TESLA: A Formally Defined Event Specification Language

Gianpaolo Cugola
Dip. di Elettronica e Informazione
Politecnico di Milano, Italy
cugola@elet.polimi.it

Alessandro Margara
Dip. di Elettronica e Informazione
Politecnico di Milano, Italy
margara@elet.polimi.it

ABSTRACT

The need for timely processing large amounts of information, flowing from the peripheral to the center of a system, is common to different application domains, and it has justified the development of several languages to describe how such information has to be processed. In this paper, we analyze such languages showing how most approaches lack the expressiveness required for the applications we target, or do not provide the precise semantics required to clearly state how the system should behave. Moving from these premises, we present TESLA, a complex event specification language. Each TESLA rule considers incoming data items as notifications of events and defines how certain patterns of events cause the occurrence of others, said to be "complex". TESLA has a simple syntax and a formal semantics, given in terms of a first order, metric temporal logic. It provides high expressiveness and flexibility in a rigorous framework, by offering content and temporal filters, negations, timers, aggregates, and fully customizable policies for event selection and consumption. The paper ends by showing how TESLA rules can be interpreted by a processing system, introducing an efficient event detection algorithm based on automata.

1. INTRODUCTION

Distributed applications often require large amount of information to be timely processed as it flows from the peripheral to the center of the system. As en example, environmental monitoring needs to process data coming from sensors deployed on field to acquire information about the observed world, detect anomalies, or predict disasters as soon as possible [9, 16]; financial applications require a constant analysis of stocks to detect trends [15]; fraud detection tools must observe continuous streams of credit card transactions to prevent frauds [26]; RFID-based inventory management performs a continuous analysis of registered data to track valid paths of shipments and to capture irregularities [28].

The traditional data processing model implemented by DataBase Management Systems (DBMSs) does not suit the timeliness requirements of these applications, as it needs information to be stored and indexed before processing. This led to the development of a number of systems specifically designed to process flows of information according to a set of deployed *rules*. Despite their common goal, these systems differ in a wide range of aspects, including architectures, data models, languages, and processing mechanisms [13]. After several years of research and development two models emerged: one comes from the database community as an evolution of active databases, and is usually called the *Data Stream Processing* (DSP) model [6], the other comes from the community working on message-oriented (in particular publish-subscribe) middleware and is usually called the *Complex Event Processing* (CEP) model [23].

In this paper we focus on the languages to express how incoming information has to be processed, and claim that none of those proposed so far is entirely adequate to fully support the needs that come from the aforementioned application scenarios. On one side, the data transformation approach adopted by the DSP model is not suited to recognize patterns of incoming items tied together by complex temporal relationships. On the other side, CEP languages are often oversimplified, providing only a small set of operators, insufficient to express a number of desirable patterns and the rules to combine incoming information to produce new knowledge. Even worse, the semantics of such languages is usually given only informally, which leads to ambiguities and makes it difficult compare the different proposals.

To overcome these limitations, in this paper we propose a new language called *TESLA (Trio-based Event Specification LAnguage)*. Each TESLA rule considers incoming data items as *notifications of events* and defines how *(complex) events* are defined from simpler ones. Despite an easy to use, clean syntax, with a limited number of different operators, TESLA is highly expressive and flexible, as it provides content and temporal constraints, parameterization, negations, sequences, aggregates, timers, and fully customizable policies for event selection and consumption. At the same time, not to incur in the semantic ambiguities affecting many existing languages, the TESLA semantics is formally specified by using *TRIO* [19, 22], a first order, metric temporal logic. Moreover, since an event detection language is pointless without a system interpreting it, we define how TESLA rules can be translated into automata to implement an efficient event detection engine.

DEBS'10, July 12-15, 2010, Cambridge, UK.
Copyright 2010 ACM 978-1-60558-927-5/10/07 ...$10.00.

The rest of the paper is organized as follows: in Section 2 we discuss the limitations we found in existing works and clarify our design goals; in Section 3 we present the TRIO logic. In Section 4 we introduce the TESLA event model and system architecture, and describe our language in details, providing the semantics for all valid operators, while in Section 5 we show how TESLA rules can be translated into automata for efficient pattern detection. Finally, we discuss related work in Section 6, providing some conclusive remarks in Section 7.

2. WHY A NEW LANGUAGE

To justify the need for a new event specification language we use an example, which illustrates the main limitation of existing approaches and shows the kind of expressiveness and flexibility we need.

2.1 A motivating example

Consider an environment monitoring application that processes information coming from a sensor network. Sensors notify their position, the temperature they measure, and the presence of smoke. Now, suppose a user has to be notified in case of fire. She has to teach the system to recognize such critical situation starting from the raw data measured by sensors. Depending on the environment, the application requirements, and the user preferences, the notion of fire can be defined in many different ways. Here we present four possible defining rules and we use them to illustrate some of the features an event processing language should provide.

i. Fire occurs when temperature higher than 45 degrees and some smoke are detected in the same area within 3 min. The fire notification has to embed the temperature actually measured.

ii. Fire occurs when temperature higher than 45 degrees is detected and it did not rain in the last hour.

iii. Fire occurs when there is smoke and the average temperature in the last 3 min. is higher than 45 degrees.

iv. Fire occurs when at least 10 temperature readings with increasing values and some smoke are detected within 3 min. The fire notification has to embed the average temperature of the increasing sequence.

First of all, they *select* relevant notifications from the history of all received ones according to a set of constraints. Two kinds of constraints are used: the first one selects elements on the basis of the values they carry, either to choose single notifications (e.g. those about temperature higher than 45 degrees) or to choose a set of related notifications (e.g. those coming from the same area). The latter case is usually called *parameterization*. The second kind of selection constraints operates on the timing relationships among notifications (e.g. selecting only those generated within 3 minutes) and allows to capture *sequences* of events (e.g., high temperature followed by smoke or vice-versa).

Beside selection, rule (*ii*) introduces *negation*, by requiring an event not to occur in a given interval. Similarly, rule (*iii*) introduces *aggregates*. In particular, it defines a function (average) to be applied to a specified set of values (temperature readings in the last 3 minutes) to calculate the value to be carried by the complex event. Rule (*iv*) is interesting as it combines the two kinds of selection constraints (those on values and those on timing) to define an *iteration* that selects those elements that bring growing temperature readings. Such kind of rules is common in various domains, like in financial applications for stock monitoring, where they are used to promptly detect relevant market trends.

Finally, when the desired combination of notifications has been detected, rules have to specify which notification to create (e.g. fire) and they have to define its inner structure (e.g. the notification has to embed a temperature reading).

If we look at existing languages, we notice how they differ in the set of operators they provide, which, in turn, determine their expressiveness. As an example, not all languages provide parameterization or iterations [13]. We think that a language for CEP should be able to express all the constructs above: selection, parameterization, negations, aggregates, sequences, and iterations. Additionally, it should be simple and unambiguous, i.e., it should be easy to write rules having a clear and precise semantics.

2.2 Limitations of existing languages

As mentioned in Section 1, most of the languages recently proposed in the literature to express rules like those above can be classified in two groups: *Data Stream Processing* (DSP) and *Complex Event Processing* (CEP) languages.

A notable representative of the first class is CQL [5], created within the Stream project [4]. Information flowing into the system is organized in homogeneous *streams* composed by timestamped tuples, all sharing the same schema. Each CQL rule (*query* in CQL jargon) takes one or more streams as inputs and produces one output stream. Queries are defined using three types of operators. *Stream-to-Relation* (S2R) operators select a portion of a stream to implicitly create a traditional database table. They are also known as *windows* and operate either on time (e.g. selecting tuples received in the last 5 minutes every time a new tuple arrives) or on the number of elements (e.g. selecting the last 10 tuples every time a new one arrives). *Relation-to-Relation* (R2R) operators are mainly standard SQL operators. *Relation-to-Stream* (R2S) operators generate new streams from tables, after data manipulation. Each CQL query is composed by a S2R operator, one or more R2R operators, and one R2S operator. A key aspect of languages like CQL is forgetting the ordering between elements when moving from streams to relations. No explicit sequencing operators are provided and timestamps, if not artificially introduced as part of the tuples' schema, cannot be referenced during R2R processing. As a result, rules like the fourth of our example, which involve a single sequence of information items, are very hard to write in CQL, or even impossible if tuples do not include explicit references to time. More in general, all queries that select elements from the history of received items using timing constraints do not find a natural support in DSP languages. In fact, such languages are mainly designed to isolate portions of input flows and to perform traditional database processing within the bound of portions, but they show severe limitations when it comes to recognize complex patterns of relevant items among received ones [13].

It is worth mentioning that more complex DSP languages, which extend the expressive power of CQL, exist: a remarkable example is represented by ESL [7], a Touring complete DSP language. With these languages detecting complex patterns becomes possible but it remains difficult to support and far from natural, since the general schema of the language remains that of CQL. Moreover, such languages, with their more complex and less common syntax, suffer some limitations typical of CEP languages (see below), mainly in terms of lack of a rigorous semantics.

CEP languages present a different processing model w.r.t. DSP ones: a model specifically designed for the detection of complex temporal patterns of incoming information. Indeed, they consider single information items flowing into the system as representations of events occurred in the observed world, and they define how complex events result from simple ones. This model is better suited to naturally express the rules in our example; however, existing languages present two problems.

First, most of them are extremely simple and present serious limitations in terms of expressiveness; for example, some languages force sequences to capture only adjacent events [8], making it impossible to express rules like *i* and *iii* above. Often negations are not allowed [21, 8], or cannot be expressed through timing constraints [3], like in rule *ii.* above. Other widespread limitations are the lack of a full-fledged iteration operator (Kleene+ [20]), that could allow rules to capture a priori unbounded repetitions of events (like in rule *iv.*), and the lack of processing capabilities for computing aggregates.

Second, a formal definition of operators is hardly ever provided, leading to semantic ambiguities. As an example of the latter problem, consider rule (i) above. It defines a simple conjunction of two items, which is supported by almost all existing CEP languages and apparently poses no issues. However, if you consider the sequence of events depicted in

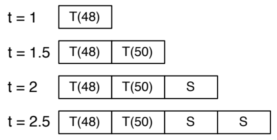

Figure 1: Monitored events history

Figure 1 some problems emerge. Suppose that all events come from the same area and `T(x)` represents the event notification *temperature* $= x$, while `S` represents the presence of smoke. Initially, an event `T(48)` occurs, followed by another one, `T(50)`. When the event `S` is received the first issue arises: how many fire notification should the system generate? Two (one for each pair `<T(48),S>` and `<T(50),S>`) or just one? And, if we choose to deliver only one fire notification which value of temperature should it embed? Rule (i), expressed in natural language, is ambiguous, but unfortunately most CEP languages are not more precise. We call this problem of deciding how to combine events when multiple choices are available the *event selection problem.* Now suppose that the system reacts by producing two notifications, what happens when another event `S` occurs, like at $t = 2.5$? In some sense the two events `T` have already been "used": should they be considered again or not? We refer to this problem of deciding whether event notifications become invalid for further processing after being considered as the *event consumption problem.*

It is worth noting that apart from a problem of lack of precise semantics, the case above also evidence a problem of expressiveness, since different applications may require different event selection and consumption policies. Not only, sometimes a single application may need different policies for different rules. Alert notifications, for example, are usually required only once, even when they can be generated by multiple combinations of events; financial analysts, on the contrary, may be interested in all possible combinations of stock events. For this reason we think that selection and consumption policies should be accessible and customizable within the rule specification language, thus allowing the rule manager to choose the most appropriate one for each application and each rule within the same application. On the contrary, existing CEP languages (e.g. [21, 25, 8]), define, often implicitly and only through an actual reference implementation, a unique selection and consumption policy and do not allow users to change them. Some remarkable exceptions exist: in particular some languages designed for Active DBMSs allow users to choose selection and consumption policies as part of the definition of a rule [14]. However, also in these cases, only a few predefined choices are possible and users cannot tailor them to their needs.

2.3 TESLA design goals

Moving from these considerations we designed TESLA to overcome the limitations found in other languages, providing a high degree of expressiveness to users while keeping a simple syntax with a rigorously defined semantics. In particular, TESLA provides selection operators, parameterization, negations, aggregates, sequences, iterations, and fully customizable event selection and consumption policies, while also supporting reactive and periodic rule evaluation within a common syntax. At the same time, we provide a formal semantics for TESLA using a first order temporal logic. The remainder of this paper discusses these issues in details.

3. TRIO: A BRIEF OVERVIEW

TRIO [19, 22] is a first order logical language augmented with temporal operators, which enable to express properties whose value change over time. TRIO temporal operators, unlike those of conventional temporal logic, provide a metric on time: they express the length of time intervals quantitatively. The meaning of a TRIO formula is not absolute: it is given with respect to a current time instant that is left implicit in the formula. These two properties make TRIO well suited to naturally specify events and their occurrence.

The alphabet of TRIO includes sets of names for variables, functions, and predicates, plus a fixed set of operators, including propositional symbols ($\wedge$, $\neg$), quantifiers ($\forall$), and the temporal operators *Futr* and *Past.* TRIO is a typed language: variables, functions, and predicates have their own type, which determines the set of values they can assume, return, or take as arguments. Among the allowed types there is a distinguished one, required to be numeric in nature: the *temporal domain.* TRIO distinguishes between *time-dependent* variables (resp. functions and predicates), whose value may change with time, and *time-independent* ones, whose value is independent from time.

The syntax of TRIO is recursively defined as follows:

- Every variable is a term
- Every n-ary function applied to n terms is a term

If a term is a variable, then its type is the type of the variable; if the term results from the application of a function, then its type is the range of the function.

- Every n-ary predicate applied to n terms of the appropriate types is a formula
- If A and B are formulas, $\neg A$ and $A \wedge B$ are formulas
- If A is a formula and x is a time-independent variable, $\forall x\, A$ is a formula
- If A is a formula and t is a term of the temporal type, then *Futr(A, t)* and *Past(A, t)* are formulas

Abbreviations for the propositional operators $\vee$, $\rightarrow$, $true$, $false$, $\leftrightarrow$ and for the derived existential quantifier $\exists$ are defined as usual.

The semantics of TRIO is formally defined in [19, 22], here we focus on the two temporal operators *Futr* and *Past*. In particular, formula *Past(A, t)* (resp. *Futr(A, t)*) is true if A holds t time units in the past (resp. future) w.r.t. the current time, which is left implicit in the formula.

A lot of temporal operators have been derived from *Futr* and *Past*. In the following we will make use of two of them: always (*Alw(A)*) and within the past ($WithinP(A, t_1, t_2)$), where A is a formula and t_1, t_2 terms of temporal domain; they are defined as follows:

$$\begin{aligned} Alw(A) = A \wedge\ & \forall t(t > 0 \rightarrow Futr(A, t)) \\ \wedge\ & \forall t(t > 0 \rightarrow Past(A, t)) \end{aligned}$$

$$WithinP(A, t_1, t_2) = \exists x(t_1 \leq x \leq t_1 + t_2 \wedge Past(A, x))$$

Before concluding this brief overview of TRIO, it is worth noting how such logic only requires the temporal domains to be numeric, and does not dictate further constraints: it can be, for example, either discrete or continuous. TESLA keeps the same property, enabling designers to choose the more suitable temporal domain according to their needs.

4. LANGUAGE DEFINITION

In this Section we present the TESLA language in details; in particular we describe the TESLA event and rule models, then we define the general structure of rules and we show how they can be translated into TRIO formulas that precisely define their semantics. Finally, we present all the possible patterns of events that can be captured through TESLA rules, describing their use through various examples.

4.1 TESLA event and rule model

In TESLA we assume events, i.e. things of interest, to occur instantaneously at some points in time. In order to be understood and processed, events have to be observed by *sources* (see Figure 2), which encode them in *event notifications* (or simply *events*). We assume that each event notification has an associated *type*, which defines the number, order, names, and types of the *attributes* that build the notification. Notifications have also a timestamp, which represents the occurrence time of the event they encode[1]. As an example, an event can be the temperature reading in a room at a specific time. A sensor may observe this event and generate the following notification:

$$Temp@10(Room = \text{"}Room1\text{"},\ Value = 24.5)$$

Where *Temp* represents the type of the notification and 10 is the timestamp. The *Temp* type defines two attributes: a string that identifies the room in which the temperature was measured, and the actual measure (a float). As attributes are ordered, notifications may also omit their names. TESLA

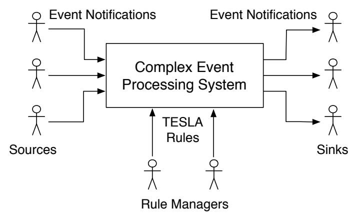

Figure 2: TESLA Reference Architecture

rules define complex events from simpler ones. The latter can be observed directly by sources or they can be complex events defined by other rules (in the following we will refer to this mechanism using the term "hierarchies of events"). *Sinks* subscribe to events and receive notifications as soon as their requests are met. Subscriptions are as simple as in traditional publish-subscribe languages and include the type of relevant events together with a filter over the content of events attributes, like in the following example:

$$Subscribe(Temp,\ Room = \text{"}Room1\text{"} and\ Value > 20)$$

Subscriptions may refer to simple events, i.e., those directly observed by sources, or to complex ones, i.e., those derived through TESLA rules. The resulting architecture distinguishes between *Rule managers* who define TESLA rules, and sinks who subscribe to events. As we will argument later, we fell that keeping the two roles separate helps building reusable rules.

4.2 Structure of the rules

Each TESLA rule has the following general structure:

$$\begin{array}{ll} define & CE(Att_1 : Type_1, ..., Att_n : Type_n) \\ from & Pattern \\ where & Att_1 = f_1, .., Att_n = f_n \\ consuming & e_1, .., e_n \end{array}$$

Intuitively the first two lines define a (complex) event from its constituents, specifying its structure — $CE(Att_1$: $Type_1$, ..., Att_n: $Type_n)$ — and the pattern of simpler events that lead to the complex one. The *where* clause defines the actual values for the attributes Att_1, .., Att_n of the new event using a set of functions f_1, .., f_n, which may depend on the arguments defined in *Pattern*. Finally, the optional *consuming* clause defines the set of events that have to be invalidated for further firing of the same rule.

4.3 Semantics of rules

Each TESLA rule associates a patter of events p with the complex event e it represents. Accordingly, to provide a precise semantics for rules we need to express the logical equality between the validity of p at a given instant and the occurrence of e at the same instant.

[1]The issue of who timestamp events, e.g., the sources or the CEP system, and the need of ad-hoc mechanisms to cope with out-of-order arrivals, have been discussed in the past [27] and are out of the scope of this paper. These are system issues that do not impact the TESLA language, which is meant to process events in timestamp order, whoever sets it.

In TRIO we can express the occurrence of an event using a time-dependent predicate, which becomes true when the event occurs. On the other hand, as we will prove through various examples, in TESLA several events of the same type, possibly with the same attribute values, may occur at the same time. To differentiate them we have to introduce the concept of *label*: it is a unique global identifier for event notifications[2]. While we can safely assume that events coming from external sources already have their own unique label, we have to define the label of those complex events defined through TESLA rules. To do so, we observe that a given set of events s can satisfy a rule r at most once (we will prove this later through the *uniqueness of selection* theorem). We leverage this property assuming that a time-independent label generation function *lab* is defined, which returns new labels taking two arguments: a unique rule identifier, and a set of labels (those of the set of notifications s that satisfied the rule leading to the new event). For labels to uniquely identify complex events, *lab* has to be injective:

$$\forall\, r_1, s_1, r_2, s_2 \\ ((lab(r_1, s_1) = lab(r_2, s_2)) \;\leftrightarrow\; (r_1 = r_2 \wedge s_1 = s_2))$$

Once labels have been introduced we can use them to formally define the occurence of events through the predicate $Occurs(Type, Label)$, which is true at the time when the event of type *Type* having label *Label* occurs, and false in every other instant. The fact that labels are unique and that a given event notification occurs only once, is formally captured through the following formulas:

$$Alw\ \forall e_1, e_2 \in E, \forall l \in L \\ ((Occurs(e_1, l) \wedge Occurs(e_2, l)) \rightarrow e_1 = e_2)$$

$$Alw\ \forall e_1, e_2 \in E, \forall l \in L, \forall t > 0\ (Occurs(e_1, l) \rightarrow \\ (\neg Past(Occurs(e_2, l), t) \wedge \neg Futr(Occurs(e_2, l), t)))$$

Where L is the set of all valid labels and E is the set of all event types. The first formula states that, in a given instant of time, there cannot be two notifications having the same label and different types. The second formula guarantees that, if an event with label l occurs at time t, no other events having the same label can occur at different times.

To reason about the content of event notifications we introduce the domain N of all valid names for event attributes. Since attributes can have different types (e.g. string, int, float) for each type X we define a time-independent function $attVal_X : L \times N \rightarrow X$: given a label l and the name of an attribute n it returns the value of the attribute in the event notification having label l. For simplicity, in the following examples we assume all attributes share a common domain *V*, so that we can use a single function generically called $attVal$ to associate attribute names to their values. For the same reason, from now on we will omit types from the *define* clause of our rules.

Using the elements defined above, a generic TESLA rule, in the form shown in Section 4.2, is translated into the following TRIO formula (we omit the translation of the *consuming* clause, as it will be addressed later):

$$Alw\ \forall l_1, .., l_m \in L, \forall n_1, .., n_n \in N \\ ((Occurs(CE, lab(r, \{l_1, .., l_m\})) \;\leftrightarrow\; Pattern) \wedge \\ (Pattern \;\rightarrow\; attVal(lab(r, \{l_1, .., l_m\}), n_1) = f_1) \wedge \\ (Pattern \;\rightarrow\; attVal(lab(r, \{l_1, .., l_m\}), n_n) = f_n))$$

[2]Notice how labels are only required to translate TESLA rules into TRIO formulas, while they do not appear inside the TESLA language, which operates at a higher level of abstraction.

Where $\mathbb{N}$ is the set of all natural numbers and $r \in \mathbb{N}$ represents a unique identifier for the translated TESLA rule, while $l_1, .., l_m \in L$ are the labels of all event notifications captured by *Pattern* and $n_1, .., n_n$ are the attribute names for the event type CE. The TRIO formula asserts that, in every instant of time in which *Pattern* becomes true, an event notification of type CE occurs, whose label is defined by the *lab* function applied to the number of the rule r and the set of labels of all event notifications captured by the pattern (and viceversa). The formula also specifies the values for the new event's attributes using the functions defined in the corresponding TESLA rule.

4.4 Valid patterns

In the discussion so far we left unspecified the inner structure of a pattern; we now introduce all operators used in TESLA to define valid patterns.

Event occurrence. The simplest type of event pattern represents the occurrence of a single event, possibly satisfying a set of constraints. As an example consider the following requirement: *generate an overflow notification when the level of water in a river overcomes 20; the notification must include the name of the river.* This can be translated into the following TESLA rule:

$$\begin{array}{ll} define & Overflow(Name) \\ from & WaterLevel(Level > 20)\ as\ WL \\ where & Name = WL.Name \end{array}$$

As the example shows, TESLA puts the constraints over the content of an event into parentheses after the type of the event. In this case a single constraint is needed but TESLA accepts conjunctions of constraints as well. To access the field of an event TESLA uses a *dot* notation *event-name.attribute-name*. A name is associated to an event using the *as* keyword. When only an event of a given type is present in the pattern, like in our example, it is also possible to omit the *as* clause and use the event type as the name of the event. We can also split the type of the selected event from the constraints on its attributes; for example the *from* clause of the previous formula could be written as *WaterLevel() as WL and WL.Level>20*.

Translating in TRIO rules involving single events is easy. Notice that to keep formulas more compact, here and in the following we assume all free variables to be universally quantified at the outermost level and all formulas to start with the *Alw* operator, which we omit. Using these conventions we provide the translation of a general rule selecting a single event:

$$\begin{array}{ll} define & CE(Att_1, .., Att_n) \\ from & SE(Att_x\ op\ Val_x) \\ where & Att_1 = f_1, .., Att_n = f_n \quad \triangleq \end{array}$$

$$(Occurs(CE, lab(r, \{l_1\})) \;\leftrightarrow \\ (Occurs(SE, l_1)\ \wedge attVal(l_1, Att_x)\ op\ Val_x)) \wedge \\ (Occurs(SE, l_1)\ \wedge attVal(l_1, Att_x)\ op\ Val_x)\ \rightarrow \\ (attVal(lab(r, \{l_1\}), Att_1) = f_1\ \wedge .. \wedge \\ attVal(lab(r, \{l_n\}), Att_n) = f_n)$$

To capture the meaning of this formula, consider the situation in which two different events of type *SE*, A and B, occur at the same time. As the two events necessarily have different labels, l_A and l_B, to satisfy the formula above two different *CE* notifications must be generated, having labels $lab(r, \{l_A\})$ and $lab(r, \{l_B\})$. The two notifications are guaranteed to be distinct (i.e. to represent different event occurrences), as the *lab* function is injective.

Event composition. To capture the occurrence of several, related events, TESLA provides three *event composition* operators: *each-within*, *first-within*, and *last-within*. All event composition operators bind the occurrence of an event to the occurrence of another one, introducing a detection window. Using the terminology introduced in Section 2 we can say that they differ from each other according to the *selection policy* they define. As an example, consider the following rules:

$$\begin{array}{ll}
define & Fire(Val) \\
from & Smoke()\ and \\
 & each\ Temp(Val > 45)\ within\ 5min\ from\ Smoke \\
where & Val = Temp.Val
\end{array}$$

$$\begin{array}{ll}
define & Fire(Val) \\
from & Smoke()\ and \\
 & last\ Temp(Val > 45)\ within\ 5min\ from\ Smoke \\
where & Val = Temp.Val
\end{array}$$

Both rules define the *Fire* event from *Smoke* and *Temp*. The first rule leads to notify a *Fire* event for each *Temp* event higher than 45 occurred within 5 minutes from the smoke detection (if any). We say that the *each-within* operator defines a *multiple selection* policy as it uses any available *Temp* notification in a given time window. The second rule, instead, creates a single *Fire* notification by selecting only the latest *Temp* event higher than 45 occurred within 5 minutes from the smoke detection. The *first-within* operator exhibits a similar behavior, by selecting the first event in the specified time interval. We say that the *last-within* and *first-within* operators define a *single selection* policy as they force the selection of at most one element. The formal definitions of all event composition operators are shown below (for space reasons we omit the attribute constraints and assignments):

$$\begin{array}{l}
define\ CE\ from\ A\ and\ each\ B\ within\ x\ from\ A \quad \triangleq \\
Occurs(CE, lab(r, \{l_0, l_1\}) \ \leftrightarrow \\
(Occurs(A, l_0)\ \wedge\ WithinP(Occurs(B, l_1), Time(l_0), x))
\end{array}$$

$$\begin{array}{l}
define\ CE\ from\ A\ and\ last\ B\ within\ x\ from\ A \quad \triangleq \\
Occurs(CE, lab(r, \{l_0, l_1\}) \ \leftrightarrow \\
(Occurs(A, l_0)\ \wedge\ WithinP(Occurs(B, l_1), Time(l_0), x) \\
\ \wedge\ \neg\exists t \in (Time(l_1), Time(l_0)]\ Past(Occurs(B, l_2), t) \\
\ \wedge\ (\neg Past(Occurs(B, l_3), Time(l_1))\ \wedge\ l_3 > l_1))
\end{array}$$

$$\begin{array}{l}
define\ CE\ from\ A\ and\ first\ B\ within\ x\ from\ A \quad \triangleq \\
Occurs(CE, lab(r, \{l_0, l_1\}) \ \leftrightarrow \\
(Occurs(A, l_0)\ \wedge\ WithinP(Occurs(B, l_1), Time(l_0), x) \\
\ \wedge\ \neg\exists t \in [x, Time(l_1))\ Past(Occurs(B, l_2), t) \\
\ \wedge\ (\neg Past(Occurs(B, l_3), Time(l_1))\ \wedge\ l_3 < l_1))
\end{array}$$

Here we used the time dependent function *Time*, which takes a label as argument and returns the occurrence time of the event having that label with respect to the current time, left implicit. Using such function, the definition of the *each-within* operator is straightforward: it only adopts the *WithinP* temporal operator, binding *CE* notifications to each B event found in the valid time interval. The definition of *last-within* and *first-within* operators introduce an additional constraint imposing no B events to occur after (resp. before) the selected one in the defined time interval. Notice that to provide a unique order between events in presence of simultaneous occurrences, we assume an ordering between labels.

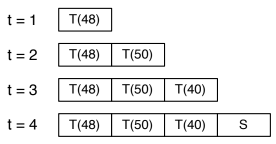

Figure 3: A possible history of event occurrences

To better understand the difference between single and multiple selection operators, consider again the two TESLA rules described above. We show an example of event history in Figure 3. `S` represents a *Smoke* event, while `T(n)` represents an event of type *Temp* (n being the measured temperature). When at `t=4` a *Smoke* event is detected, the rule that uses the *each-within* operator combines it with every *Temp* event greater than 45, which result in two different *Fire* notifications. On the contrary, the rule that uses the *last-within* operator results in a single *Fire* notification, that combining the *Smoke* event with the latest *Temp* event greater than 45 (i.e. the one received at time `t=2`).

TESLA also offers a generalized version of the *first-within* and *last-within* operators, called *k-first-within* and *k-last-within*. They can be used to capture the second, third, etc. event occurrence from the beginning (resp. end) of a specified interval. The formal definition of the semantics of these operators is omitted as it can be easily derived from the definition of the basic *single selection* operators.

As a final remark, notice that TESLA allows the definition of rules that combine multiple composition operators; they can be connected in series, defining chains of event occurrences; or in parallel, allowing more event occurrences to be bound to a single one. The rule below shows both options:

$$\begin{array}{ll}
define & D() \\
from & A()\ and\ each\ B()\ within\ 5min\ from\ A\ and \\
 & last\ C()\ within\ 3min\ from\ A\ and \\
 & last\ D()\ within\ 6min\ from\ B\ and \\
 & first\ E()\ within\ 2min\ from\ D\ and \\
 & E\ within\ 8min\ from\ A
\end{array}$$

Notice, in particular, the use of the *within* operator in the last line, which introduces an additional timing constraint for an already defined event E.

Parameterization. In Section 2 we introduced parameterization as one of the required features in an event specification language. As an example, consider again the rule about *Fire* notifications as defined through the *each-within* operator in the previous section. Knowing that a *Smoke* and a *Temp* events occurred within 5 minutes may be meaningless if we don't know whether the two events come from the same area. To express similar relationships, TESLA introduces parameters through the *$* operator. Suppose that both *Temp* and *Smoke* events have an attribute called *Area*:

the following rule exemplifies the use of parameters to force the two events of interest to come from the same area:

$$\begin{array}{ll} define & Fire(Val) \\ from & Smoke(Area = \$x)\ and \\ & each\ Temp(Val > 45\ and\ Area = \$x) \\ & within\ 5min\ from\ Smoke \\ where & Val = Temp.Val \end{array}$$

Defining the semantics of parameters in TRIO is straightforward, we simply add one or more constraints on the values of attributes. For example, the rule above requires $attVal(l_0, Area) = attVal(l_1, Area)$ where l_0, l_1 are the labels of the selected *Smoke* and *Temp* events.

Theorem: Uniqueness of selection. As mentioned at the very beginning of this section, all TRIO formulas used so far are correct under the assumption that a set of events can be selected by a given rule only once. We call this assumption: *uniqueness of selection*. Together with the adoption of an injective function to define labels, it makes it impossible to generate different events sharing the same label. As the operators described so far define all the event selection strategies allowed in TESLA, we are now ready to prove that the uniqueness of selection assumption is satisfied[3].

Proof. All TESLA rules joins the occurrence of a (complex) event to the occurrence of a pattern of (simpler) events, one of which must occur at the same time of the complex one, while the others occur in the past. This guarantees that a given rule r is satisfied by a set of events E only once, at time t. In fact, future evaluations of the same rule r at time $t_1 > t$ would require at least a new event e to occur at evaluation time t_1. On the other hand, since rules only refer to current and past events, e cannot be part of E, so the pattern of events satisfying r at t_1 must differ from E.

Timers. Several application domains require rules to be evaluated periodically. TESLA supports periodic rules using special events called *timers*. As an example, we may require a rule to be evaluated only at 9.00 of Friday by using $Timer(H = 9, M = 00, D = Friday)$ in its *from* clause. This approach keeps the syntax of the language simple, using a uniform approach for both reactive and periodic rules, and it does not change the semantics of rules (without impacting the uniqueness of selection property).

Negation. Applications often need to reason not only about the events occurred, but also about those that did not occur. As an example, we could detect a fire when *Temp* and *Smoke* events are detected in the same area in absence of *Rain*. To deal with similar cases, TESLA introduces the *not* operator, which defines an interval of time in which a given event must not occur. Such interval can be defined in two ways: using two events as the interval bounds or using a single event together with the duration of the interval. The following rules introduce the two cases:

$$\begin{array}{ll} define & Fire(Val) \\ from & Smoke(Area = \$x)\ and \\ & each\ Temp(Val > 45\ and\ Area = \$x) \\ & within\ 5min\ from\ Smoke\ and \\ & not\ Rain(Area = \$x)\ between\ Temp\ and\ Smoke \\ where & Val = Temp.Val \end{array}$$

$$\begin{array}{ll} define & Fire(Val) \\ from & Smoke(Area = \$x)\ and \\ & each\ Temp(Val > 45\ and\ Area = \$x) \\ & within\ 5min\ from\ Smoke\ and \\ & not\ Rain(Area = \$x)\ within\ 5min\ from\ Smoke \\ where & Val = Temp.Val \end{array}$$

The semantics of these two forms that the *not* operator may assume is defined below. Notice that the first syntax is allowed only when the relative order between the two events defining the interval of time is known. This happens when both events belong to a common chain of event occurrences.

$$\begin{array}{l} define\ D\ from\ A\ and\ each\ B\ within\ x\ from\ A \\ and\ not\ C\ between\ B\ and\ A\ \triangleq \\ Occurs(D, lab(r, l_0, l_1))\ \leftrightarrow \\ (Occurs(A, l_0)\ \wedge\ WithinP(Occurs(B, l_1), x)\ \wedge \\ \neg\exists t \in [Time(l_0), Time(l_1))\ (Past(Occurs(C, l_2)), t)) \end{array}$$

$$\begin{array}{l} C\ when\ A\ and\ not\ B\ within\ x\ from\ A\ \triangleq \\ Occurs(c, lab(r, l_0))\ \leftrightarrow\ (Occurs(A, l_0)\ \wedge \\ \neg\exists t \in [Time(l_0), Time(l_0) + x)\ (Past(Occurs(B, l_1)), t)) \end{array}$$

Event consumption. As discussed in Section 2, one of the main limitations of existing CEP languages is the lack of customizable event selection and consumption policies. While the *xxx-within* operators introduced so far allow users to adopt the preferred event selection policy, TESLA uses the *consuming* clause to deal with event consumption, allowing users to specify the selected events that have to become invalid for future detections (by the same rule). As an example, consider the following rule:

$$\begin{array}{ll} define & Fire(Val) \\ from & Smoke()\ and\ each\ Temp(Val > 45) \\ & within\ 5min\ from\ Smoke \\ where & Val = Temp.Value \\ consuming & Temp \end{array}$$

It consumes all selected *Temp* events, so a new *Smoke* would not fire the rule until a new *Temp* (followed by a further *Smoke*) happens.

To define the semantics of rules that include the *consuming* clause, we introduce a new time dependent TRIO predicate called *Consumed*. Given a rule identifier r and a label l, *Consumed(r,l)* remains false until the event with label l is consumed by the rule r, and it holds true after event consumption. Formally, we ask the *Consumed* operator to satisfy the following properties:

$$\begin{array}{l} Alw\ \forall l \in L, \forall r \in \mathbb{N} \\ (Consumed(r, l) \rightarrow \forall\ t > 0, Futr(Consumed(r, l), t)) \end{array}$$

$$\begin{array}{l} Alw\ \forall l \in L, \forall e, r \in \mathbb{N}, \forall S \\ ((\neg\exists t > 0\ (Past(Occurs(e, lab(r, S), t)))\ \wedge\ l \in S) \\ \rightarrow\ \neg Consumed(r, l)) \end{array}$$

The former guarantees that once an event with label l has been consumed by a rule r, it always remains consumed in the future (w.r.t. r). The latter guarantees that if an event e has not (yet) been captured by a rule r (i.e. it is not

[3]In the remainder of the paper we present new operators that extend the expressiveness of TESLA. None of them, however, introduces new selection mechanisms: at most they add new constraints, reducing the set of complex events which may occur. Accordingly, none of them influences the uniqueness of selection.

part of a set of labels S selected by the rule in the past), it cannot be considered as consumed w.r.t. r. To show how the *Consumed* predicate can be used to formalize the semantics of rules including a *consuming* clause, we provide the translation of the TESLA rule above in TRIO:

$$Occurs(Fire, lab(r, \{l_1, l_2\})) \leftrightarrow (Occurs(Smoke, l_1) \\ \wedge\ WithinP(Occurs(Temp, l_2), Time(l_1), 5) \wedge \\ attVal(l_2, Value) > 45 \ \wedge\ \neg Consumed(r, l_2))$$

$$(Occurs(Smoke, l_1) \wedge \\ WithinP(Occurs(Temp, l_2), Time(l_1), 5) \wedge \\ attVal(l_2, Value) > 45 \ \wedge\ \neg Consumed(r, l_2)) \ \rightarrow \\ \forall t > 0\ Futr(Consumed(r, l_2), t)$$

The first formula differs from the standard translation used so far as it requires all events appearing in the *consuming* clause (i.e. *Temp*) not to be consumed at evaluation time. The second specifies that, if at time t the pattern is satisfied selecting the *Temp* event with label l_2, such event has to be considered consumed in the future. Using the *consuming* clause together with single selection operators it is possible to define rules that always capture a specific event (e.g. the first, or the last) among non consumed ones only. To the best of our knowledge no other languages based on patterns are expressive enough to define similar rules. Also notice that event consumption is valid only within a rule. We think that this semantics is the most natural for a CEP system, in which multiple rules use the same event notifications independently, possibly with different aims. Defining a *global consume* operator would be straightforward, however we preferred not to introduce it as we think that it would have made rule definition and management a harder tasks.

Aggregates. The importance of aggregates has been discussed in Section 2. Aggregates apply a function to a set of values S to generate a new value v. TESLA allows v to be used wherever a value is allowed; in particular v can be assigned to an attribute of the complex event being defined or it can be used inside the constraints that select the relevant events. TESLA aggregates capture values from events in a specified time interval. As for negations, time intervals can be specified through the occurrence of two events or through a single occurrence plus the duration of the interval. The following examples show the two cases:

```
define    AvgTemp(Val)
from      Timer(M%5 == 0)
where     Val = Avg(Temp().Value)
          within 5min from Timer
```

```
define    HighVal(Name, Val)
from      Stock(Name = $y, Val = $x) and
          last Opening() within 1day from Stock and
          $x > Avg(Stock(Name = $y).Val)
          between Opening and Stock
where     Val = S.Val,  Name = S.Name
```

The first rule is evaluated periodically (every 5 minutes) and generates an *AvgTemp* notification embedding the average value of temperature readings in the last 5 minutes. The second rule generates a new *HighVal* notification when the value of a *Stock* overcomes the average value computed from the last *Opening*.

The following TRIO formula provides the semantics for a generic aggregate function *Fun* applied to the attribute *Val* of events X occurred between A and B. The formula defines a *Set* including all values of attribute *Val* in events of type X occurred between A and B. Formally *Set* is defined as a set of label-value couples, in such a way that the same value coming from n different events is considered n times. The function *Fun* uses values in *Set* to produce the result.

$$Fun(X.Val)\ between\ A\ and\ B\ =\ Y \ \triangleq \\ \forall\ Set\ (\forall x(x \in Set\ \leftrightarrow \exists l \in L(x =< l, attVal(l, Val) > \\ \wedge\ withinP(Occurs(X, l), Time(B), Time(A)))) \\ \rightarrow\ Fun(Set) = Y)$$

Hierarchies of events. Most languages for CEP do not provide a separation between event definition rules and users subscriptions; subscribers deploy so called *composite subscriptions* that embed the pattern of events they are interested in. While this approach has no impact on the expressiveness of the language, in our opinion it makes definition of rules more difficult, as it does not allow complex events to be reused. On the contrary, TESLA enables complex events defined through a rule to be used inside other rules; this way users may easily create very expressive *hierarchies of events*. We think that this approach better fits the nature of many applications in which sources provide a high volume of low level information items which need to be filtered and combined at different logical levels to produce results for the final users. As an example, consider a weather forecast application: sensors provide information about their locations and the temperature they read. As a first step, a rule manager may define an average temperature event that is generated every 5 minutes and include all the readings coming from a given area. Then, these events can be combined into patterns defining temperature trends. Finally, trends can be used together with other data (e.g. about wind) to provide weather forecast. Notice that a correct definition of hierarchies require no circular dependencies to exist between rules. This constraint cannot be verified directly inside the TESLA language, as it requires knowledge that is outside the scope of single rules, however it can be easily checked by a CEP system at rule deploy time.

Iterations. Several existing languages for CEP define adhoc operators to express bounded or unbounded iterations of patterns (Kleen closures) [20, 8, 21]. Such operators may be useful to detect trends: as an example they enable patterns involving continuously increasing values for a stock. A precise definition of iteration operators has to take into account a number of aspects: selection and consumption policies, minimum and maximum number of iterations, relations between attribute values, termination criteria, etc., which complicate the language, both syntactically and semantically. On the other hand TESLA, with its ability to define hierarchies of events and to adopt different selection and consumption policies for different rules, is expressive enough to not require special operators to capture iterations. A great advantage in term of elegance and simplicity. As an example, suppose we want to capture every iteration of events of type A, where the attribute *Val* never decreases, to notify an event B that contains the number of A that are part of the iteration. This is captured by the following TESLA rules:

```
define    RepA(Times, Val)
from      A()
where     Times = 1 and Val = A.Val
```

```
define      RepA(Times, Val)
from        A($x) and last RepA(Val ≤ $x) within 3min
            from A
where       Times = RepA.Times + 1 and Val = $x
consuming   RepA

define      B(Times)
from        RepA()
where       Times = RepA.Times
```

Notice how TESLA provides a high degree of flexibility. For example, it is easy to modify the event selection and consumption policies or content and timing constraints in the above rules to change the events actually captured, e.g., to capture only the longest iteration occurred.

5. EVENT DETECTION AUTOMATA

Processing of TESLA rules involves detecting patterns of interest from the history of events and generating the corresponding (complex) events. Different techniques could be used to achieve these goals: here we present an efficient detection algorithm based on automata, which evaluates events incrementally, as they occur.

5.1 Ordering in TESLA rules

TESLA rules define a *partial order* among the events to be selected. Consider for example the following rule R:

```
define   CE()
from     A(Va > 1)
         and each B(Vb > 2) within 2 min from A
         and each C(Vc < 3) within 4 min from A
         and each D(Vd = 5) within 4 min from B
         and D within 5 min from C
         and each E() within 3 min from B
```

The ordering among events captured by this rule is represented by the *ordering graph* in Figure 4, where an arrow from an event e_1 to e_2 means that e_1 cannot occur after e_2. Ordering graphs capture the fact that some events are bound

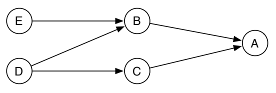

Figure 4: Ordering relations graph

(directly or indirectly) to each other (e.g., D precedes C), while others are not (e.g., C may occur either before or after B). Ordering graphs are acyclic, as it is not possible to specify satisfiable rules with circular timing dependencies; and they are rooted, as TESLA rules force all events to depend directly or indirectly from a unique reference event occurring at evaluation time. A valid pattern satisfying a TESLA rule may start with any of the events that are leaves in the respective ordering graph (e.g., D or E in our example) and it must contain all the events in the graph in the correct order (i.e., the partial order captured by the graph).

5.2 Event detection

To build the event detection automata for a rule r, we start from the ordering graph of r and we create a different, linear, *model of automaton* for each path starting from a leaf and arriving to the root of the graph. As shown in Figure 5, each transition from state s_1 to state s_2 is labeled with the *set of constraints* that an incoming event of type s_2 has to satisfy to trigger the transition, plus the *maximum time* for the transition to be triggered. We capture the fact that different paths in the same ordering graph g share one or more nodes by joining the corresponding states in the automata models derived from g. This is shown using dashed lines in Figure 5. Notice how we used the term "automaton model"

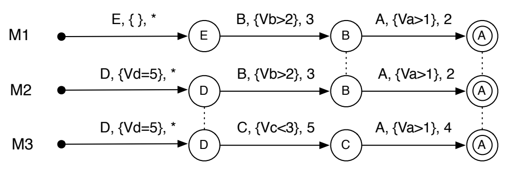

Figure 5: Event detection automata for Rule R

and not simply "automaton" on purpose. Indeed, as we will better explain later, different *instances* of a single model are created during event processing, in order to capture every possible sequence of events that satisfy a given rule.

Detecting simple sequences. To describe the behavior of event detection automata we start by considering those rules that capture a single sequence of events, like the one below:

```
define   CE()
from     A(Va > 1)
         and each B(Vb > 2) within 2 min from A
         and each E() within 3 min from B
```

which is a simplified version of rule R considering only the sequence of events captured by automaton $M1$ (see Figure 5). For each of these rules we have a different linear automaton model like $M1$. Event processing starts by creating a single instance for each of these automata, then, for each incoming event, it creates new automata instances, or moves existing ones from state to state, or deletes some of them. More precisely, an automaton instance A in a state X reacts to the detection of an event e that satisfies the constraints for the transition exiting X, by firstly duplicating itself, creating a new instance A_1, then using e to move A_1 to the next state (while A remains in state X). Those events that do not satisfy currently enabled transitions are simply ignored, while automata instances are deleted if they are unable to progress within the maximum time associated to each transition[4]. Finally, we trigger a rule when an instance of the corresponding automaton model arrives to its accepting state, represented with a double circle in the figures above.

As an example of how this algorithm works, consider our previous rule, resulting in automaton model $M1$. Figure 6 shows what happens when the sequence of events at the bottom is captured. Initially (time $t = 0$) there is a single instance Aut of $M1$, in its initial state, waiting for events of type E. Since this instance will never change, we simplified figure by only drawing it at time $t = 0$. At time $t = 1$ an event $E1$ occurs, which satisfies all the constraints to enter

[4]Notice that an automaton in its initial state cannot be deleted, as transitions exiting the initial state do not include timing constraints.

state E. Consequently, Aut duplicates itself, creating Aut_1, which uses event $E1$ to move to state E. We record this fact by labeling the fired transition with the triggering event $E1$ and the time of occurrence, i.e., $E1@1$. At time $t = 4$, a new instance of E is captured, resulting in a new duplicate of Aut, Aut_2, which moves to state E. At time $t = 5$ Aut_1 has been deleted, since no events of type B had occurred in the 3 minutes following $E1$, while the occurrence of B_1 was captured by Aut_2 which spawned Aut_{21} in state B. At $t = 6$ the new event $B2$ resulted in creating a new copy Aut_{22} of Aut_2 in state B. Finally, at time $t = 8$, when $A1$ occurs, Aut_2 and Aut_{21} have been deleted, while Aut_{22} reacts to the new event by spawning Aut_{221}, which, using $A1$, arrives at the final state A, recognizing the valid sequence ($E2$, $B2$, $A1$).

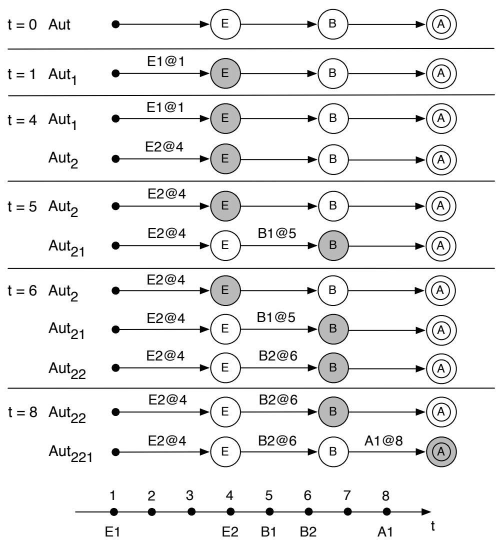

Figure 6: An example of sequence detection

Detecting complex patterns. Once clarified how event processing proceeds for simple rules involving single sequences of events, we are ready to explain what happens in the general case. In presence of rules like Rule R above, processing is complicated by the fact that several automata models share one or more states, as shown by dashed lines in Figure 5. Take for example the relation involving states B in automaton models $M1$ and $M2$: it represents the fact that a sequence captured by an instance of $M1$ can be valid for the whole rule R only if it shares an event B with at least an instance of $M2$. To capture this semantics, we modify the behavior of event processing as follows. Consider an automaton instance A_1 with model M_1, using an event x to move to a state X. Suppose state X in model M_1 is connected with states X in models $M_2, .., M_n$. When A_1 arrives at state X, if at least an automaton instance for each model $M_2, .., M_n$ exist, which used x to move to state X, then the transition is accepted. Otherwise A_1 is deleted. In fact, even if A_1 could recognize a valid sequence s, we are sure event x in s would not satisfy some of the constraints in models $M_2, ..., M_n$, so s cannot satisfy the entire rule.

Event selection and consumption. By creating new automaton instances for each accepted event, the algorithm described above selects all events that satisfy the timing constraints defined in a rule. This is suitable when the rule is defined only through the *each-within* operator, which results in a multiple detection semantics. However, when operators like *first-within* and *last-within* are used, which perform single selections, only a subset of all possible patterns has to be considered. On the other hand, extending our algorithm to capture the semantics of these operators is trivial. In fact, when an automaton instance A arrives in its accepting state using an event e, it is sufficient to compare the pattern it detected with those of all other instances using e. All patterns that do not match the selection policies stated by the corresponding rule (e.g., patterns including an event that is not the "first" when the rule used a *first-within* operator) are discarded. Similar issues result from event consumption: as already consumed events cannot participate in new patterns, all automaton instances that used an event e to arrive to their current state are immediately deleted as soon as e is consumed.

Negation. TESLA includes two forms of negations for a specified event e: the first requires e not to occur between other two events e_1 and e_2, while the second requires e not to occur in a given period before an event e_3. Encoding the first kind of negation in our algorithm is straightforward: when an automaton instance A arrives at a state e_1 waiting for event e_2 to proceed, we simply delete A if e occurs before e_2. The second kind of negation is instead implemented using a timer: suppose we don't want e to occur in the 5 minutes before e_3. Every time an event e is detected a new timer of 5 minutes is set; no new events of type e_3 will be considered while the timer is active.

Aggregates. Aggregate functions require our algorithm to store the set of values they will be applied to. Time intervals from which values are extracted are defined exactly as in negations: using two events e_1 and e_2 as bounds or defining a time span t before the occurrence of an event e_3. In the first case bounds are well defined: we can start recording values of interest when an automaton instance arrives at state e_1 and stop when it moves to state e_2. In the second case we don't know when e_3 will occur, so we need to keep track of all values of interest registered in a time period long t. When an occurrence of e_3 is detected, the recorded values are used as input to the aggregate function.

Performance. The expressiveness of TESLA makes it impossible to set a theoretical upper bound on the number of events that a detection engine needs to keep in memory for processing. Consider for example the following rule:

```
define    CE()
from      A() and each B() within 10 min from A
```

To process it, we need to store all incoming events of type B occurred in the last ten minutes. Since the rate of those events is not known a priori, we cannot set a limit on storage. Moreover, as discussed before, each incoming event that causes a state transition on an automaton instance I also duplicates I. In the worst case, all automata instances are affected by the arrival of an event and consequently the overall number of deployed instances may grow exponentially. While this is bad, the same results hold for all languages that allow users to express time bounded sequences of events [3]. To avoid this potential explosion of the state space, it would be necessary to strongly reduce the expressiveness of TESLA, for example forcing the sequence construct to capture only adjacent events as in [8].

On the other hand, the average case is not necessary the worst one. To measure the performance of our detection algorithm in a practical situation we are implementing it in C++. While the system is not finished, yet, we tested a first, unoptimized version of the prototype against a challenging synthetic workload, in which every input event caused the duplication of many automata and the detection of a large number of complex events. To stress our algorithm we deployed a large number of rules and forced a constant rate of input events. Our preliminary results are encouraging: using an Intel Core2 processor running at 2.53 GHz we could process 5000 rules (25000 automata states) with a constant input rate of 100 events/s with about 98% of cpu usage and less than 700MB RAM usage. In this scenario we registered a peak of more than 1.5 million automata instances, which resulted in detecting more than 62000 events/s.

Note that these results were obtained using a preliminary, single threaded prototype; however, since automata capturing different rules are completely independent from each other, our algorithm can be easily parallelized. Preliminary tests in this direction show that, given a fixed input rate, the maximum number of rules that can be evaluated without loss of information increases linearly with the number of threads and processors.

6. RELATED WORK

The problem of defining a suitable language for processing information flows has been addressed in several works. It first emerged in the field of active database systems, where *Event Condition Action* (*ECA*) rules were used to specify reactive behaviors. Some of the languages for defining ECA rules included operators for event composition. Examples are HiPac [14], Samos [17], Sentinel [10] and Ode [18]. Interestingly, some of these systems addressed the problem of event selection by allowing users to choose among a set of predefined policies; however, languages were not flexible enough to enable the definition of new policies.

Many languages for processing flows of information have been developed by the *Data Stream Processing* community. Examples are CQL [5], ESL [7], and StreaQuel [11]. All of them share the same processing paradigm, in which *windows* operators are used to split input streams into parts, which are processed using declarative, SQL like, languages to build new output streams. Processing is usually performed incrementally, by connecting several rules together, so that results produced by a rule become input for others. Some systems explicitate this model by offering graphical tools, in which predefined rules and user defined ones are seen as building blocks to be combined by drawing the flow of information from block to block [1, 24]). As discussed in Section 2, this approach does not naturally allow to capture the kind of complex events we focus in this paper.

Most recent proposals coming from the *Complex Event Processing* community [28, 30, 25, 21, 8] offer relatively simple languages, which usually do not include customizable event selection and consumption policies, complete support for aggregates or reuse of patterns to form hierarchies of events. Remarkable exceptions are represented by the pattern languages defined by Sase+ [20, 3] and Amit [2], which present many similarities with our approach. Sase+ defines flexible event selection strategies, while offering a precise semantics for all operators in terms of NFA automata; it also supports parameterization, negation, and aggregates. However, Sase+ rules can specify only single sequences of events, while TESLA may define complex patterns that include different sequences. Moreover, selection policies are not completely customizable in SASE+ and apply to entire rules, rather then to single operators, as in TESLA. Finally, SASE+ does not consider event consumption and periodic evaluation. Amit introduces the concept of *lifespan* to specify the valid period in which a pattern of events can be captured. As in TESLA, different lifespans may be concurrently open to capture different occurrences of an event. Amit also provides customizable event selection and consumption policies. On the other hand, Amit patterns cannot include a number of timing constraints defined in TESLA, especially those including negations, and it does not include aggregates. Finally, Amit operators and their compositions are not formally defined, making the behavior of some rules potentially unclear.

While different algorithms for event processing have been proposed, the automata paradigm is the most common in existing systems [8, 3, 21]. To the best of our knowledge, no other proposals present in the literature allow to capture complex patterns that include and combine several sequences of events, like our event detection automata. Moreover, each existing automata-based algorithm is tailored for a specific language having its own selection policies; for this reason no one is general enough to capture the entire expressiveness of TESLA, which provides a high degree of freedom for rule definition.

Finally, there are a number of studies on underlying event and time models for complex event processing systems [29, 31], which provide a teoretical exploration on the nature of events.

7. CONCLUSIONS

In this paper we presented TESLA, an event specification language for CEP. TESLA provides a simple and compact syntax while offering high expressiveness and flexibility: it supports content-based event filtering and allows to easily capture complex relations among temporally related patterns of events. It supports parameterization, negations, and aggregates, offering standard and periodic rules within a single framework. It also clearly separates the role of rules, used to define complex events, from the role of subscriptions, used to express the interests of sinks, allowing rules to be easily combined together in strongly expressive hierarchies. All these features, together with the ability of specifying fully customizable policies for event selection and consumption, allows TESLA to easily define event iterations without requiring an explicit Kleene operator, i.e., keeping the language syntax simple and elegant. Moreover, TESLA is among the first languages for CEP to offer a formal semantics, expressed using a temporal logic. This eliminates the ambiguities typical of most other languages and allows system designers to formally check the correctness of their implementation. Finally, we have shown how TESLA rules can be efficiently interpreted, introducing an event detection algorithm based on automata. As a next step, we are currently integrating TESLA with RACED [12], our protocol for distributed event processing. We are confident that this would allow us to combine language expressiveness with system scalability.

Acknowledgment

This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom; and by the Italian Government under the projects FIRB INSYEME and PRIN D-ASAP.

8. REFERENCES

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[2] A. Adi and O. Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, 2004.

[3] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD*, pages 147–160, New York, NY, USA, 2008. ACM.

[4] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. Stream: the stanford stream data manager (demonstration description). In *SIGMOD*, pages 665–665, New York, NY, USA, 2003. ACM.

[5] A. Arasu, S. Babu, and J. Widom. The cql continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, pages 1–16, New York, NY, USA, 2002. ACM.

[7] Y. Bai, H. Thakkar, H. Wang, C. Luo, and C. Zaniolo. A data stream language and system designed for power and extensibility. In *CIKM*, pages 337–346, New York, NY, USA, 2006. ACM.

[8] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White. Cayuga: a high-performance event processing engine. In *SIGMOD*, pages 1100–1102, New York, NY, USA, 2007. ACM.

[9] K. Broda, K. Clark, R. M. 0002, and A. Russo. Sage: A logical agent-based environment monitoring and control system. In *AmI*, pages 112–117, 2009.

[10] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In *VLDB*, pages 606–617, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD*, pages 668–668, New York, NY, USA, 2003. ACM.

[12] G. Cugola and A. Margara. Raced: an adaptive middleware for complex event detection. In *ARM*, pages 1–6, New York, NY, USA, 2009. ACM.

[13] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. Technical report, Politecnico di Milano, 2010. Submitted for Publication.

[14] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The hipac project: combining active databases and timing constraints. *SIGMOD Rec.*, 17(1):51–70, 1988.

[15] A. J. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.

[16] Event zero, http://www.eventzero.com/solutions/environment.aspx.

[17] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Rules in Database Systems*, pages 23–39, 1993.

[18] N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *VLDB*, pages 327–336, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[19] C. Ghezzi, D. Mandrioli, and A. Morzenti. Trio: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, 1990.

[20] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. On supporting kleene closure over event streams. In *ICDE*, pages 1391–1393, 2008.

[21] G. Li and H.-A. Jacobsen. Composite subscriptions in content-based publish/subscribe systems. In *Middleware*, pages 249–269. Springer-Verlag New York, Inc., 2005.

[22] A. Morzenti, D. Mandrioli, and C. Ghezzi. A model parametric real-time logic. *ACM Trans. Program. Lang. Syst.*, 14(4):521–573, 1992.

[23] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[24] Oracle cep. http://www.oracle.com/technologies/soa/complex-event-processing.html.

[25] P. R. Pietzuch, B. Shand, and J. Bacon. Composite event detection as a generic middleware extension. *IEEE Network*, 18:44–55, 2004.

[26] N. P. Schultz-Møller, M. Migliavacca, and P. R. Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*, 2009.

[27] U. Srivastava and J. Widom. Flexible time management in data stream systems. In *PODS '04*, pages 263–274, New York, NY, USA, 2004. ACM.

[28] F. Wang and P. Liu. Temporal management of rfid data. In *VLDB*, pages 1128–1139. VLDB Endowment, 2005.

[29] W. White, M. Riedewald, J. Gehrke, and A. Demers. What is "next" in event processing? In *PODS*, pages 263–272, New York, NY, USA, 2007. ACM.

[30] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *SIGMOD*, pages 407–418, New York, NY, USA, 2006. ACM.

[31] D. Zimmer. On the semantics of complex events in active database management systems. In *ICDE '99*, page 392, Washington, DC, USA, 1999. IEEE.