeBalance}(\texttt{Acnt}, \texttt{Bal}, \texttt{Bal}')$ to change an account's balance; and, finally, $\texttt{transfer}(\texttt{Amt}, \texttt{Acnt}, \texttt{Acnt}')$ for transferring an amount from one account to another. In $\mathcal{TR}^{ev}$ (and also in $\mathcal{TR}$) these can be defined in a logic programming style by the following rules:

$$\begin{aligned}
\texttt{transfer}(\texttt{Amt}, \texttt{Acnt}, \texttt{Acnt}') &\leftarrow \texttt{withdraw}(\texttt{Amt}, \texttt{Acnt}) \otimes \texttt{deposit}(\texttt{Amt}, \texttt{Acnt}') \\
\texttt{withdraw}(\texttt{Amt}, \texttt{Acnt}) &\leftarrow \texttt{balance}(\texttt{Acnt}, \texttt{B}) \otimes \texttt{changeBalance}(\texttt{Acnt}, \texttt{B}, \texttt{B} - \texttt{Amt}) \\
\texttt{deposit}(\texttt{Amt}, \texttt{Acnt}) &\leftarrow \texttt{balance}(\texttt{Acnt}, \texttt{B}) \otimes \texttt{changeBalance}(\texttt{Acnt}, \texttt{B}, \texttt{B} + \texttt{Amt}) \\
\texttt{changeBalance}(\texttt{Acnt}, \texttt{B}, \texttt{B}') &\leftarrow \texttt{balance}(\texttt{Acnt}, \texttt{B}).\texttt{del} \otimes \texttt{balance}(\texttt{Acnt}, \texttt{B}').\texttt{ins}
\end{aligned}$$

$P, \langle d_1, d_2, d_3, d_4, d_5 \rangle \models \texttt{transfer}(\texttt{10}, \texttt{ac1}, \texttt{ac2})$ holds, if d_1 is e.g. a state where $\texttt{balance}(\texttt{ac1}, \texttt{20})$ and $\texttt{balance}(\texttt{ac2}, \texttt{30})$ are true, d_2 is a state obtained from d_1 by deleting $\texttt{balance}(\texttt{ac1}, \texttt{20})$; d_3 is d_2 plus $\texttt{balance}(\texttt{ac1}, \texttt{10})$; d_4 is d_3 minus $\texttt{balance}(\texttt{ac2}, \texttt{30})$; and finally d_5 is obtained from d_4 by adding $\texttt{balance}(\texttt{ac2}, \texttt{40})$.

We can also define complex event rules and their associated responses. E.g., the following event $\mathbf{o}(\texttt{balanceViolation}(\texttt{Acnt}))$ occurs the first time the account balance is updated into a negative value, and in that case, the bank charges 5€ to the customer for that violation. This is expressed in $\mathcal{TR}^{ev}$ as follows:

$$\begin{aligned}
\mathbf{o}(\texttt{balanceViolation}(\texttt{Acnt})) &\leftarrow (\mathbf{o}(\texttt{balance}(\texttt{Acnt}, \texttt{B}).\texttt{del}) \otimes \mathbf{o}(\texttt{balance}(\texttt{Acnt}, \texttt{B}').\texttt{ins})) \\
&\qquad \wedge (\texttt{B}' < \texttt{0} \leq \texttt{B}) \\
\mathbf{r}(\texttt{balanceViolation}(\texttt{Acnt})) &\leftarrow \texttt{balance}(\texttt{Acnt}, \texttt{B}) \otimes \texttt{changeBalance}(\texttt{Acnt}, \texttt{B}, \texttt{B} - \texttt{5})
\end{aligned}$$

Now imagine that we start on a state d_1' where $\texttt{balance}(\texttt{ac1}, \texttt{5})$ is true instead of $\texttt{balance}(\texttt{ac1}, \texttt{20})$. Then, $\texttt{transfer}(\texttt{10}, \texttt{ac1}, \texttt{ac2})$ to succeed from d_1' needs an expanded path $\langle d_1', d_2', d_3', d_4', d_5', d_6', d_7' \rangle$, where d_6', d_7' satisfy the action $\texttt{changeBalance}(\texttt{ac1}, -\texttt{5}, -\texttt{10})$, changing the balance of the account a_1 into $\texttt{balance}(\texttt{ac1}, -\texttt{10})$. I.e., for the transaction to succeed, it needs to respond to the event $\mathbf{o}(\texttt{balanceViolation}(\texttt{Acnt}))$ that becomes true during its execution.

The later example shows how one can use $\mathcal{TR}^{ev}$ to reason about what are the paths that make a transaction succeed, given a set of basic primitives (actions and queries) defined by a pair of relational oracles. Note that oracles have an important role in $\mathcal{TR}^{ev}$, as one can only write transactions combining these primitives after knowing exactly what oracle primitives are available. The logic then takes care of the semantics of complex (trans)actions, defining over what paths such a complex transaction can succeed.

Moreover, while the previous relational oracles are rather simple, nothing prevent us from using more powerful and expressive oracles, or to combine of several oracles into one, making $\mathcal{TR}^{ev}$ useful in more sophisticated applications. In the following, we provide a new oracle definition that, as we shall see, can be used with $\mathcal{TR}^{ev}$ to perform event stream reasoning. We start by defining a data oracle based on RDF with an ontology model defined in OWL:

Definition 11 (RDF data Oracle). *A state is an RDF graph G, i.e., a set of RDF triples of the form $(s\ p\ o)$ together with an OWL ontology. The data oracle ($\mathcal{O}^d$) is defined such that $\mathcal{O}^d(G) \models (s\ p\ o)$ iff $(s\ p\ o) \in Closure(G)$, where $Closure(G)$ is the closure of the graph under the ontology.*

Just like a state in the relational database oracle is represented by the set of formulas that are in the database, a state in the latter oracle is simply a set of instances defined in RDF triples, together with the ontology. This oracle also assumes a function Closure(G) that computes the whole model of the RDF instance graph under the ontology.

Based on this function, we now define the possible transitions for an RDF/OWL graph, where the primitive actions are insertions and deletions of graphs composed by RDF instances[4]. In this case, inserting (or deleting) an RDF instance graph means to add (or remove) every individual triple to the graph. Notice that insertion a triple is the special case where the graph is a set of just one element. Similarly to the relational oracle, we assume the primitives *graph.ins* and *graph.del* where *graph* is a set of RDF triples. Recall that the syntactic choice of *.ins* and *.del* has no particular meaning in $\mathcal{TR}^{ev}$, and we could have chosen any other representation as e.g., $insert(graph)$ and $delete(graph)$.

Definition 12 (RDF transition Oracle). *Let g_1 be an RDF graph, i.e., a set of RDF triples of the form $(s\ p\ o)$.*
$\mathcal{O}^t(D_1, D_2) \models g_1.ins$ iff both statements are true:
- $D_2 = D_1 \cup \{(s\ p\ o) : (s\ p\ o) \in g_1\}$ *and;*
- $\mathcal{O}^t(D_1, D_2) = \{g_1.ins\} \cup \{\mathbf{o}((s\ p\ o).ins) : (s\ p\ o) \in Closure(D_2) \backslash Closure(D_1)\}$

$\mathcal{O}^t(D_1, D_2) \models g_1.del$ iff both statements are true:
- $D_2 = D_1 \cap \{(s\ p\ o) : (s\ p\ o) \in g_1\}$ *and;*
- $\mathcal{O}^t(D_1, D_2) = \{g_1.del\} \cup \{\mathbf{o}((s\ p\ o).del) : (s\ p\ o) \in Closure(D_1) \backslash Closure(D_2)\}$

Notice that in the latter definition, $\mathcal{O}^t$ explicitly defines a set of primitive events true in a transition of states. This definition of $\mathcal{O}^t$ allows one to distinguish

[4] To simplify, and since in most SSW applications this is not needed, we do not consider the case of updating the OWL ontology.

between the primitive actions executed by $\mathcal{TR}^{ev}$, and the primitive events that occurred as a result of this action. Namely, while in the insertion of an instance graph g_1, $\mathcal{O}^t$ only makes $g_1.ins$ true, it also satisfies the occurrences of primitive actions executed as a consequence of $g_1.ins$. This allows us to reason about what action was really executed ($g_1.ins$) in the transition by $\mathcal{TR}^{ev}$, but also about what happened inside the oracle as a consequence of this action. As we shall see next, this allows us to use application's knowledge to reason about what events hold, not only inside $\mathcal{TR}^{ev}$'s rules, but also at the oracle level.

4 An example combining event stream reasoning and transaction execution

After defining oracles to reason about RDF/OWL graphs, we can now show how to use these oracles for SSW domains. Moreover, in these domains, it is often useful to use more than one representation semantics of states and actions. In fact, this is the case in the application example described in the introduction, where we need to combine data produced by a sensor network (published in RDF/OWL), with the government's relational database comprising information about drivers, fines, addresses, etc.

Although formally we can only have one oracle defining the primitives to query ($\mathcal{O}^d$), and one oracle defining the primitives to execute actions ($\mathcal{O}^t$), nothing prevents these oracles from being instantiated with more than one semantics. This is easily done by partitioning the oracle primitives ($\mathcal{P}_\mathcal{O}$) into as many as needed and, based on this partition, use $\mathcal{O}^t$ and $\mathcal{O}^d$ as "meta-oracles" deciding in which semantics a formula should be evaluated. Next we illustrate how to do this, and how to perform stream reasoning using the previously defined oracles.

Example 2. Consider the situation from the introduction, where we have a government's application to detect and issue fines for traffic violations. To detect traffic violations, the government depends on a sensor network deployed on some road. To model this network, its sensors, and sensors' observations, we have a Semantic Sensor Network based on OWL ontology, that publishes observations data using RDF triples. Besides information about the sensors, this ontology also describes information about the vehicles observed by the sensors. Such an ontology can include e.g., that `lightVehicle` and `heavyVehicle` are subclasses of `motorVehicle`[5], and that `sensor1` and `sensor2` are instances of type `Sensor`:

```
ov : vehicle          rdf : type            owl : Class .
ov : motorVehicle     rdfs : subClassOf     ov : vehicle .
ov : lightVehicle     rdfs : subClassOf     ov : motorVehicle .
ov : heavyVehicle     rdfs : subClassOf     ov : vehicle .
ov : sensor           rdf : type            owl : Class .
ov : sensor1          rdf : type            ov : sensor .
ov : sensor2          rdf : type            ov : sensor .
```

[5] Although, for this example, we chose to express the properties and knowledge about vehicles in our local ontology, we could have alternatively used any other external ontology to describe vehicles like, e.g., the Vehicular Sales Ontology [12].

where as usual `rdf`,`rdfs` and `owl` are the default namespaces for RDF, RDFS and OWL, and `ov` is the application's namespace where the objects and properties of the vehicular ontology are defined, and which includes additional statements.

The information about drivers, fines and addresses is on a government's relational database, and actions are performed w.r.t. this database. E.g., the following $\mathcal{TR}^{ev}$ rules define that processing a given violation `V` of a vehicle with plate `P` at a date-time `DT` is done by identifying, in the government's relational database, the cost `Cost` of the violation and the driver `D` of the vehicle, to insert into the database that the fine was issued for that driver, and to notify the driver:

$$\begin{aligned}
&\texttt{processViolation}(\texttt{P},\texttt{DT},\texttt{V}) \leftarrow \texttt{fineCost}(\texttt{V},\texttt{Cost}) \otimes \texttt{isDriver}(\texttt{P},\texttt{D}) \otimes \\
&\qquad \texttt{fineIssued}(\texttt{P},\texttt{D},\texttt{DT},\texttt{Cost}).\texttt{ins} \otimes \texttt{notifyFine}(\texttt{P},\texttt{D},\texttt{DT},\texttt{Cost}) \\
&\texttt{notifyFine}(\texttt{P},\texttt{D},\texttt{DT},\texttt{Cost}) \leftarrow \texttt{hasAddress}(\texttt{D},\texttt{Addr}) \otimes \texttt{sendLetter}(\texttt{D},\texttt{Addr},\texttt{P},\texttt{DT},\texttt{Cost})
\end{aligned}$$

Then, we can write events of interest in $\mathcal{TR}^{ev}$. E.g., in the following (simplified) rules we define the event $\mathbf{o}(\texttt{passingSpeedA}_1(\texttt{P},\texttt{VType},\texttt{S},\texttt{DT}))$ which detects if a vehicle with plate `P` and type `VType` has passed in area $\texttt{a}_1$ at time `DT` with speed `S`; or $\mathbf{o}(\texttt{passingWrongWay}(\texttt{P},\texttt{DT}))$ detecting any vehicle plate `P` passing the road in the wrong way at time `DT`, as long as this vehicle has the type `motorVehicle`:

$$\begin{aligned}
&\mathbf{o}(\texttt{passingSpeedA}_1(\texttt{P},\texttt{VType},\texttt{S},\texttt{DT}_2,\texttt{S}_2)) \leftarrow \\
&\quad ([\mathbf{o}((\texttt{Obs}_1\ \texttt{ov:plateRead}\ \texttt{P}).\texttt{ins}) \wedge\ \mathbf{o}((\texttt{Obs}_1\ \texttt{ov:vehicleDetected}\ \texttt{VType}).\texttt{ins}) \\
&\quad \wedge \mathbf{o}((\texttt{Obs}_1\ \texttt{ov:dateTime}\ \texttt{DT}_1).\texttt{ins}) \wedge\ \mathbf{o}((\texttt{Obs}_1\ \texttt{ov:readBy}\ \texttt{sensor1}).\texttt{ins})] \\
&\quad \otimes [\mathbf{o}((\texttt{Obs}_2\ \texttt{ov:plateRead}\ \texttt{P}).\texttt{ins}) \wedge\ \mathbf{o}((\texttt{Obs}_2\ \texttt{ov:vehicleDetected}\ \texttt{VType}).\texttt{ins}) \\
&\quad \wedge \mathbf{o}((\texttt{Obs}_2\ \texttt{ov:dateTime}\ \texttt{DT}_2).\texttt{ins})) \wedge\ \mathbf{o}((\texttt{Obs}_2\ \texttt{ov:readBy}\ \texttt{sensor2}).\texttt{ins})]) \\
&\quad \wedge ((\texttt{DT}_2 > \texttt{DT}_1) \wedge \texttt{S} = (10/\texttt{DT}_1 - \texttt{DT}_2)) \\
&\mathbf{o}(\texttt{passingWrongWay}(\texttt{P},\texttt{DT}_1)) \leftarrow \\
&\quad (\mathbf{o}((\texttt{Obs}_1\ \texttt{ov:plateRead}\ \texttt{P}).\texttt{ins}) \wedge \mathbf{o}((\texttt{Obs}_1\ \texttt{ov:vehicleDetected}\ \texttt{motorVehicle}).\texttt{ins}) \\
&\quad \wedge \mathbf{o}((\texttt{Obs}_1\ \texttt{ov:dateTime}\ \texttt{DT}_1).\texttt{ins}) \wedge \mathbf{o}((\texttt{Obs}_1\ \texttt{ov:readBy}\ \texttt{sensor2}).\texttt{ins})) \\
&\quad \otimes (\mathbf{o}((\texttt{Obs}_2\ \texttt{ov:plateRead}\ \texttt{P}).\texttt{ins}) \wedge \mathbf{o}((\texttt{Obs}_2\ \texttt{ov:vehicleDetected}\ \texttt{motorVehicle}).\texttt{ins}) \\
&\quad \wedge \mathbf{o}((\texttt{Obs}_2\ \texttt{ov:dateTime}\ \texttt{DT}_2).\texttt{ins}) \wedge \mathbf{o}((\texttt{Obs}_2\ \texttt{ov:readBy}\ \texttt{sensor1}).\texttt{ins})) \wedge\ (\texttt{DT}_1 < \texttt{DT}_2) \\
&\mathbf{o}(\texttt{passingSpeed}(\texttt{P},\texttt{VType},\texttt{S},\texttt{DT}_2,\texttt{a1})) \leftarrow \mathbf{o}(\texttt{passingSpeedA}_1(\texttt{P},\texttt{VType},\texttt{S},\texttt{DT}_2))
\end{aligned}$$

$$\begin{aligned}
&\mathbf{r}(\texttt{passingSpeed}(\texttt{P},\texttt{VType},\texttt{S},\texttt{DT},\texttt{A})) \leftarrow \\
&\qquad \texttt{maxSpeed}(\texttt{VType},\texttt{A},\texttt{MS}) \otimes (\texttt{MS} \leq \texttt{S}) \otimes \texttt{processViolation}(\texttt{P},\texttt{DT},\texttt{speed}) \\
&\mathbf{r}(\texttt{passingSpeed}(_,\texttt{VType},\texttt{S},_,\texttt{A})) \leftarrow \texttt{maxSpeed}(\texttt{VType},\texttt{A},\texttt{MS}) \otimes (\texttt{MS} > \texttt{S}) \\
&\mathbf{r}(\texttt{passingWrongWay}(\texttt{P},\texttt{DT})) \leftarrow \texttt{processViolation}(\texttt{P},\texttt{DT},\texttt{wrongWay})
\end{aligned}$$

In the rules above we also define what is executed whenever these events occur. Namely, we say that `processViolation` is only executed for the event `passingSpeed` if the vehicle's detected speed exceeds the speed limit, and always executed if `passingWrongWay` is detected.

With these rules, our system can prove statements of the form: $P, S_1{-} \models obs_1.ins \otimes obs_2.ins \otimes \ldots \otimes obs_n.ins$ where, based on given starting state S_1, $\mathcal{TR}^{ev}$ computes the path $\langle S_1\ {}^{O_1}\!\!\rightarrow \ldots\ {}^{O_{n-1}}\!\!\rightarrow S_n\rangle$ satisfying the sequence of observations obtained so far. I.e., it computes *how* the system should evolve in order to respond to these observations, in a transactional way, and according to a $\mathcal{TR}^{ev}$ program P containing the rules above. Note that, since we are considering two different KBs, each state S_i in the path is a composed state (G_i, D_i), where

G_i is the RDF graph describing vehicles and sensors' observations, and D_i is a state of the government's relational database. With this setting, let's assume we want to prove $P, S_1 \sim\!\models (\texttt{ov:obs}_1).\texttt{ins} \otimes (\texttt{ov:obs}_2).\texttt{ins}$ where:

```
ov : obs1    rdf : type                ov : Observation ;
             ov : plateRead            "01-01-AA" ;
             ov : dateTime             1426325213000 ;
             ov : vehicleDetected      ov : heavyVehicle ;
             ov : readBy               ov : sensor1 .
ov : obs2    rdf : type                ov : Observation ;
             ov : plateRead            "01-01-AA" ;
             ov : dateTime             1426325213516 ;
             ov : vehicleDetected      ov : heavyVehicle ;
             ov : readBy               ov : sensor2 .
```

Then, based on the ontology definition, we know that $\texttt{heavyVehicle} \sqsubseteq \texttt{vehicle}$, and thus $\mathbf{o}((\texttt{ov:obs}_1\ \texttt{ov:vehicleDetected}\ \texttt{motorVehicle}).\texttt{ins})$ will hold at the same time (i.e., transition) as $\mathbf{o}((\texttt{ov:obs}_1).\texttt{ins})$. In a similar way, the event $\mathbf{o}((\texttt{ov:obs}_2\ \texttt{ov:vehicleDetected}\ \texttt{motorVehicle}).\texttt{ins})$ will hold at the same as $\mathbf{o}((\texttt{ov:obs}_1).\texttt{ins})$. From this, $\mathbf{o}(\texttt{passingWrongWay}(\texttt{"01-01-AA"}, 1426325213000)$ holds for the same transition as where the actions $(\texttt{ov:obs}_1).\texttt{ins} \otimes (\texttt{ov:obs}_2).\texttt{ins}$ occur, and thus the *transaction* $(\texttt{ov:obs}_1).\texttt{ins} \otimes (\texttt{ov:obs}_2).\texttt{ins}$ will only succeed in an expanded path where the driver of vehicle `"01-01-AA"` is fined and notified, for the infraction of passing the road in the wrong way.

5 Discussion and Final Remarks

In this paper we propose a set of oracle instantiations to make $\mathcal{TR}^{ev}$ useful for domains involving sensor networks and Semantic Web technologies. With it, one can use $\mathcal{TR}^{ev}$ to reason about what complex events occur, and what transactions need to be executed to respond to these events. Moreover, like in EP/Stream Reasoning solutions [1,19,15], $\mathcal{TR}^{ev}$ can use the domain's application knowledge to reason about what complex events occur. This reasoning can be done either inside the oracle, using the oracle's domain knowledge to trigger primitive events, but also inside $\mathcal{TR}^{ev}$ rules, where we use this knowledge to decide what should be the response of the system for a given event.

Since EP/Stream Reasoning only deal with detecting complex event patterns, and not with executing actions, our work can be better compared with ECA solutions. While several ECA languages exist for several domains like the Semantic Web [3,6,18] they normally do not support the execution of transactions. Some exceptions exist, but are either only procedural like [18], or can only detect simple events based on database inserts and deletes [21,14].

This is, in fact, one thing that distinguishes $\mathcal{TR}^{ev}$ from most solutions: combining the ability to detect and reason about complex and sophisticated event patterns, with the execution of complex transactions, and to do this in a way that can be useful for a wide range of applications by plugging in different oracles. The example presented in Section 4 uses a concrete oracle parametrization combining RDF/OWL and relational database semantics, and which is interesting

for SSW applications. With it, one can use the sensor network ontology to help reason about the events that occur in a given transition, while simultaneously combining the execution of (trans)actions in the relational database.

References

1. 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.
2. J. Bailey, G. Dong, and K. Ramamohanarao. On the decidability of the termination problem of active database systems. *Theor. Comput. Sci.*, 311(1-3):389–437, 2004.
3. 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.
4. A. J. Bonner and M. Kifer. Transaction logic programming. In *ICLP*, pages 257–279, 1993.
5. 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.
6. 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.
7. J. Chomicki, J. Lobo, and S. A. Naqvi. Conflict resolution using logic programming. *IEEE Trans. Knowl. Data Eng.*, 15(1):244–249, 2003.
8. M. Compton, C. A. Henson, H. Neuhaus, L. Lefort, and A. P. Sheth. A survey of the semantic specification of sensors. In *SSN09*, pages 17–32, 2009.
9. S. Costantini and G. D. Gasperis. Complex reactivity with preferences in rule-based agents. In *RuleML*, pages 167–181, 2012.
10. P. Fodor and M. Kifer. Tabling for transaction logic. In *ACMPPDP*, pages 199–208, 2010.
11. A. S. Gomes and J. J. Alferes. Transaction Logic with (complex) events. *Theory and Practice of Logic Programming, On-line Supplement*, To appear, 2014.
12. M. Hepp. Vehicle Sales Ontology. `http://www.heppnetz.de/ontologies/vso/ns`. Accessed: 2015-03-18.
13. R. A. Kowalski and F. Sadri. A logic-based framework for reactive systems. In *RuleML*, pages 1–15, 2012.
14. 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.
15. 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.
16. D. L. McGuinness, F. Van Harmelen, et al. OWL web ontology language overview. *W3C recommendation*, 10(2004-03):10, 2004.
17. R. Müller, U. Greiner, and E. Rahm. AgentWork: a workflow system supporting rule-based workflow adaptation. *Data Knowl. Eng.*, 51(2):223–256, 2004.
18. 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.
19. Y. Ren and J. Z. Pan. Optimising ontology stream reasoning with truth maintenance system. In *ACM CIKM*, pages 831–836, 2011.
20. A. P. Sheth, C. A. Henson, and S. S. Sahoo. Semantic sensor web. *IEEE Internet Computing*, 12(4):78–83, 2008.
21. C. Zaniolo. Active database rules with transaction-conscious stable-model semantics. In *DOOD*, pages 55–72, 1995.