The Journal of Systems and Software 85 (2012) 1709–1728

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Complex event processing with T-REX

Gianpaolo Cugola, Alessandro Margara*

Dip. di Elettronica e Informazione, Politecnico di Milano, Italy

ARTICLE INFO

Article history:
Received 16 November 2010
Received in revised form 20 March 2012
Accepted 25 March 2012
Available online 6 April 2012

Keywords:
Complex Event Processing
Event middleware
Information processing
Distributed systems

ABSTRACT

Several application domains involve detecting complex situations and reacting to them. This asks for a *Complex Event Processing* (CEP) middleware specifically designed to timely process large amounts of *event notifications* as they flow from the peripheral to the center of the system, to identify the *composite events* relevant for the application. To answer this need we designed T-Rex, a new CEP middleware that combines expressiveness and efficiency. On the one hand, it adopts a language (TESLA) explicitly conceived to easily and naturally describe composite events. On the other hand, it provides an efficient event detection algorithm based on automata to interpret TESLA rules. Our evaluation shows that the T-Rex engine can process a large number of complex rules with a reduced overhead, even in the presence of challenging workloads.

1. Introduction

Several systems operate by observing a set of *primitive events* that happen in the external environments, interpreting and combining them to identify higher level *composite events*, and finally sending the notifications about these events to the components in charge of reacting to them, thus determining the overall system's behavior. Examples are sensor networks for environmental monitoring (Broda et al., 2009; Event Zero, 2011); financial applications requiring a continuous analysis of stocks to detect trends (Demers et al., 2006); fraud detection tools, which observe streams of credit card transactions to prevent frauds (Schultz-Møller et al., 2009); RFID-based inventory management, which performs a continuous analysis of registered data to track valid paths of shipments and to capture irregularities (Wang and Liu, 2005). More in general, as observed in (Luckham, 2001), the information system of every complex company can and should be organized around an *event-based core* that realizes a sort of nervous system to guide and control the operation of the other sub-systems.

The general architecture of such event-based applications is shown in Fig. 1. At the peripheral of the system are the *sources* and the *sinks*. The former observe primitive events and report them, while the latter receive composite event notifications and react to them. The task of identifying composite events from primitive ones is performed by the *Complex Event Processing (CEP) engine*. This fundamental component is usually part of a *CEP middleware* (which also includes the client-side libraries to access the engine) and operates by interpreting a set of *event definition rules*, which describe how composite events are defined from primitive ones.

Recently both the academia and industry proposed a number of CEP engines. As we will better explain in Section 6, they present significant differences in terms of the *rule language* they adopt, which, in turn, strongly influences the processing algorithm they adopt and their overall system architecture.

Many of these systems set their roots in the data management domain. Their goal is to timely process generic data, not necessarily event notifications, using *standing queries* (Babcock et al., 2002), which continuously calculate new results as new data enters the system. This functionality is usually embedded into a DBMS-like relational core, using rule languages directly derived from SQL. These languages provide an effective way to filter, join, and transform large volumes of streaming data, but they do not naturally fit the event-processing domain. In particular, the "data transformation" approach they follow is not suited to "event recognition", i.e., to recognize patterns of event notifications tied together by complex temporal relationships. This issue is addressed by those systems that are built around a native concept of event notification. Unfortunately, in most cases they trade expressiveness for efficiency. They adopt simple rule languages that cannot express a number of desirable event patterns and the actions needed to aggregate the information carried by primitive events into composite events.

T-Rex, the CEP middleware presented in this paper, solves this dichotomy by offering a language, TESLA (Cugola and Margara, 2010), explicitly designed to model in an easy and natural way the complex ordering and timing relationships that join primitive events and the actions required to aggregate them, while, at the

* Corresponding author.
E-mail addresses: cugola@elet.polimi.it (G. Cugola), margara@elet.polimi.it (A. Margara).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.03.056

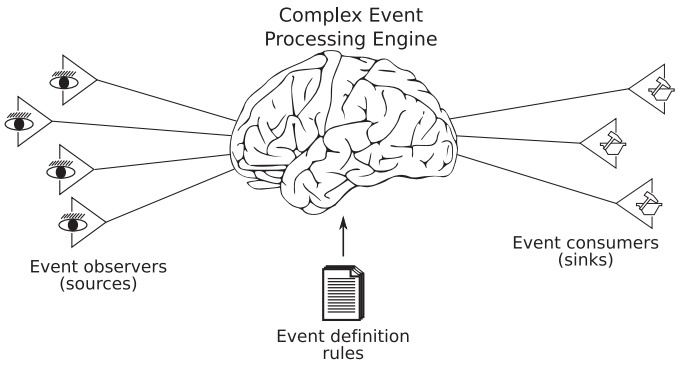

Fig. 1. The high-level view of an CEP application.

same time, remaining simple enough to allow an efficient processing of rules.

In this paper we present the main characteristics of TESLA and we show how the T-Rex engine translates TESLA rules into automata to provide efficient event detection. We describe the architecture of T-Rex in details and we analyze its performance, comparing it to a state of the art commercial product.

The rest of the paper is organized as follows: next section introduces a simple case study to illustrate the requirements of a CEP rule language and how TESLA meets them. The same case study is also used to describe how T-Rex operates. This part is divided into two sections: Section 3 describes how TESLA rules are translated into automata, while Section 4 describes how the T-Rex engine implements such automata to efficiently process event notifications. Finally, Section 5 evaluates the performance of T-Rex, while Section 6 describes related work, and Section 7 ends the paper with brief concluding remarks.

2. The TESLA language

Consider an environment monitoring application that processes information coming from a sensor network. Sensors notify their position, the air temperature they measure, and the presence of smoke and rain. Now, suppose a user wants to detect the presence 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 presence of fire can be detected in many different ways. Here we present four possible definitions of the fire event, and we use them to illustrate some of the features an event processing language should provide.

D1. There is fire when there is smoke and a temperature higher than 45° in the same area of smoke (detected within 5 min from smoke). The fire notification has to embed the temperature actually measured.
D2. There is fire in presence of a temperature higher than 45° and in absence of rain (in the last hour).
D3. There is fire when there is smoke and the average temperature in the last 5 min in the same area is higher than 45°.
D4. There is fire when there is smoke preceded by at least 10 temperature readings with increasing values in the same area within 5 min from smoke. The fire notification has to embed the average temperature of the increasing sequence.

First of all, these rules *select* relevant notifications 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°) 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 5 min) and they allow to capture *sequences* of events (e.g., high temperature followed by smoke).

Beside selection, definition *D2* introduces *negation* by requiring an event not to occur in a given interval. Similarly, definition *D3* introduces *aggregates*. In particular, it defines a function (average) to be applied to a specified set of elements (temperature readings in the last 5 min) to calculate the value associated with the composite event. Definition *D4* is interesting as it combines the two kinds of selection constraints (those on values and those on timing) to define an *iteration* that selects only those elements that bring growing temperature readings. Finally, when the desired combination of events has been detected, rules have to specify which notification to create (e.g., fire) and they have to define its inner structure (e.g., fire notification has to embed an area and a temperature reading).

A language for CEP should be able to express all the constructs above: selection, parameterization, negations, aggregates, sequences, and iterations. Unfortunately, as we will better explain in Section 6 by discussing related work, this is rarely the case (Cugola and Margara, 2011). Moving from this consideration we designed TESLA: an event definition language that provides a high degree of expressiveness to users, while adopting a structure and organization that allows efficient detection of rules. Next section presents the main TESLA features by examples, showing how it can be used to express the four rules above. The interested reader can find a complete description of the language, including a rigorous definition of its semantics, in (Cugola and Margara, 2010).

2.1. 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 Fig. 1), which encode them in *event notifications* (or simply *events*). We assume that each event notification has a *type*, which defines the number, order, names, and types of the *attributes* that build the notification. Notifications have also a timestamp, which represents the time of occurrence of the event they encode. In the following we assume events enter the system in timestamp order: mechanisms to cope with out-of-order arrivals of events have been discussed in the past and can be adopted to ensure this property (Srivastava and Widom, 2004).

In our fire scenario, the air temperature in a given area at a specific time is captured by the following event notification:

$$Temp@10(area=''A1'', value = 24.5)$$

where `Temp` represents the type of the notification and `10` is the timestamp. The `Temp` notification type includes two attributes: a string that identifies the area in which the temperature was measured, and the actual reading (a float).

TESLA rules define composite events from simpler ones. The latter can be observed directly by sources (i.e., they can be primitive events) or they can be composite events defined by other rules. This mechanism allows the definition of "hierarchies of events".

Each TESLA rule has the following general structure:

```
Rule R
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
```

Intuitively the first two lines define a (composite) 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 new 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.

2.2. TESLA by examples

To present the operators supported by TESLA in an easy and clear way, we use the four definitions of `Fire` presented above and show how they can be encoded in TESLA.

If we carefully look at the definition *D1* we may notice that it is ambiguous. Indeed, it is not clear what happens if the `Smoke` event is preceded by more than one `Temp` event. In such cases we say that the *selection policy* of the rule is ambiguous (Cugola and Margara, 2011). TESLA is both very rigorous and very expressive about this point, allowing users to precisely define the selection policy they have in mind. By using the `each-within` operator, Rule R1 below:

```
Rule R1
define      Fire(area: string, measuredTemp: double)
from        Smoke(area=$a) and
            each Temp(area=$a and value>45)
            within 5 min from Smoke
where       area=Smoke.area and measuredTemp=Temp.value
```

adopts a *multiple* selection policy: when a `Smoke` event is detected it notifies as many `Fire` events as the `Temp` events observed in the last 5 min. Other policies can be encoded by substituting the `each-within` operator with the `last-within` or the `first-within` operators, or with their generalized versions `n-last-within` and `n-first-within`. As an example, in the presence of three temperature readings greater than 45° followed by a `Smoke` event, Rule R1 would notify three `Fire` events, while the rule below:

```
Rule R2
define      Fire(area: string, measuredTemp: double)
from        Smoke(area=$a) and
            last Temp(area=$a and value> 45)
            within 5 min from Smoke
where       area=Smoke.area and measuredTemp=Temp.value
```

would notify a single `Fire` event with the last temperature read.

Another source of ambiguity in Definition *D1* relates with the *consumption policy* (Cugola and Margara, 2011; Chakravarthy et al., 1994) to use. In particular, imagine a multiple selection policy is adopted as in Rule R1 and consider the case when three `Temp` events are immediately followed by two `Smoke` events. When the first `Smoke` event is detected three `Fire` events are notified, but what happens when the second `Smoke` arrives? Are the three `Temp` events already used to notify the first three `Fire` events reused, thus notifying three more `Fire` events, or they are not? Again, TESLA allows users to define the consumption policy they prefer. Indeed, Rule R1 does not explicitly *consume* the events it uses, so every temperature reading may be used more than once in presence of several `Smoke` events, while Rule R3 below uses the `consuming` clause to model the fact that `Temp` events can be used at most once:

```
Rule R3
define      Fire(area: string, measuredTemp: double)
from        Smoke(area=$a) and
            each Temp(area=$a and value> 45)
            within 5 min from Smoke
where       area=Smoke.area and measuredTemp=Temp.value
consuming   Temp
```

Besides these aspects, Rule R1 (but also R2 and R3) show how in TESLA the occurrence of a composite event is always bound to the occurrence of a simpler event (`Smoke` in our example), which implicitly determines the time at which the new event is detected. This anchor point is coupled with other events (`Temp` in our example) through ad-hoc operators, like the `each-within`, which capture the temporal relationships that join together events in sequences. Finally Rules R1, R2, and R3 provide also an example of parameterization, which is modeled in TESLA by using the variable `$a` to bind the values of attribute `area` in different events.

The second definition of `Fire` introduced at the beginning of this section can be used to show how time-based negations are expressed in TESLA:

```
Rule R4
define      Fire(area: string, measuredTemp: double)
from        Temp(area=$a and value> 45) and
            not Rain(area=$a) within 5 min from Temp
where       area=Temp.area and measuredTemp=Temp.value
```

Beside time-based negations, TESLA allows interval-based negations, by requiring a specific event not to occur between other two events.

Rule R5 shows the use of aggregates (i.e., the function `Avg`) to express the `Fire` definition *D3*.

```
Rule R5
define      Fire(area: string, measuredTemp: double)
from        Smoke(area=$a) and
            45 < $t=Avg(Temp(area=$a).value
            within 5 min from Smoke)
where       area=Smoke.area and measuredTemp=$t
```

Also aggregates can be time-based (as in this example) or interval-based.

As a final example, we show how the `Fire` definition *D4* can be expressed in TESLA. In this case we combine different rules together, taking advantage of TESLA ability to define composite events starting from other composite events. In particular, we first define the `NotIncrTemp` event that happens each time a new temperature is read that is lower or equal than the previous one (Rule R6a). Using this event, we define `Fire` when there is smoke preceded by at least 10 temperature readings and no `NotIncrTemp` in the last 5 min (Rule R6b). Finally, with Rule R6c we capture the case when the last *NotIncrTemp* precedes `Smoke` by less than 5 min but from that time there were at least 10 other temperature readings (which are necessarily increasing).

```
Rule R6a
define      NotIncrTemp(area: string)
from        Temp(area=$a and value=$t) as T1 and
            last Temp(area=$a and value> =$t) as T2
            within 5 min from T1 and
            not Temp(area=$a) between T2 and T1
where       area=T1.area
Rule R6b
define      Fire(area: string, measuredTemp: double)
from        Smoke(area=$a) and
            not NotIncrTemp(area=$a)
            within 5 min from Smoke and
            Count(Temp(area=$a)
            within 5 min from Smoke) > = 10
where       area=Smoke.area and measuredTemp=
            Avg(Temp(area=$a) within 5 min from Smoke)
Rule R6c
define      Fire(area: string, measuredTemp: double)
from        Smoke(area=$a) and
            last NotIncrTemp(area=$a) as N
            within 5 min from Smoke and
            Count(Temp(area=$a)
            between Smoke and N) > = 10
where       area=Smoke.area and measuredTemp=
            Avg(Temp(area=$a) between Smoke and N)
```

Notice that this ability to combine rules together is a key feature of TESLA and T-Rex: other systems, include ad-hoc *iteration* operators to deal with complex sequences as *D4*. As we will better motivate in Section 6, these operators are difficult to use in conjunction with customizable selection and consumption policies. The ability to define hierarchies of events is more general, resulting in a simpler and elegant solution applicable in a wide range of cases.

Summary of the language. In conclusion, TESLA provides all the key constructs identified above: selection of primitive events, definition of sequences, parameterization, negation, aggregates, and iteration expressed through recursive rules. It combines these constructs with user definable selection and consumption policies.

3. Event detection automata

In this section we introduce the automata-based processing algorithm used in T-Rex. We show the rational behind the algorithm and we describe in details how TESLA rules are translated into automata and how they are used during processing.

3.1. Event detection automata. Why?

Sequences of events are at the heart of TESLA: a composite event occurs when the last event (the *terminator*) of all the sequences in a rule is detected, e.g., Rule R1 fires when the `Smoke` event is detected. When we first started to deal with the design of a processing algorithm for TESLA, we evaluated two opposite approaches. The first one postpones all the processing to the time in which a suitable terminator is found: in this case the processing performs an analysis of all previously received events to capture valid patterns. The second approach processes rules incrementally: it recognizes and stores partial sequences in the form of automata as soon as they are detected. Both algorithms present advantages and disadvantages: the first one requires less memory (the memory required to store automata) but may introduce a higher processing delay, the second should reduce latency in detecting composite events, indeed when the terminator arrives most of the processing has already been done.

To better understand the potential of the two approaches, we initially implemented two separate prototypes and compared their behaviors under different loads. As expected, we found that the first approach uses a fraction of the memory required by the second one. At the same time the memory demand of the second approach, albeit greater, was small enough to allow us to complete all our tests, even the most demanding ones, on a machine with 8 GB of RAM (see Section 5). The differences in processing time were significantly influenced by the features of the workload we considered: in particular, the incremental approach provided greater advantages with a multiple selection policy and higher rates of potentially terminator events (i.e., the hardest workloads). Since we were interested in optimizing performance in spite of the amount of resources required, we choose the second approach (and prototype), which is the one we consider in the remainder of the paper. Notice that a similar choice is done by most existing event and stream processing systems, both from the academia (Brenna et al., 2007) and the industry (Esper, 2011).

Before entering into the details of the T-Rex processing algorithm we introduce the terminology we use. T-Rex translates each rule into what we call an *automaton model*, which is composed of one or more *sequence models*. At the beginning, each sequence model is instantiated in a *sequence instance* (or simply *sequence*). New sequences are created at run-time, while event notifications enter the engine. More specifically, when a new event notification enters T-Rex several things may happen:

- new sequences can be created by duplicating existing ones;
- existing sequences can be moved from a state to the following one;
- existing sequences can be deleted, either because they arrive to an accepting state, representing the detection of a new composite event, or because they become invalid, i.e., unable to proceed any further.

3.2. Creation of automata

As described in Section 2, each TESLA rule filters event notifications according to their type and content, and uses the `*-within` operators to define one or more sequences of events. Additional constraints between events can be introduced using parameters and negations. Consider for example the following Rule R7, which combines Rules R1 and R4 (also introducing the `Wind` event) to build a complex enough rule to describe our algorithm in all its aspects:

```
Rule R7
define      Fire(area: string, measuredTemp: double)
from        Smoke(area=$a) and
            each Temp(area=$a and value>45)
            within 5min from Smoke and
            each Wind(area=$a and speed>20)
            within 5min from Smoke and
            not Rain(area=$a) between Smoke and Wind
where       area=Smoke.area and measuredTemp=Temp.value
```

Rule R7 captures two sequences of events that concur in defining the `Fire` event. One is composed by a `Temp` event followed by `Smoke`, the other is composed by a `Wind` event followed by `Smoke`. These two sequences share a common event (i.e., `Smoke`). This is typical with TESLA, as it forces all sequences defined by a rule to share (at least) their last event: the terminator that implicitly determines the time at which the rule fires and the composite event is detected. Besides this aspect, Rule R7 defines additional relationships between the two sequences of events that define `Fire`:

1. the `Smoke`, `Wind`, and `Temp` events have to refer to the same "area". This is captured by using the shared parameter `$a`;
2. the interval between the `Wind` and `Smoke` events cannot include any `Rain` event. This is captured by using the `not-between` TESLA operator.

Algorithm 1. Translation Algorithm

```
createSequenceModels(seqMods)                                   1
joinStates(seqMods)                                             2
addParameters(seqMods)                                          3
addNegations(seqMods)                                           4
addAggregates(seqMods)                                          5

function createSequenceModels(seqMods)                         7
  for each ev in getInitEvents()                                8
    seqMod = new sequenceModel(ev)                              9
    currentEv = ev                                              10
    while currentEv !=getTerminatorEvent()                      11
      nextEv = getNextEvent(currentEv)                          12
      selPolicy = getSelectionPolicy(currentEv,nextEv)          13
      win = getWindowBetween(currentEv,nextEv)                  14
      addTransition(seq,currentEv,nextEv,selPolicy,win)         15
    setFinalState(seqMod,currentEv)                             16
    addSequence(seqMods,seqMod)                                 17

function joinStates(seqMods)                                    19
  for each seqMod in seqMods                                    20
    for each state in states                                    21
      for each sm in getSeqModelHavingState(seqMods,state)      22
        joinStates(seqMod,sm,state)                             23

function addParameters(seqMods)                                 25
  for each param in getParameters()                             26
    ev1 = getFirstParameterEvent(param)                         27
    ev2 = getSecondParameterEvent(param)                        28
    att = getAttribute(param)                                   29
    op = getOperator(param)                                     30
    addParameter(seqMods,ev1,ev2,att,op)                        31

function addNegations(seqMods)                                  33
  for each neg in getNegations()                                34
    negEv = getNegationEvent(neg)                               35
    ev1 = getFirstEvent(neg)                                    36
    if isTimeBased(neg)                                         37
      win = getWindowFor(neg)                                   38
      addTimeBasedNegation(seqMods,negEv,ev1,win)               39
    else                                                        40
```

Algorithm 1. (*Continued*)

```
    ev2 = getSecondEvent(neg)                                   41
    addIntervalBasedNegation(seqMods,negEv,ev1,ev2)             42

function addAggregates(seqMods)                                 44
  for each agg in getAggregates()                               45
    aggEv = getAggregateEvent(agg)                              46
    ev1 = getFirstEvent(agg)                                    47
    if isTimeBased(agg)                                         48
      win = getWindowFor(agg)                                   49
      addTimeBasedAggregate(seqMods,aggEv,ev1,win)              50
    else                                                        51
      ev2 = getSecondEvent(agg)                                 52
      addIntervalBasedAggregate(seqMods,aggEv,ev1,ev2)          53
```

These considerations can be generalized into Algorithm 1, which translates a generic TESLA Rule R into an event detection automaton model. First, the sequences of events captured by R are identified and a *sequence model* (i.e., a linear, deterministic, finite state automaton) is built for each of them. This step is performed by the `createSequenceModels` function (line 7): it processes sequences one by one, starting from their initiating events, as returned by the `getInitEvents` function (line 8). For each sequence S captured by Rule R, a new state is created. The first state is defined when the sequence model is initialized (line 9), while the following ones are iteratively added by calling the `addTransition` function (line 14). This function creates a transition between two states, labeled with the *content and timing constraints* that an incoming event has to satisfy to trigger the transition. Finally, the last state is annotated as an accepting one (line 16) and the created sequence is added to the sequence models (line 17). As an example, Fig. 2 shows the sequence models, $M1$ and $M2$, derived from Rule R7.

After the sequence models originating from Rule R are built, they are annotated and combined in a single automaton model by introducing the relationships among the states captured by the rule (modeled through dashed lines in Fig. 2). In particular, the `joinStates` function connects together the states shared by two or more sequence models. Afterwards, parameters, negations, and aggregates are introduced. Fig. 2 shows the automaton model that results from Rule R7, which includes the relationships indicating the fact that S is a shared state, while states T, W, and S are connected by a parameter, and states W and S are connected by the negated event `Rain` that should not occur between them.

3.3. Processing algorithm

To describe how automaton models created from TESLA rules are used to process incoming events we first describe the processing algorithm for single sequence models, then we show how sequences can interact to capture generic TESLA rules, like Rule R7 above.

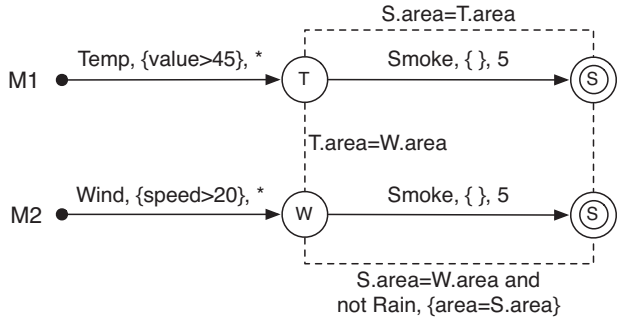

Fig. 2. Event detection automata for Rule R7.

3.3.1. Processing single sequences

Consider a rule that captures a single sequence of events like Rule R1 (reported below for clarity), which represents the first half of Rule R7. As we mentioned above, this rule is translated into the sequence model $M1$ of Fig. 2.

Rule R1

```
define    Fire(area: string, measuredTemp: double)
from      Smoke(area=$a) and
          each Temp(area=$a and value>45)
          within 5min from Smoke
where     area=Smoke.area and measuredTemp=Temp.value
```

When the rule is deployed, the corresponding model is created and a single sequence is instantiated from the model and installed in the system, waiting in its initial state for the arrival of appropriate events. When a new event e arrives, the algorithm reacts as follows: (i) it checks whether the type, content, and arrival time of e satisfy a transition for one of the existing sequences; if not, the event is immediately discarded. If a sequence Seq in a state s_1 can use e to move to its next state s_2, the algorithm (ii) creates a copy S' of Seq, and (iii) it uses e to move S' to state s_2. Notice that the original sequence Seq remains in state s_1, waiting for further events. Sequences are deleted when it becomes impossible for them to proceed to the next state since the time limits for future transitions have already expired. Notice that a sequence in its initial state is never deleted as it cannot expire.

As an example of how processing of sequences works, consider Rule R1 and the corresponding model $M1$. Fig. 3 shows, step by step, how the set of incoming events drawn at the bottom is processed. At time $t = 0$ a single sequence Seq is present in the system, waiting in its initial state. Since Seq does not change with the arrival of new events, we omit it in the figure for all time instants greater than 0. At time $t = 1$ an event $T1$ of type `Temp` enters the system. Since it matches type, content, and timing constraints for the transition to state T, we duplicate Seq, creating Seq_1, which advances to T. Similarly, at $t = 2$, the arrival of a new event $T2$ of type `Temp` creates a new sequence Seq_2 from Seq and moves it to state T. At time $t = 5$ a `Smoke` event arrives but from the wrong area, so it is immediately discarded. At time $t = 7$, Seq_1 is deleted, since it has no possibility to proceed further without violating the timing constraint of its outgoing transition. At the same time, the arrival of event $T3$ generates a new sequence Seq_3. At time $t = 8$ Seq_2 is deleted, while the arrival of an event $S2$ of type `Smoke` from the correct area duplicates Seq_3, generating and advancing Seq_{31} to its accepting state S. This means that a valid sequence, composed by events $T3$ and $S2$ has been recognized. After detection, the sequence Seq_{31} is deleted. Similarly, at $t = 9$, the arrival of $S3$ causes the creation of sequence Seq_{32} and the detection of the valid sequence composed by $T3$ and $S3$.

Notice that active sequences store the information about the events they used to reach their current state, i.e., their content and the time when they occurred. As we said, the occurrence time of events is used to check when a sequence can be deleted, while the content of events is used to check the value of relevant attributes, like `area` in our example, and to build the content of the generated events.

As a final remark, notice that sequences may include negations, which constrain an event e not to occur in a given interval i. In particular, the interval i can be expressed using two events (say, e_1 and e_2) belonging to the same sequence, or using an event and a maximum time-span (say, e_3 and t). The two cases are processed using different techniques: in the first case, when e arrives all sequences that have already reached the state associated with the event e_1, but not that associated with e_2, are immediately deleted; in the second case, instead, all e events occurred within t are stored. When a sequence arrives at the state associated with event e_3, we check if there are instances of e arrived within t: if so, the sequence is immediately deleted, otherwise it may proceed.

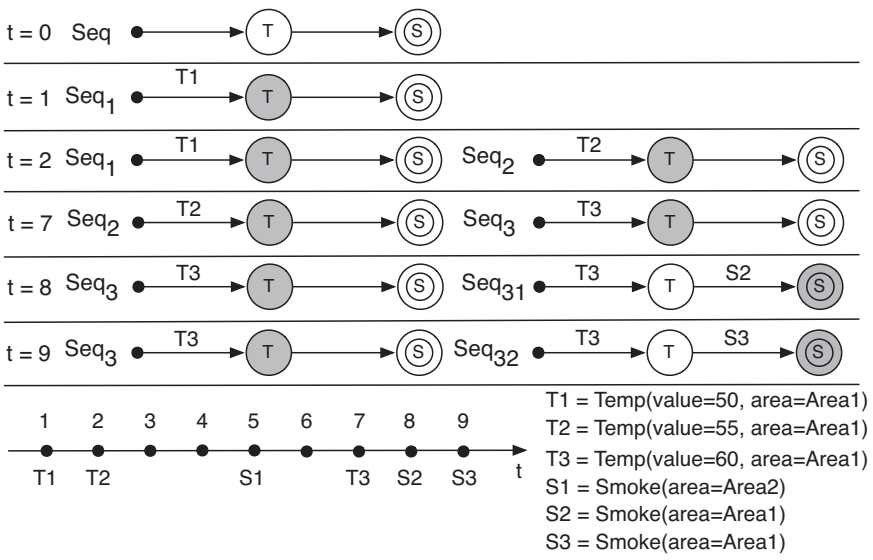

Fig. 3. An example of sequence processing.

3.3.2. Processing complete rules

As we said, TESLA rules usually capture multiple sequences of events, always sharing at least one event with each other. Consider for example Rule R7: it is translated into the automaton of Fig. 2, composed of models $M1$ and $M2$, which share the state S entered when an event of type `Smoke` is detected.

Now suppose that the arrival of a `Smoke` event participates into three sequence instances for model $M1$ and two sequence instances of model $M2$, causing all of them to move to their final accepting state. This means that six different composite events `Fire` are detected, all sharing the same `Smoke` event. Indeed, since both sequences are using a multiple selection policy, each possible couple of sequences ($S1$, $S2$) (with $S1$ instance of model $M1$ and $S2$ instance of model $M2$) captures a valid set of events for the entire rule.

On the contrary, if a `Smoke` event is only used to accept sequences of type $M1$, but none of type $M2$ (or vice-versa), we can safely discard all sequences using it, since they have no chance to be combined with any instance of $M2$ (resp. $M1$) to satisfy all rule's constraints.

More in general, when a sequence S, instance of a sequence model M, can move to state X shared with models $M_1, \ldots, M_n$ using an event e, our algorithm first checks if e can cause a transition to X in at least one sequence of each model $M_1, \ldots, M_n$. If this is not the case, then S is deleted, otherwise it advances to state X.

The same steps are followed to verify all the relationships defined among different sequences, like those resulting from the use of parameters. As an example, Rule R7 includes a parameter constraint on the `area` attribute of `Temp` and `Wind` events. This constraint has to be checked when two sequences (one instance of $M1$, the other instance of $M2$) that may potentially result in detecting a `Fire` event, both reach their final state.

3.3.3. Selection and consumption policies

Another feature of TESLA that we need to capture in our processing algorithm is the definition of event selection and consumption policies. Capturing event consumption is straightforward: simply, when a sequence S resulting from a Rule R arrives to its accepting state using and consuming an event e, we can delete all other sequences resulting from the same Rule R that made use of e and have not yet arrived to their accepting state. Indeed, they are all invalidated by the consumption of e.

With respect to event selection, the described algorithm, by duplicating sequence instances at each state transition, captures all possible sequences of events that satisfy the content and timing constraints of a rule, i.e., this algorithm always captures the multiple selection policy typical of the `each-within` operator. However, TESLA includes operators, like `first-within` and `last-within`, which define different selection policies in which only a subset of all possible sequences need to be selected. The simplest way to implement these operators is to select or discard sequences only when they arrive to their final accepting state. In the most general case this approach is also the only possible one. Consider for example the following Rule R8, which adds a consuming clause to Rule R2:

```
Rule R8
define      Fire(area: string, measuredTemp: double)
from        Smoke(area=$a) and
            last Temp(area=$a and value>45)
            within 5 min from Smoke
where       area=Smoke.area and measuredTemp=Temp.value
consuming   Temp
```

Now suppose we receive two `Temp` events, say $T1$ and $T2$, in this order. When $T2$ arrives we cannot discard $T1$, as we could initially be tempted to do; indeed, the arrival of a `Smoke` event would cause the generation of a new composite event, but also the consumption of $T2$, making $T1$ become the last event received, and so the right candidate for further rule evaluations. On the other hand, rule specific optimization are possible: take for example Rule R2, which does not consume events. In this case, when $T2$ occurs, $T1$ cannot become the last occurred event for next evaluations, so it can be safely deleted. More in general, in the presence of a `last-within` operator associated with an event X, without a `consuming` clause for X, we can safely discard sequences composed by previous events of type X when a new one is detected. Similar rule-specific optimization's are possible also when the `first-within` operator is involved. Consider for example the following Rule R9:

```
Rule R9
define      Fire(area: string, measuredTemp: double)
from        Smoke(area=$a) and
            first Temp(area=$a and value>45)
            within 5 min from Smoke
where       area=Smoke.area and measuredTemp=Temp.value
```

and suppose we receive two events of type `Temp`, $T1$, at time $t = 1$, and $T2$, at time $t = 4$. If we receive an event S of type `Smoke` at time $t = 5$ we need to duplicate only the sequence including $T1$ (i.e., the first event of type `Temp` within 5 min from S) not the one including

*T*2. Notice that in this case, in which the `first-within` operator is used, we cannot delete the sequence including *T*2, as this event can become the first `Temp` event valid when time proceeds and *T*1 becomes too old to be considered again. However, by duplicating only the sequences that include the first detected event we greatly reduce the number of automata stored in the system.

4. Implementation of T-Rex

In this section we describe the implementation of T-Rex, starting from its architecture and presenting the features that contribute to the efficient processing of events. Indeed, while our automata-based event detection algorithm minimizes processing time by incrementally computing results as new events enter the system, it may generate a large number of sequence instances. Accordingly, in implementing this algorithm in C++, we adopted advanced memory management techniques to avoid duplicating data (like events) shared by multiple sequences, and we used ad-hoc indexing techniques to minimize the number of sequences to consider for each incoming event.

The general architecture of T-Rex is presented in Fig. 4. When a new rule is deployed into the system, the `Rule Manager` processes it, generates its corresponding automaton model *M*, and passes *M* to the `Automaton Models` component, which stores the information about sequence models (i.e., states, constraints on transitions, parameters, etc.) as well as the information about inter-sequence relationships.

Notice that a Rule R may include two types of constraints that affect the sequences originating from R: some of them, like the type of the event triggering a transition, can be directly associated to the sequence models defined from R as they do not depend from the events actually captured, others have to be associated to sequence instances. We name the constraints of the first type *static* constraints, while constraints of the second type are *dynamic*. For instance, consider Rule R1, the constraint `Temp.value>45` is static and the same holds for the timing constraint – `Temp` must occur `within 5 min from Smoke`. Conversely, the constraint `Smoke.area=$a` is dynamic, as it depends on the value of the attribute `area` of event `Temp`, which is specific to the sequence being considered and unknown at model creation time. During processing, the `Rule Manager` identifies static constraints and uses them to compile the `Static Index`, which is used to efficiently perform a preliminary type and content-based filtering of incoming events. Since this is the same filtering at the base of traditional publish-subscribe systems, we were able to re-use well known techniques developed by the community working on publish-subscribe. In particular, the `Static Index` implements a counting algorithm as described in (Carzaniga and Wolf, 2003).

When a new event enters the system, it is first put in a FIFO `Queue`. This component is used to avoid the loss of input events due to temporary bursts. Administrators can choose the maximum size of the queue according to their needs: having a long queue decreases the probability of losing events, but increases latency. When the next event exits the queue, the `Static Index` identifies which sequence models and which states it may influence. If none is influenced, the event is immediately discarded; otherwise the event is delivered to the `Sequences` component, which stores all active sequence instances generated for each model. To speed up the access to sequence instances from the related model, the `State Index` maps sequence models to existing instances according to their current state, i.e., for each sequence model *M* and state *S* it maps the instances of *M* waiting in *S*. Once all the sequences affected by an incoming event have been selected, T-Rex performs a further selection by looking at dynamic constraints. In particular: (i) it checks timing constraints, deleting those sequences that violate them, and (ii) it looks at parameters, ignoring those sequences that present non-valid parameter values for the event under processing. All remaining sequences are duplicated and advanced, as described in Section 3. Notice that sequences generated from different models are completely independent from each other and can be safely processed in parallel. Accordingly, T-Rex performs actual processing of sequence instances using a pool of threads, taking advantage of multi-core hardware.

Since a single incoming event may be used in many different sequences, we limit memory usage by avoiding an explicit copy of its content. Instead, each sequence keeps references to the events it used to arrive to its current state, and a single copy of used events is saved in the `Stored Events` data structure. As soon as all sequences using an event *e* are deleted from the system, T-Rex removes *e* from `Stored Events`.

When one or more sequences arrive to their accepting state, they are forwarded to the `Generator` component, which performs several operations: (i) it retrieves the set of events that compose the various sequences it receives; (ii) it uses the information stored into the `Automaton Models` component to combine those sequences that are instances of the same automaton model, as described in Section 3; (iii) it computes the values of the attributes of the new composite event notifications captured by the received sequences.

Finally, composite events are sent to the `Subscription Manager`, which keeps track of the interests (subscriptions) of clients to deliver them the matching events.

External clients communicate with the T-Rex engine through a set of `Adapters`, which expose the interfaces to deploy new rules, and to send and receive events. Up to now we have implemented three different adapters: the first one allows the communication with local clients, running in the same process of the T-Rex system, by mean of direct method calls. It has been used during the testing and evaluation phase of the system, as discussed in Section 5. The second and the third adapters allow the interaction with remote clients written in the C++or Java languages. They implement the marshaling and unmarshaling of messages and TESLA rules, as well as the communication through sockets.

5. Evaluation

Our evaluation had two main goals: (i) comparing T-Rex with other processing systems that could handle some of the rules described in Section 2; (ii) studying the performance of our system in a wide range of scenarios.

5.1. Comparison with a state of the art system

As we will better motivate in Section 6, finding a system having features comparable with those of T-Rex was not a simple task. We finally decided to use Esper (Esper, 2011) for many reasons: it is an enterprise level product, widely used (it includes a commercial version, EsperHA) and mature (we used version 4.0.0); it adopts an expressive language with a rich syntax but also puts great emphasis on efficiency and performance; it is open source and provides extensive documentation; it is embedded in Java, which makes the task of writing and executing tests easier.

All the results discussed below have been collected using a 2.8 GHz AMD Phenom II PC, with 6 cores and 8 GB of DDR3 RAM, running Linux. We use a local client to generate events at a constant rate and to collect results. Using this approach, the interaction with the CEP engine is realized entirely through local method invocations; this choice eliminates the impact of the communication layer on the results we collected and allows us to measure the raw performance of the two CEP engines.

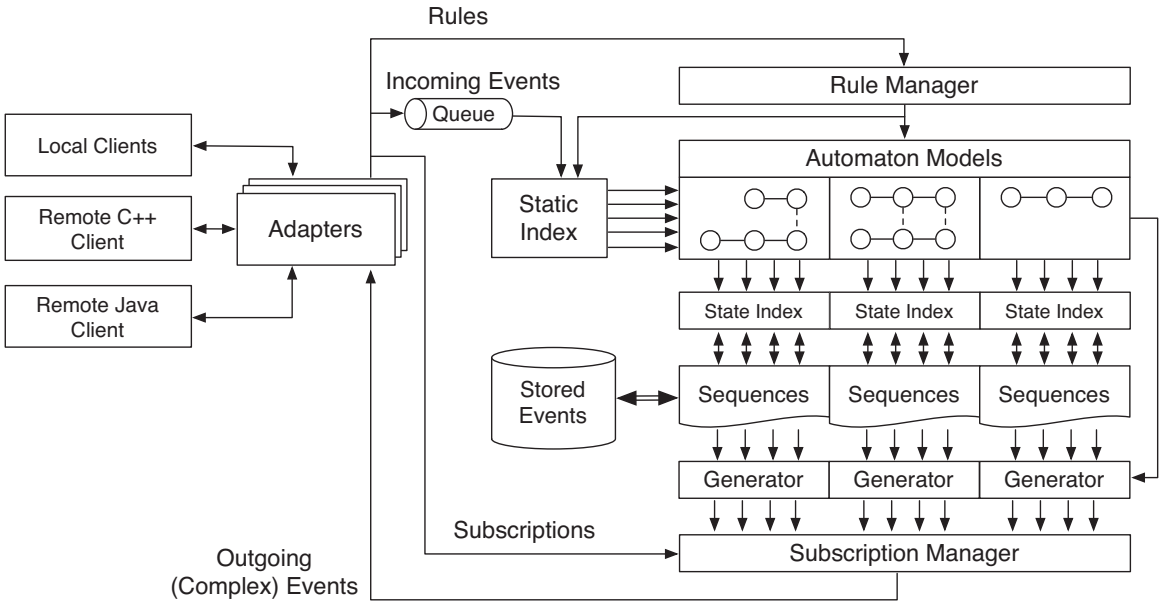

Fig. 4. The architecture of T-Rex.

We set the maximum size of the input queue to 100 events, small enough to emphasize a difference in the processing time through a loss of events (but we also analyze the impact of this factor on the overall system performance).

We studied the behavior of the two systems at different input rates, executing each test 10 times: for each measure we plot the average value and the 95% confidence interval.

5.1.1. *Event filtering*

At the base of a CEP engine is the operation of selecting (filtering) input events before using them to capture complex patterns. For this reason we decided to start our analysis by comparing the filtering capabilities of T-Rex and Esper.

To do so we deployed 1000 different rules on each system. Each rule filters input events according to the value of an attribute and returns a new event containing such a value. On average, each event is selected by a single rule; this means that, ideally, the event output rate should be equal to the input rate.

Fig. 5 shows the performance of the two systems in terms of *output throughput* (simply "throughput" in the remainder of the section), i.e., composite events generated per second. Since the two systems were processing identical loads with the same set of deployed rules, they should theoretically detect the same number of composite events. However, when the input rate increases, both engines drop input events as their input queue becomes full: the rate at which events start to be dropped as well as the number of dropped events and consequently the throughput is a measure of the processing overhead.

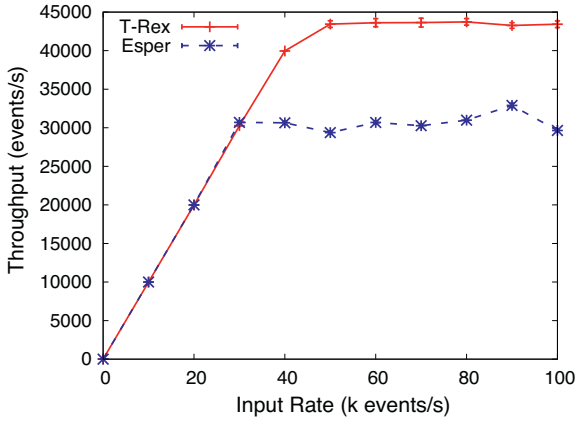

Fig. 5. Filtering: comparison between T-Rex and Esper.

We can observe how both systems can handle high event input rates, in the order of tens of thousands events per second. Esper, however, starts to drop input events at a rate of 30,000 events/s, with a maximum throughput that is below 35,000 composite events/s. T-Rex, instead, can handle up to 40,000 events per second without dropping from the input queue and registers a maximum throughput of about 45,000 composite events/s.

This measure highlights how T-Rex processes events faster than Esper when the deployed rules simply involve filtering of input events. Since the filtering operation does not require the engine to store intermediate partial results, this test isolates the contribution of the `Static Index` component (see Fig. 4) and the counting algorithm it adopts. Differently from T-Rex, Esper uses a tree-based indexed structure to store and evaluate filtering predicates.

5.1.2. *Pattern detection*

As a second step, we wanted to compare the performance of the two engines on detecting patterns of primitive events. This is a key operation for several application scenarios, ranging from environmental monitoring to financial analysis.

To perform this test, we adopt one of the definitions of fire introduced in Section 2. More specifically, we used definition *D1* in both its incarnations, i.e., the one with a multiple selection policy (as modeled by TESLA Rule R1), and the one with a single selection policy (as modeled by TESLA Rule R2). To stress the two systems, each test was performed with 1000 different rules deployed, all having the same structure (that of Rule R1 and Rule R2), but defining 10 different composite events ($\texttt{Fire}_1$, defined from $\texttt{Temp}_1$ and $\texttt{Smoke}_1$,..., $\texttt{Fire}_{10}$, defined from $\texttt{Temp}_{10}$ and $\texttt{Smoke}_{10}$) and asking for a different minimum temperature (from 1 to 100). The value of the temperature of incoming $\texttt{Temp}_x$ events was uniformly distributed between 1 and 100, while all events shared the same `area` to maximize the probability of using events. Given these assumptions, each incoming event was selected by 50 different rules, on average.

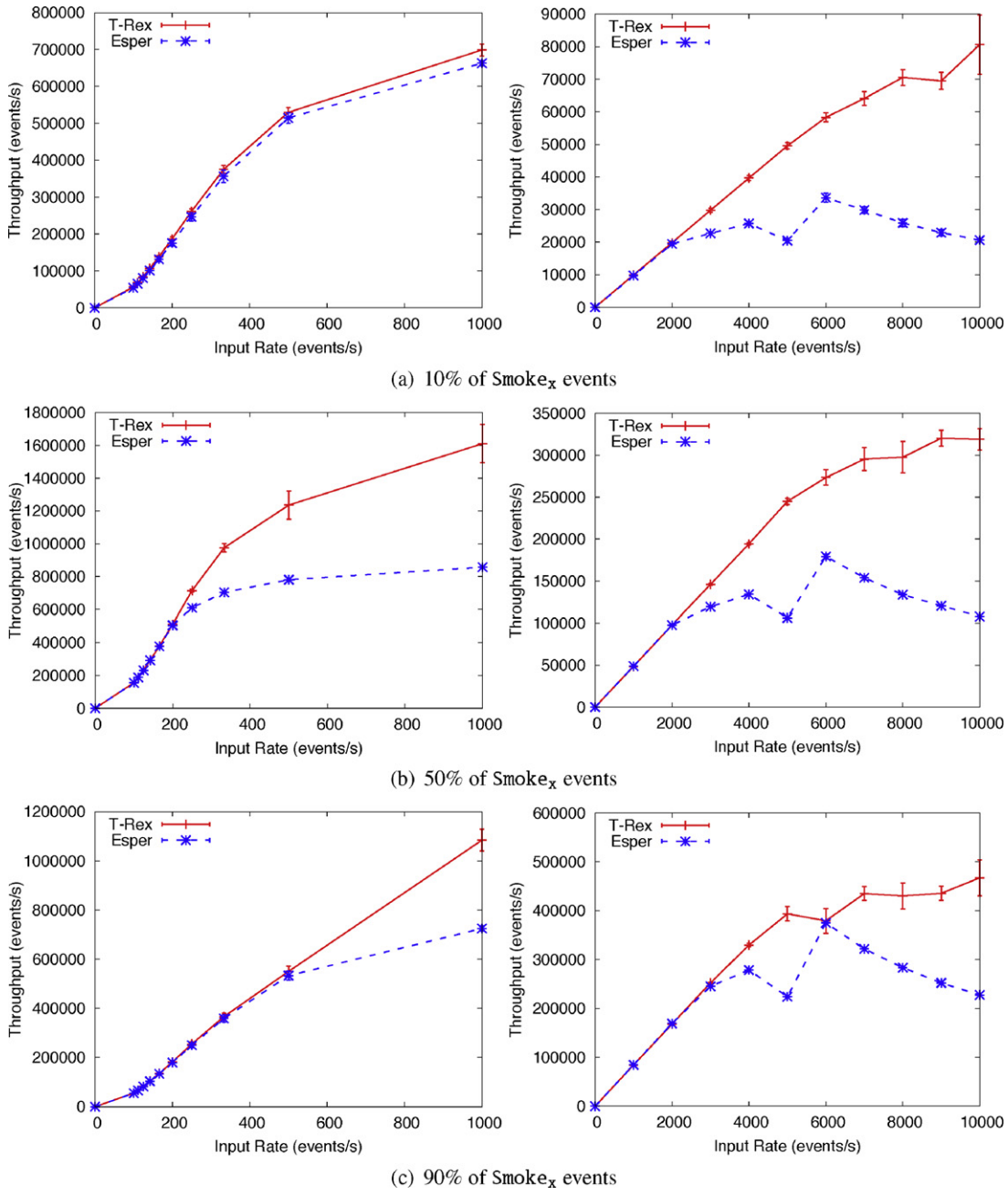

Fig. 6. Comparison between T-Rex and Esper using Rule R1 (left) and R2 (right).

Both when using Rule R1 and Rule R2, we evaluated the engines with three different loads, generating respectively 10%, 50%, and 90% of $\texttt{Smoke}_x$ events (the remaining events are $\texttt{Temp}_x$). Both systems were configured to take advantage of the six cores available on the hosting machine.

Fig. 6 shows the results we obtained in the different scenarios in terms of throughput. The left column includes the results regarding Rule R1 (multiple selection policy), while the right column shows the results for Rule R2 (single selection policy).

In general, we observe that both systems perform very well, with a throughput of hundreds of thousands of composite events detected per second, even if the hardware we used for the tests is entry level for a production CEP system. Even more important from our point of view is that T-Rex outperforms Esper in all scenarios. In particular, T-Rex is capable of processing more input events before starting to drop them from the input queue, and when it drops them it drops less than Esper. The tests also show how the different selection policies associated with Rule R1 and Rule R2 influence the behavior of both engines: Esper performs better when the multiple selection policy is adopted (its performance are closer to those of T-Rex), while the gap between the two engines increases when the single selection policy is adopted. Also notice how the throughput of T-Rex seems to be still increasing even at the highest input rates we were able to generate. The same is not true for Esper: when Rule R2 is adopted (right column) the throughput starts decreasing after the input rate overcomes 6000 events/s; moreover, even at lower rates, it exhibits an irregular trend.

5.1.3. *Aggregates*

The previous experiments considered two important aspects of a CEP system: the capability of filtering input events and that of detecting patterns of events. As shown in Section 2, another

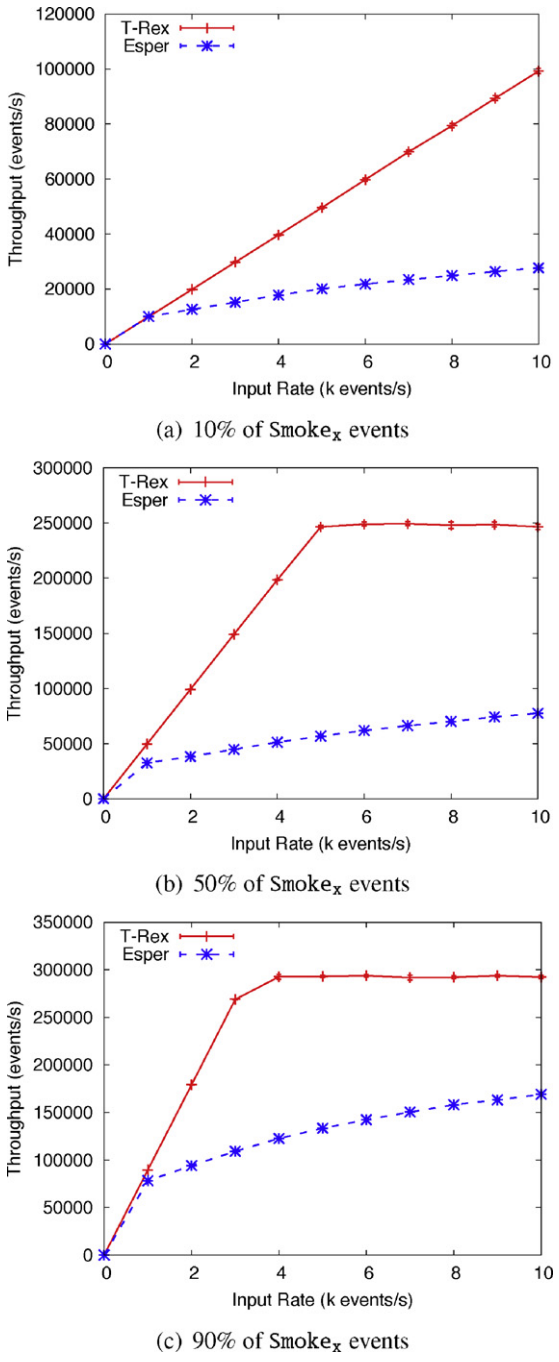

Fig. 7. Comparison between T-Rex and Esper using Rule R5. (a) 10% of $\texttt{Smoke}_x$ events. (b) 50% of $\texttt{Smoke}_x$ events. (c) 90% of $\texttt{Smoke}_x$ events.

important feature for a CEP system is the capability of computing the content of composite events by aggregating the values of attributes in primitive events. To evaluate the performance of T-Rex and Esper in this area we used Rule R5. As in the previous test, we adopted 10 different definitions of $\texttt{Fire}(\texttt{Fire}_1, \ldots, \texttt{Fire}_{10})$ and 100 different thresholds for the value of $\texttt{Temp}_x$, resulting in 1000 different deployed rules. To increase the throughput of the system, the workload was generated to always satisfy the constraints on the values of $\texttt{Temp}_x$: this way every input $\texttt{Smoke}_x$ event led to the generation of 100 $\texttt{Fire}_x$ events. Also in this case we evaluated the engines with three different loads, generating respectively 10%, 50%, and 90% of $\texttt{Smoke}_x$ events.

Fig. 7 shows the results we obtained. If we look at the performance of T-Rex, we observe that it never drops input events when the number of $\texttt{Smoke}_x$ events is small (10%). The same is not true for Esper, which drops events even in this relatively easy case. Things become harder when the percentage of $\texttt{Smoke}_x$ events increases. Both systems drop events as the input rate increases and the throughput flattens, but this phenomenon affects T-Rex much later than Esper, which starts dropping events at a relatively low input rate of 1000 events/s.

5.2. In-depth analysis with synthetic workloads

The direct comparison with Esper demonstrated the efficiency of our system (a research prototype) when compared with an advanced commercial product. As stated at the beginning of this section, we are also interested in studying the behavior of T-Rex under different loads. Accordingly we performed several tests using different synthetic workloads.

We defined a default scenario, in which 1000 rules were deployed in the system, all of them defining a sequence composed by two events. Each incoming event was relevant for exactly 1% of the deployed rules; each rule (and each state inside a rule) had the same probability to select incoming events. The windows between two consecutive events in a sequence were 15 s long on average, ranging uniformly between 14 and 16 s. Starting from this scenario we studied the behavior of T-Rex when changing some parameters. In particular we focused on: (i) the number of events defined in each sequence, (ii) the number of sequences defined in each rule, (iii) the number of rules deployed in the system, (iv) the percentage of events selected by each rule, and (v) the average size of the windows. Moreover, we evaluated the scalability of our system by measuring how performance changes when increasing the number of CPU cores used (unless otherwise stated we use all available cores). Furthermore, we studied the influence of some TESLA constructs, namely negation and consumption, on the performance of T-Rex. Finally, we analyzed the impact of the size of the input queue on the performance of the system.

Since the selection policy used in rules greatly influences the behavior of the system, we ran all experiments twice: once using the multiple selection policy (i.e., the `each-within` operator) and once using the single selection policy (i.e., the `last-within` operator).

During our tests we measured the average latency introduced by T-Rex for processing a single input event. In particular, we measured the time elapsed from the instant when the event starts to be actively processed (i.e. when it exits the queue of input events) to the instant when all sequences affected by that event have been processed, including the time needed to produce new composite events for those sequences that arrived at their final state, if any. This measure is extremely important: first of all, it tells us how each parameter impacts the performance of T-Rex. Second, it tells us which is the maximum input rate T-Rex can handle. For example, if the average processing time is 1 ms, we can theoretically handle up to 1000 input events/s. Notice, however, that considering the average processing time we only obtain a theoretical maximum: single events may take longer to be processes; accordingly, when using a finite input queue, we may start dropping input events before this maximum rate. For this reason, for all our tests we also measure and plot the 99th percentile of the processing time.

To provide more details we also plot the throughput curves for each workload we tested. By comparing the trends in throughput we can learn more about the input rate at which events actually start to be dropped from the finite queue, but also about the kind of rules we are processing: some of them, indeed, generate significantly more output (i.e., composite) events than others.

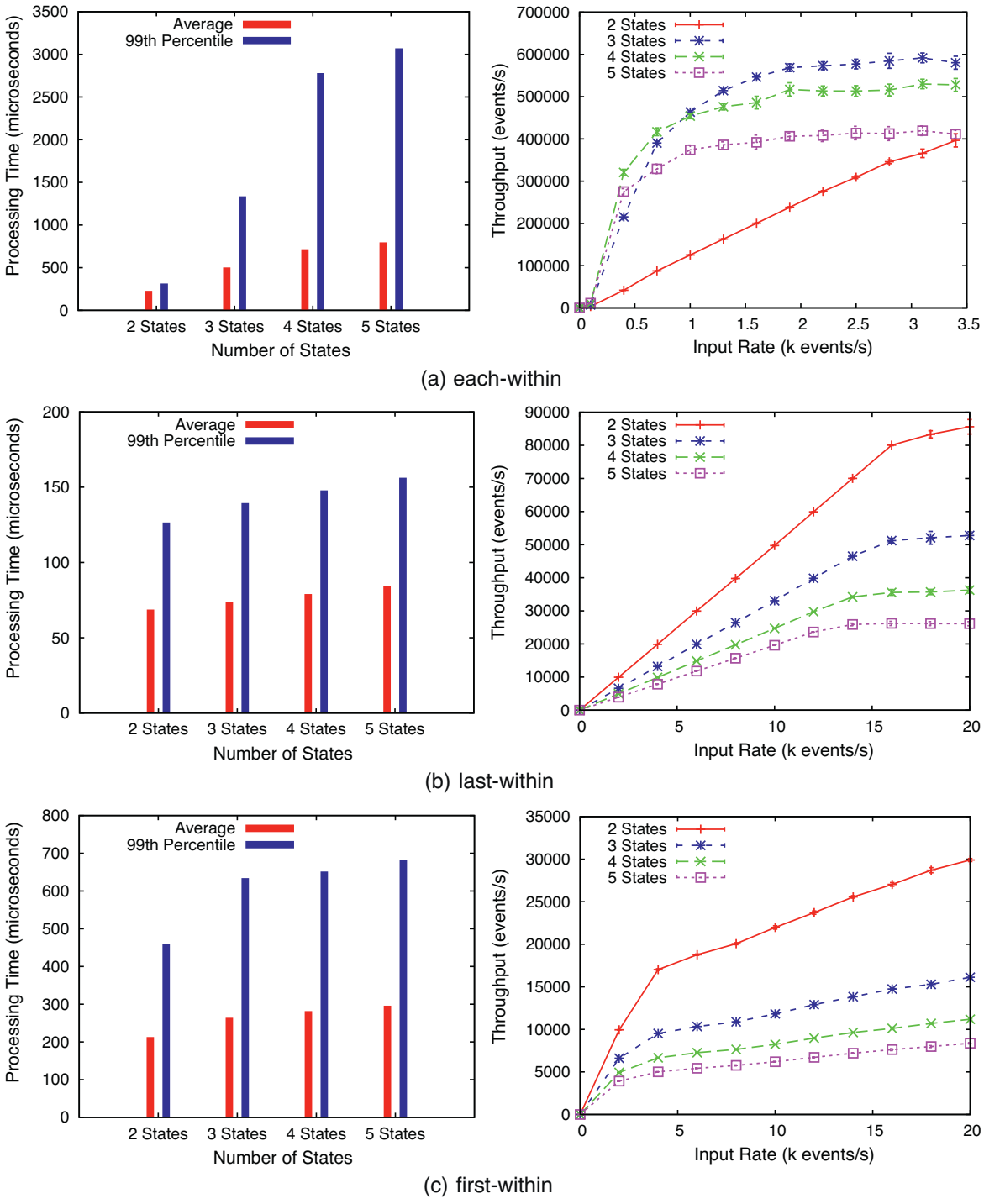

Fig. 8. Number of states in sequences. (a) Each-within, (b) last-within and (c) first-within.

5.2.1. Number of states in sequences

Fig. 8 studies the behavior of T-Rex when changing the number of events captured by each sequence, i.e., the number of states composing each sequence model.

Increasing the number of states in a sequence increases the probability for our algorithm to duplicate sequence instances; accordingly the processing time constantly increases with the number of states. This trend is particularly evident when using a multiple selection policy (Fig. 8a), with the average processing time moving from about 200 μs for processing sequences of 2 states to 800 μs for processing sequences of 5 states. In this case we also observe a higher gap between the average processing and the 99th percentile of the processing time, meaning that increasing the length of sequences also increases the number of events that exhibit a processing time significantly higher w.r.t. the average.

The trend is instead less evident when adopting a `last-within` policy (Fig. 8b) with the average processing time moving from 70 to less than 90 μs. Similarly, the gap between the average and the 99th percentile of the processing time remains constant. This is a result of the optimization described in Section 3 for rules involving a `last-within` operator, which allows T-Rex to minimize duplication of sequences when new events arrive.

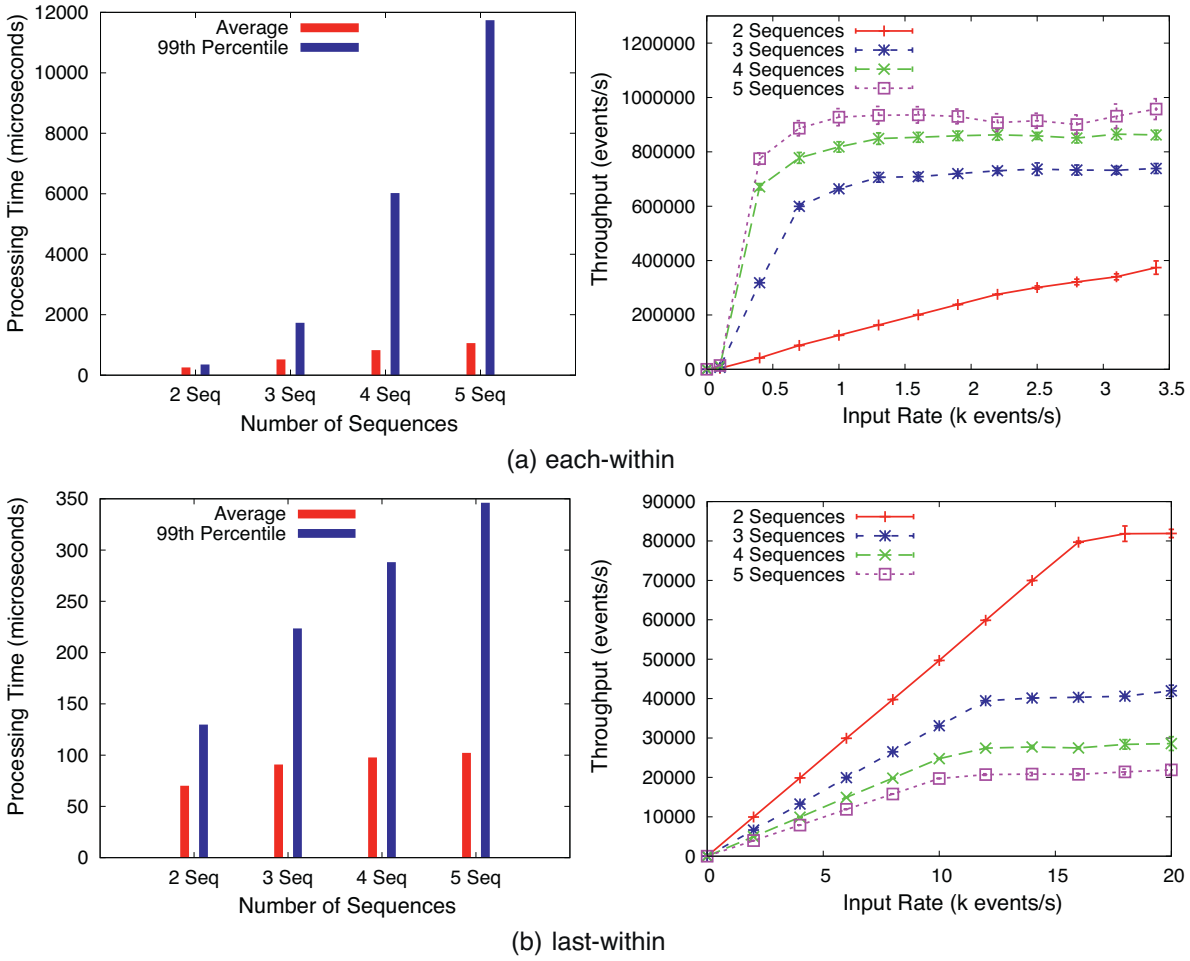

Fig. 9. Number of sequences in rules. (a) Each-within and (b) last-within.

As for the throughput, in presence of a multiple selection policy we expect it to grow with the number of states per sequence, since the number of possible combinations of primitive events increases. This trend is confirmed when the number of states moves from 2 to 3, but when it grows further the throughput decreases. This is an effect of the increased processing overhead, which results in dropping more and more input events, thus negatively affecting throughput. On the other hand, we may notice that using a multiple selection policy with long sequences represents an extreme situation. Indeed it is very hard to find real applications interested in all the possible combinations of long series of events. Such an application, in fact, would need to process a number of composite events (the output of the CEP system, as in Fig. 8a, right) much higher than the number of primitive events captured. On the contrary, we expect applications to use a CEP system to filter out most events, while returning only a small amount of useful information. However, even under this stressing conditions, T-Rex can handle more than 1000 input events/s, producing up to 600,000 composite events.

Different considerations hold when the `last-within` operator (i.e., a single selection policy) is used. In this case the throughput grows linearly until at least 12,000 input events/s. This means that our default queue of 100 elements allows T-Rex to easily handle this large input traffic. We also observe that increasing the length of sequences makes the throughput curve diverge from its linear behavior (i.e., it makes T-Rex drop events) sooner: this is due to the slight increase in processing time. Finally, we notice that the longer the sequences the lower the throughput. This is easily explained by remembering that a single selection policy inhibits duplication of sequences while longer sequences require more input events to arrive to an accepting state.

In this first scenario we also tested the usage of the `first-within` operator (Fig. 8c). Ideally, this case should resemble the one using the `last-within` operator, with a low processing time and the ability of accepting a large number of input events before starting to drop them. However we notice some differences: the processing overhead is not as high as in the case of a multiple selection policy, but it is higher than in the case of the `last-within` operator. This can be explained by remembering what we observed at the end of Section 3: the `last-within` operator, in absence of consumption clauses, allows greater optimization than the `first-within` operator, with less sequences that need to be created and more that can be deleted each time a new event arrives. As a result, the performance of T-Rex when processing rules using the `first-within` operator is somehow in the middle between the performance measured when using the `each-within` and those measured when using the `last-within`. This trend appears in all the scenarios we tested, consequently we will omit the analysis of the `first-within` policy in the following discussions.

5.2.2. Number of sequences in rules

Fig. 9 shows how the system reacts when the number of sequences composing each rule varies. When a multiple selection policy is used (Fig. 9a), we observe a linear increase in the average processing time: indeed, the system needs to perform additional

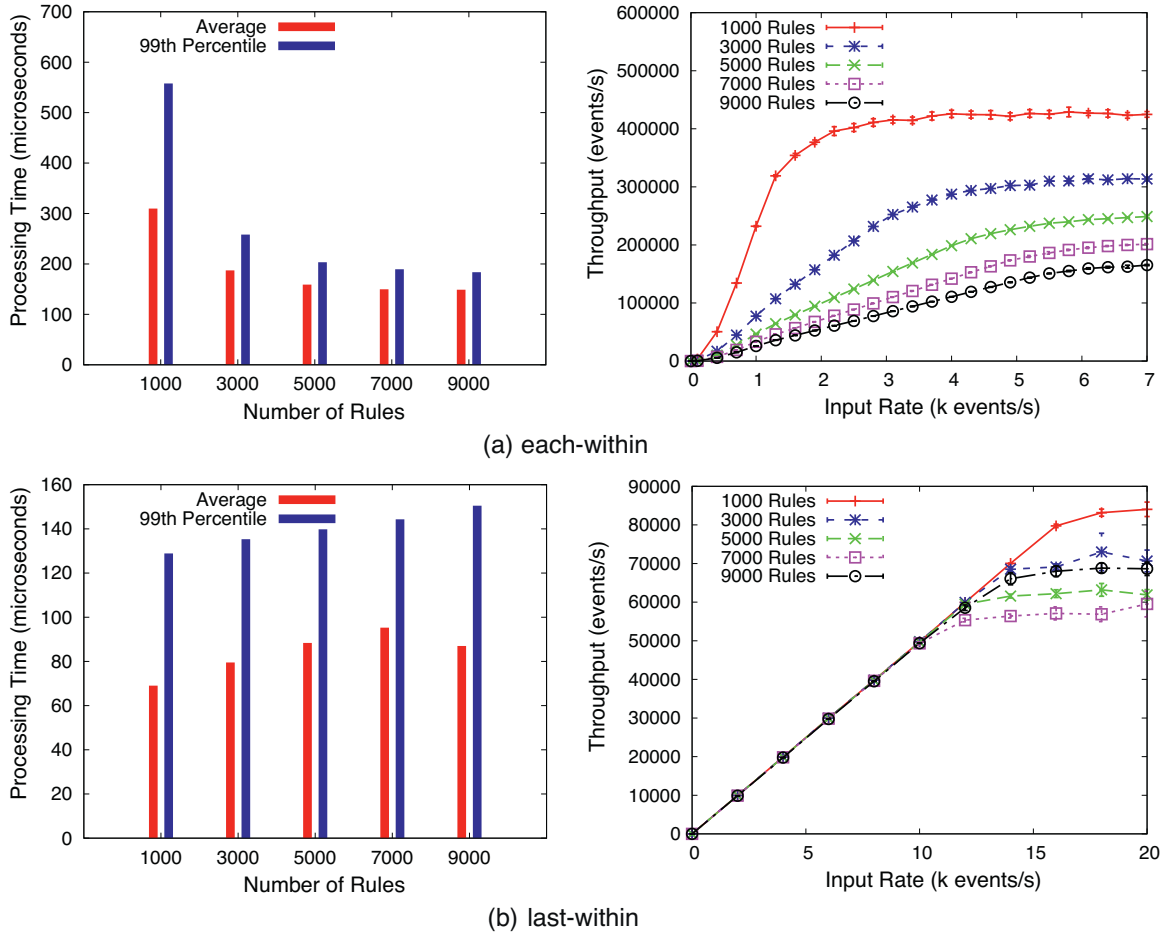

Fig. 10. Number of deployed rules. (a) Each-within and (b) last-within.

work to combine sequences together and to produce the higher number of composite events that derives from this combination. Interestingly, this operation has a huge impact on the 99th percentile of the processing time, which rapidly diverges from the average as the number of sequences increases. This is an indication of spikes in the processing time, which have to be carefully considered to size the input queue in order to limit the number of input events dropped.

By looking at the throughput graph, it becomes even more evident how the number of sequences influences the work to be done: more sequences means more possibility to combine events, and hence more composite events to generate.

The same is not true when a single selection policy is adopted (Fig. 9b). The processing time registers a moderate increase (from about 70 μs to 100 μs, on the average), while the throughput decreases. Indeed, with the same fraction of events captured by each state in a rule, the system keeps about the same number of active sequence instances, independently from the number of sequences composing each rule, but more input events are needed to generate a composite one when the number of sequences per rule increases.

Also in this case, it is important to observe how the use of a multiple selection policy combined with a large number of sequences represents an extreme scenario, which we tested only to stress the system. Even in this case T-Rex can handle about 1000 input events, producing about a million composite events per second. This is a significant result, since not only the processing of input events, but also the generation of composite events (and their internal data structure) demands for computational resources.

5.2.3. Number of deployed rules

Fig. 10 studies how T-Rex reacts when the number of deployed rules increases, but the number of those triggered by each incoming event is fixed to 10. This is an important case since it resembles realistic deployments where a single CEP engine serves different applications, each interested in different kinds of events.

In this scenario, when a multiple selection policy is adopted (see Fig. 10a) the number of generated events decreases with the number of rules; indeed, each rule receives fewer events and thus has fewer chances to combine them into valid sequences. This effect is emphasized by the fact that the rate of events entering each rule decreases with a higher number of rules, and thus the probability of violating timing constraints (and canceling existing sequences) becomes higher. This also affects the average processing time, that slightly decreases when the number of rules increases. At the same time, as processing of rules becomes simpler, the gap between the average and the 99th percentile of the processing time drops.

The same is not true when a single selection policy is adopted (Fig. 10b). In this case, T-Rex does not duplicate sequences per-se, and the effect of a growing number of rules only negatively affects the processing time, with more sequences to consider at each step. In terms of throughput this results in dropping events earlier when more rules are present. Since in our tests each event always triggers the same number of rules (i.e., 10), in absence of dropped events the throughput is not influenced by the number of deployed rules.

5.2.4. Number of triggered rules

Fig. 11 studies the behavior of T-Rex when the number of rules triggered by an incoming event grows. As expected, under these

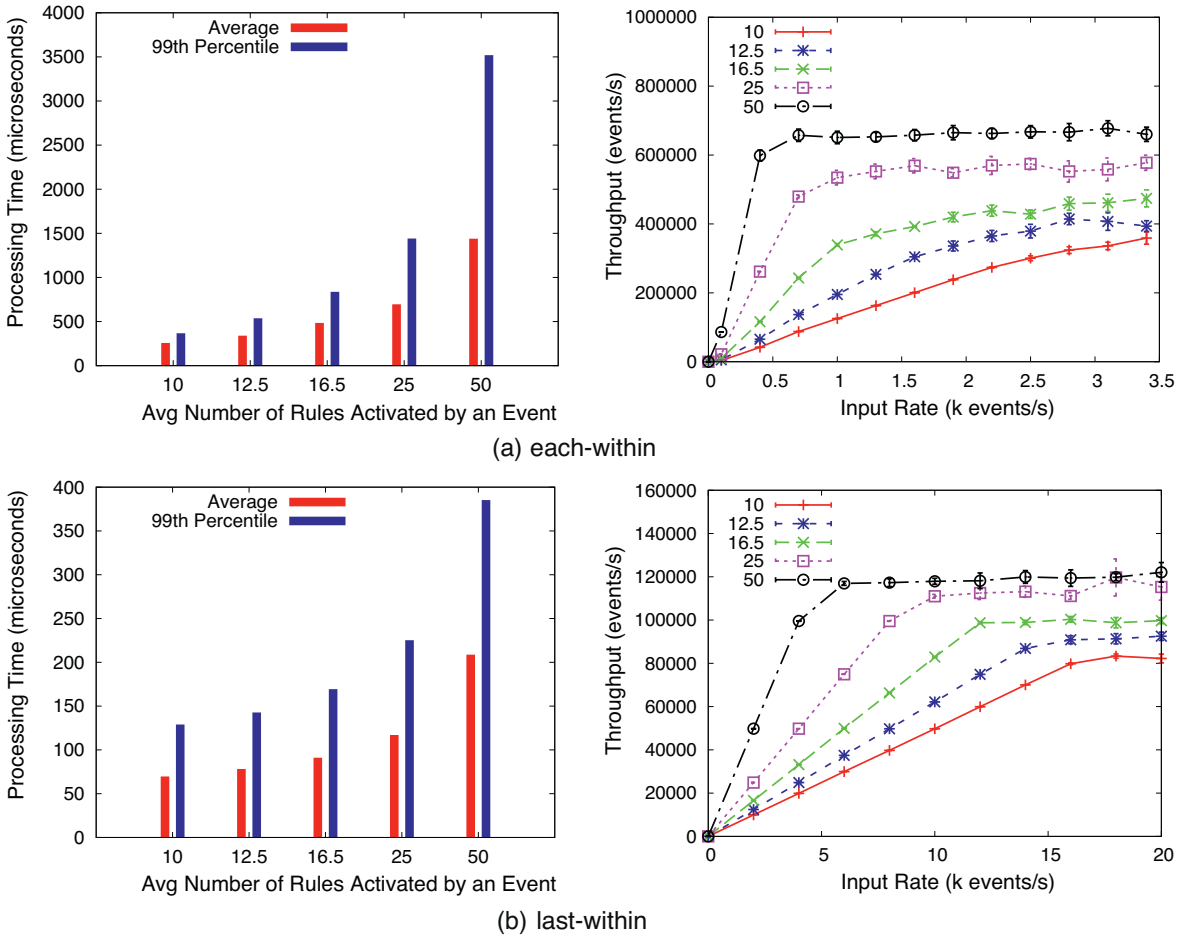

Fig. 11. Number of triggered rules. (a) Each-within and (b) last-within.

conditions the processing time grows, independently from the selection policy adopted. Most importantly, doubling the number of rules involved in the processing of an input event doubles the average processing time as well, meaning that T-Rex does not introduce any additional overhead and scales linearly. As already observed in other scenarios, the gap between the average and the 99th percentile of the processing time increases with the complexity of processing.

Similar considerations hold for the throughput, which grows when the number of rules triggered by each event grows. Notice, once again, that this scenario is explicitly designed to stress the system: the number of output events generated per second is extremely large if compared with the input rate.

5.2.5. Average windows size

Fig. 12 analyzes the impact of the windows size on processing. As expected, when using a multiple selection policy (Fig. 12a) both processing time and throughput grow, since more events enter the window, more sequences are created, and more composite events are generated. Again, scalability is good since the increase in processing time (both its average and its 99th percentile) is less than linear.

Conversely, a single selection policy is not influenced by the windows size (see Fig. 12a). Indeed, to stress the system we are delivering events at a very high rate such that, even with the shortest size of windows considered, there are enough events entering the system to complete detection.

5.2.6. Scalability

Fig. 13 studies the impact of multi-core hardware on the performance of T-Rex. As stated in Section 4, our system makes use of a thread pool to process different rules in parallel. To execute this test we first studied the best number of thread in the thread pool when varying the number of available cores. Then, using this value, we tested the processing time and the throughput with different cores. To limit the number of cores that T-Rex could use, we exploited the *taskset* Linux command. To increase the load of the system, we adopted our default scenario with a multiple selection policy and a low selectivity (each event triggers 100 rules on average). Since the tests were executed on a 6 core machine, our analysis is limited to 5 cores (one is used by the client generating and submitting events).

What we observe is that the use of multiple cores increases the performance of the system. This is particularly evident when moving from 1 to 2 cores, but also moving to 3–5 cores continues to bring benefits. Notice that each thread in the thread pool processes an equal number of rules; however, we cannot know in advance the cost for processing each rule. This implies that different threads may require different times to accomplish their tasks. Since the system waits for all threads to complete, the overall processing time is that of the slowest thread. This explains why performance does not scale linearly with the number of cores. Additionally, the filtering of events is performed using a highly efficient, but sequential counting algorithm. Fig. 13 shows that, when moving from 1 to 5 cores, we reduce the processing time by a half and we almost double the maximum throughput.

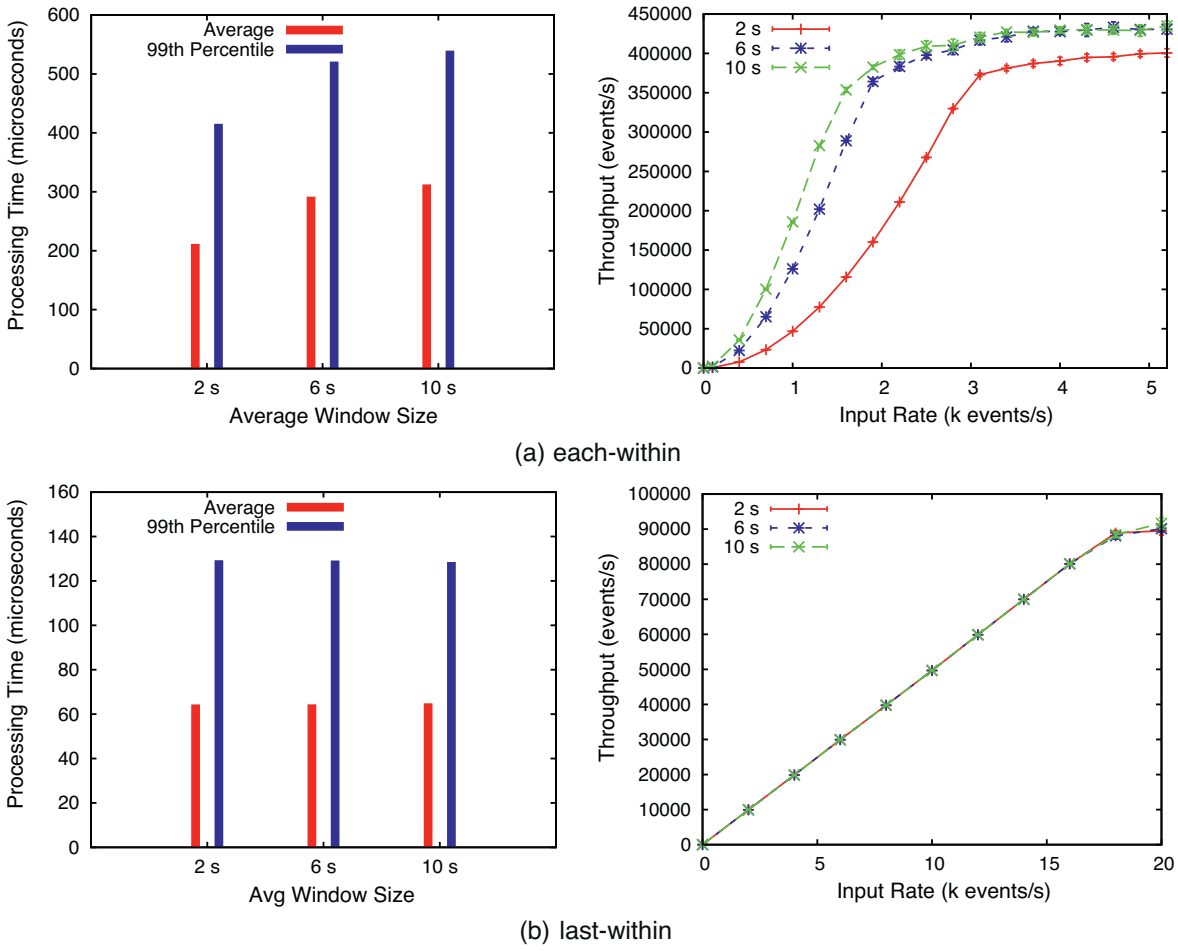

Fig. 12. Average window size. (a) Each-within and (b) last-within.

5.2.7. Use of negation

Fig. 14 investigates the impact of using negation in rules. In particular, we use Rule R3 (see Section 2). To maximize the generation of events we operated as for Rule R1 and R2, deploying 1000 different rules with the structure of Rule R3 but defining 10 different composite events ($\texttt{Fire}_1$, defined from $\texttt{Temp}_1$ and $\texttt{Rain}_1$, ..., $\texttt{Fire}_{10}$, defined from $\texttt{Temp}_{10}$ and $\texttt{Rain}_{10}$) and asking for a different minimum temperature in $\texttt{Temp}_x$ events (from 1 to 100). Then, we studied the behavior of T-Rex when changing the percentage of $\texttt{Rain}_x$ events w.r.t. all primitive events. Since the input rate of events was particularly high, we decided to adopt a small window of 1 s, meaning that $\texttt{Rain}_x$ events become invalid after 1 s.

What we observe is that the processing time is not influenced by the percentage of $\texttt{Rain}_x$ events. Indeed the system stores all $\texttt{Rain}_x$ events received in the last second and checks if any has been stored when receiving a valid $\texttt{Temp}_x$. On the contrary, the percentage of negative events strongly influences the throughput, with fewer composite events captured when the percentage of $\texttt{Rain}_x$ events grows. As a final note we observe how the presence of negation increases the confidence interval, meaning that different runs have

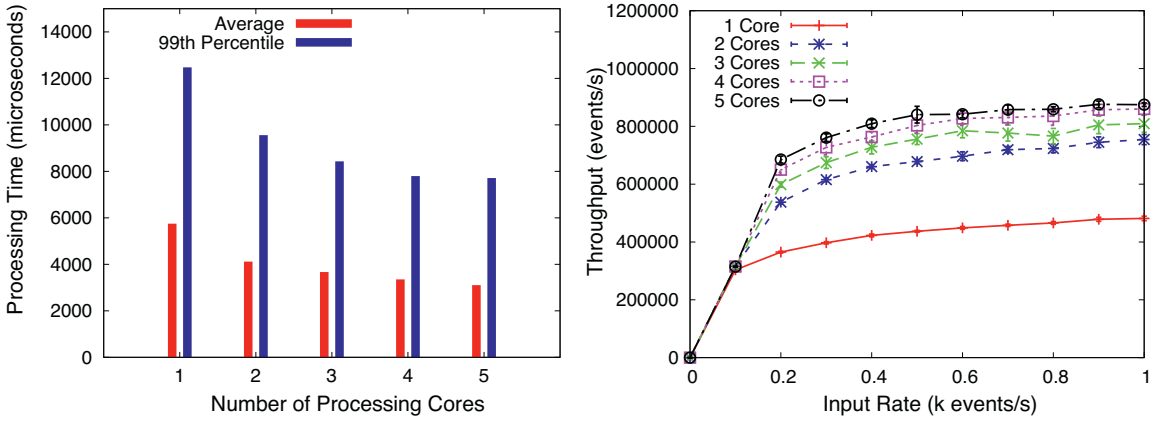

Fig. 13. Scalability.

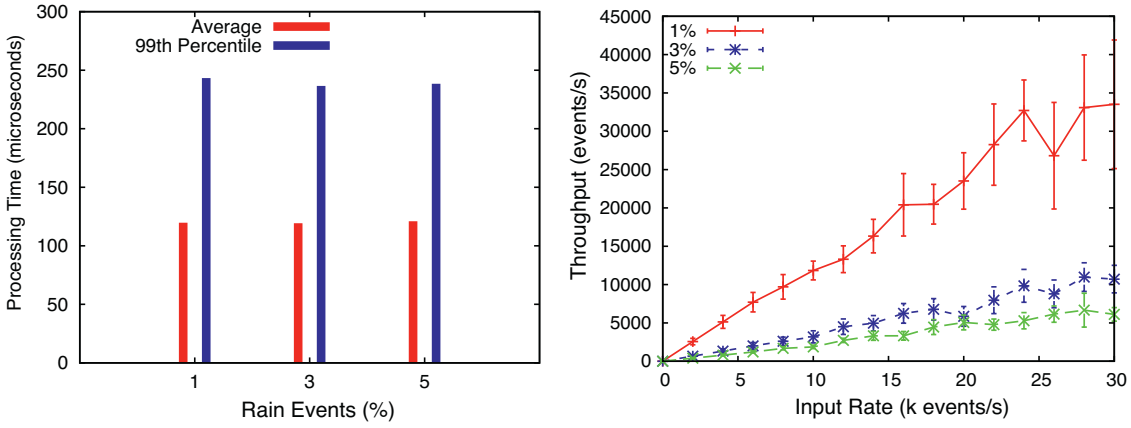

Fig. 14. Use of negation.

a high probability to produce different results. Indeed, when using a negation the order in which events are received can significantly influence the output rate.

5.2.8. Use of the consuming clause

Fig. 15 studies the impact of adding a consuming clause to our default scenario. In particular, since this scenario defines a sequence of two states, we test the performance when asking the consumption of the events participating in the first state.

When adopting a multiple selection policy (Fig. 15a) the processing time decreases if we add the consuming clause. Indeed, in the multiple selection case a large number of automata is generated during processing. Adding a consuming clause helps the system remove some of them, i.e., those using a consumed event. This also contributes in greatly reducing the number of generated events.

When adopting a single selection policy (Fig. 15b), instead, the number of generated events is not seriously affected by the presence of a consuming clause. This is because, even when consuming one or more events, it is highly probable to find another event of the same kind within the required time window that allows the system to produce a composite event. In this case, the consuming clause adds some overhead to the system, which cannot delete sequences as new events arrive, as explained at the end of Section 3. This is also visible in the throughput graph: adding the consuming clause makes the system drop input events earlier.

Finally, we observe how in presence of a consuming clause the gap between the average and the 99th percentile of the processing time significantly increases, both with a single and a multiple selection policy. Indeed, there are cases in which the system needs to iterate over a large number of stored sequences to identify those that include a consumed event.

5.2.9. Size of the input queue

An important parameter to consider when deploying the T-Rex engine is the size of the input queue. A large input queue helps reducing the loss of events in case of temporary spikes in the processing time of individual events or in the input rate.

As we have seen, there are cases in which the 99th percentile of the processing time for a single event significantly diverges from the average, showing that there are (a few) input events that introduce a higher processing latency (e.g., see Fig. 9).

Fig. 16 investigates how the size of the input queue impacts the number of input events dropped, as the input rate increases. We adopt our default scenario, increasing the number of states in each sequence from 2 to 3. Indeed, as shown in Fig. 8 this case presents a higher gap between the average processing time and the 99th percentile, and thus better emphasizes the need of a large input queue.

With a low input rate, both with a single and a multiple selection policy, no event is dropped, independently from the size of the input queue. When the input rate increases, we observe two different behaviors in the case of a multiple and of a single selection policy. In the first case (see Fig. 16a) increasing the size of the input queue reduces the percentage of dropped events. Indeed, with a multiple selection policy, certain input events lead to the generation of a large number of composite events, thus demanding for higher processing times: a larger size of the input queue reduces the impact of these processing delays. However, the difference in the percentage of packets dropped when moving from a size 10 to a size of 10,000 is relatively small (about 10%). In the case of a single selection policy (see Fig. 16b) the processing times are more uniform. Accordingly, increasing the size of the input queue has almost no impact on the percentage of dropped events (about 3% moving from a size of 10 to a size of 10,000).

The second reason for adopting a large size for the input queue is to tolerate temporary spikes in the rate of input events. To test the benefits of the input queue under this kind of workloads, we modified the behavior of the event generator and repeated the test above. In particular, in this experiment we submit events in spikes of k events each, waiting 1 s between a spike and the following one. During spikes we deliver one event every 10 μs, much more than the maximum rate T-Rex can handle. We then measure the number of events dropped from the input queue as k moves from 10 to 10,000.

In this case we observe a significant influence of the size of the input queue on the percentage of events dropped, both using a single selection policy and a multiple selection policy. Finally, if we consider the selection policies, we observe how the reduced processing time required when a single selection is adopted, allows T-Rex to drop less events with rules that use the `last-within` operator (see Fig. 17b).

5.2.10. Discussion

The use of synthetic workloads allowed us to perform a broad analysis of T-Rex, taking into account a large number of parameters that influence its performance. We provide here some conclusive remarks on the most significant findings.

First of all, the shape of the rules deployed into the system has a great impact on the maximum throughput of composite events produced, which varies from a few thousands events per second to more than one million in the scenarios we tested.

As for the processing times, they are significantly lower when a single selection policy is used inside rules, which we expect to be

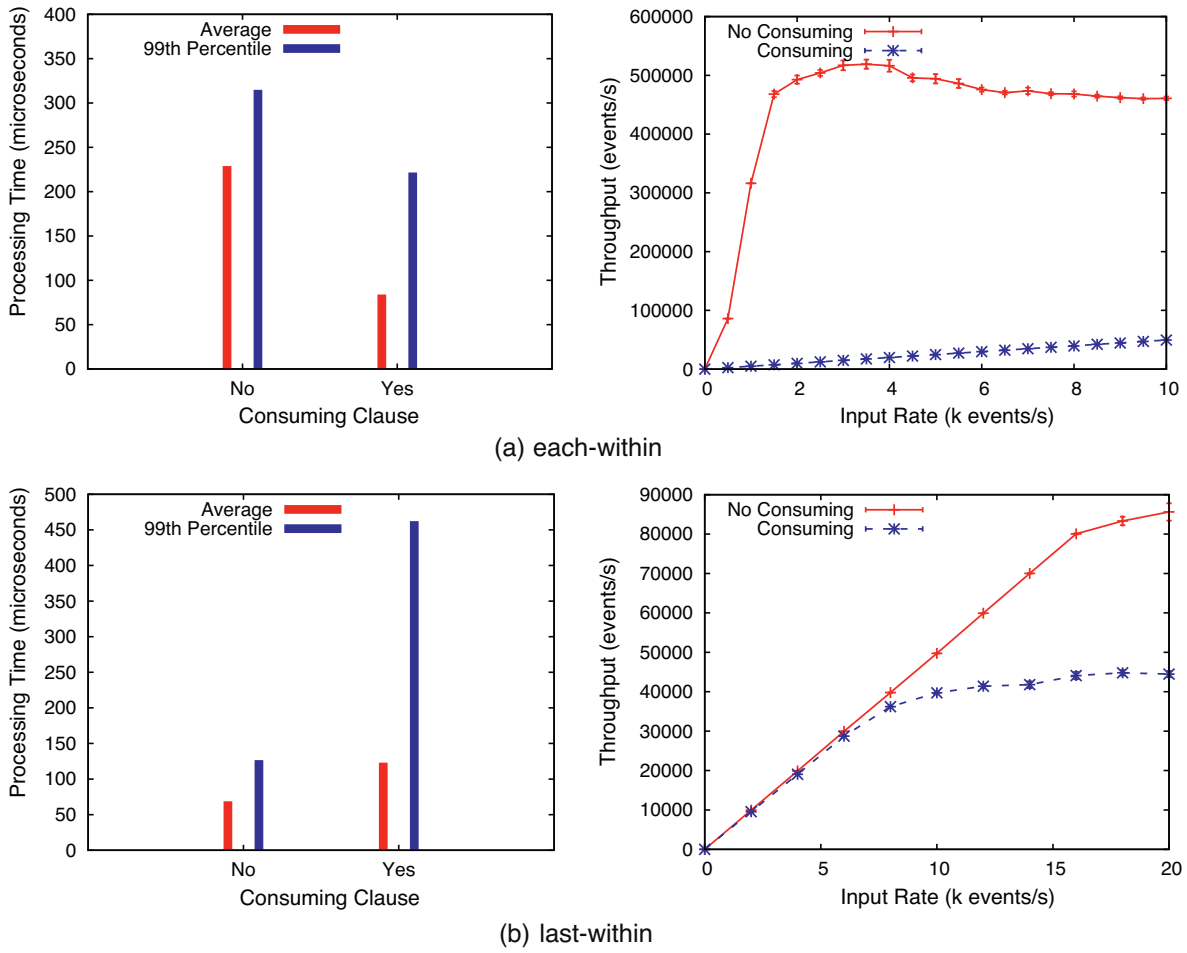

Fig. 15. Use of the consuming clause. (a) Each-within and (b) last-within.

a common scenario for many event-based applications. Moreover, most of the parameters we tested have a more visible impact on the performance of the system when a multiple selection policy is used. This happens, for example, when increasing the length of sequences (Fig. 8), the number of sequences defined in each rule (Fig. 9), or the number of rules triggered by each primitive event (Fig. 11).

Our tests also show that T-Rex scales well with the number of deployed rules and with the number of processing cores, showing constantly increasing performance when moving from 1 to 5 cores.

Finally, we studied how the size of the input queue impacts on the completeness of the results generated. This parameter is of primary importance, since it must be tuned by the system administrator to meet the specific application workload and requirements.

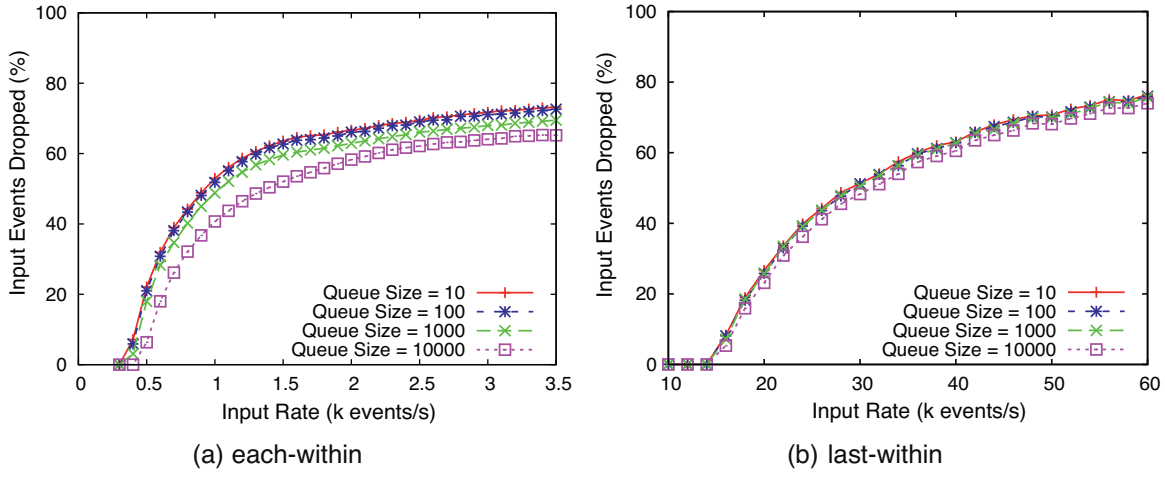

Fig. 16. Size of the input queue. (a) Each-within and (b) last-within.

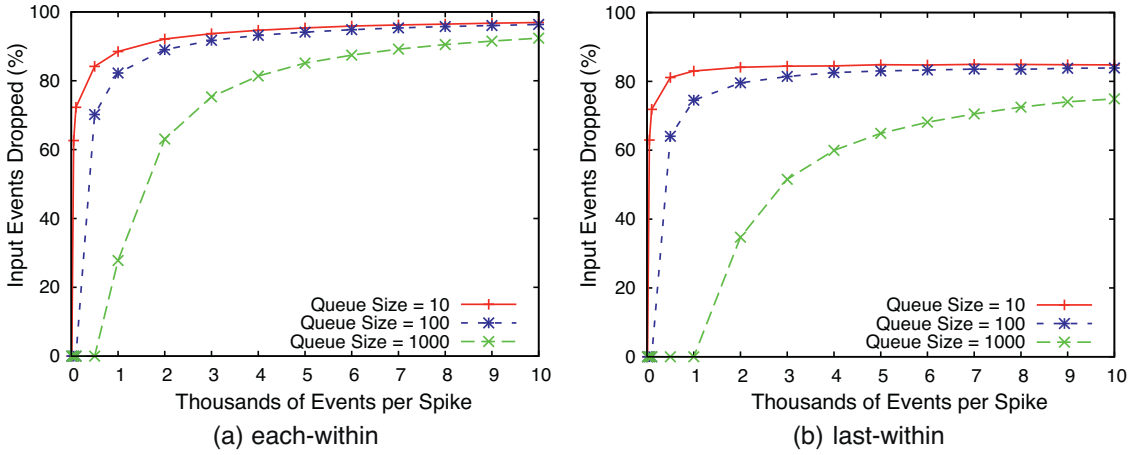

Fig. 17. Tolerance to spikes. (a) Each-within and (b) last-within.

We have seen that the size of the input queue has only a marginal impact when the input rate of events remains almost constant, while it becomes relevant to tolerate spikes in the input rate.

6. Related work

The last few years have seen an increasing interest around Complex Event Processing, with several CEP systems (Luckham, 2001; Etzion and Niblett, 2010) being proposed both from academia and industry. The interested reader can find a detailed study of the field in (Cugola and Margara, 2011), where we analyze and compare in great detail more than 35 systems. Despite all existing solutions have been designed to accomplish the same goal, i.e., to timely process large amount of flowing data, they present different data models and rule definition languages, as well as processing algorithms and system architectures.

6.1. Data models and rule definition languages

The data model determines the way each system models and interprets incoming information: some systems are explicitly designed to capture and process event notifications (Mühl et al., 2006), while others consider generic information streams. The latter are usually known as *Data Stream Management Systems (DSMSs)* (Babcock et al., 2002), a term coined in the database community as opposed to *DataBase Management Systems (DBMSs)*. While DBMSs need data to be stored and indexed before processing, DSMSs allow data to be processed in streams, as it flows from the sources to the sinks.

In practice, the data model a system adopts significantly affects the structure of the rule definition language it uses: systems designed to deal with event notifications adopt languages focused on the detection of patterns involving (often complex) timing relationships among incoming events. On the other hand, DSMSs usually rely on languages derived from SQL, which specify how incoming information items have to be transformed, i.e., selected, joined together, and modified, to produce one or more output streams.

It is our belief that the data transformation approach adopted by DSMSs is not suited to recognize patterns of incoming items tied together by complex temporal relationships, as those we focus in this paper. To motivate this claim, we consider CQL (Arasu et al., 2006), a language often taken as a reference in the DSMS community. It uses a syntax very similar to SQL where each rule is defined using three types of operators. *Stream-to-Relation* (S2R) operators (also known as *windows*) select a portion of a stream to implicitly create a traditional database table. *Relation-to-Relation* (R2R) operators are mostly standard SQL operators. *Relation-to-Stream* (R2S) operators generate new streams from tables, after data manipulation. Each CQL rule is composed by a S2R operator, one or more R2R operators, and one R2S operator. Consequently, the entire processing happens in relational tables, forgetting the ordering among data. The only mechanism that takes timestamps into consideration, windows, operates separately and in advance with respect to the actual processing. This, together with the fact that no explicit sequencing operator is provided, and that timestamps, if not artificially introduced as part of the tuples' schema, cannot be referenced during R2R processing, makes it very complex and un-natural to capture ordering relationships among data. This limitation is so strong that rules like those capturing the fourth definition of fire in our initial example are impossible to write, while other rules require a formulation that is hard to conceive and to understand. In summary, such languages are 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.

To overcome this limitation, some languages for DSMSs were recently proposed, which extend the expressive power of CQL adding explicit support to patterns. This is the case for the language of Esper (Esper, 2011) and for several other languages developed for commercial systems like StreamBase (Streambase, 2011) and Oracle CEP (Oracle, 2011). With these languages, detecting complex patterns becomes possible, but it remains difficult to support and far from natural. Indeed the general schema of the language remains that of CQL, with pattern detection occurring within the bounds of the relations defined by S2R operators (i.e., windows). In our opinion, this approach makes rules, and in particular selection and consumption policies, difficult to write and understand. Moreover, since windows cut the input flows before pattern detection is applied, the pattern has a limited range of action: certain items may be discarded by windows, without even being considered by the pattern detection process; other items may remain in the window for multiple processing cycles, and consequently take part of pattern detection more than once.

At the opposite side of the spectrum are languages that were explicitly designed to capture composite events from primitive ones (Etzion and Niblett, 2010). They natively consider information flowing into the system as notifications of events occurred in the observed world at a specific time, and they define how composite events result from primitive ones. Accordingly, they are better suited to detect complex temporal patterns of incoming

information as required by CEP applications. Unfortunately, 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 (Brenna et al., 2007), making it impossible to express composite event definitions like (*D1*) and (*D3*) in Section 2. Negations are rarely supported (Li and Jacobsen, 2005; Brenna et al., 2007) or they cannot be expressed through timing constraints (Agrawal et al., 2008) as required by the fire definition (*D2*). Other widespread limitations are the lack of a full-fledged iteration operator (Kleene+(Gyllstrom et al., 2008)), to capture a priory unbounded repetitions of events as in (*D4*), and the lack of processing capabilities for computing aggregates. Finally, none of these languages allows to define the selection and consumption policies rule by rule, as in TESLA.

The languages that present more similarities with TESLA are the language for complex event detection presented in (Pietzuch et al., 2003), Sase+(Gyllstrom et al., 2008; Agrawal et al., 2008), Amit (Adi and Etzion, 2004), and Etalis (Anicic et al., 2011, 2010). In Pietzuch et al. (2003), the authors propose a language based on regular expressions, with the addition of timed operator to express windowed sequences. W.r.t. TESLA, this language does not offer customizable selection and consumption policies, aggregates, and multiple sequences inside a single rule. Similarly, Sase+supports parameterization, negation, and aggregates, while providing a way to express different selection policies. 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, and they cannot specify aggregates. Etalis provides a rich set of operators for specifying how composite events derived from primitive ones. Interestingly, similarly to TESLA, Etalis does not introduce any operator for iterations, but allows the definition of recursive rules. Etalis, however, does not support explicit time-based windows and customizable selection and consumption policies.

Finally, there are commercial systems (Tibco, 2011; Progress-Apama, 2011; Oracle, 2011) that, beside a declarative approach, offer the possibility to specify how incoming information has to be processed in an imperative way, using scripting languages. This maximizes the flexibility of the system, but potentially requires more effort from the user.

6.2. Processing algorithms and system architectures

The language used to specify rules significantly influences the processing algorithms adopted and their performance.

DSMSs usually translate the set of deployed rules into a query plan composed of primitive operators (e.g., selection, projection, join) that transform the input streams into one or more output streams. Different operators are then connected through queues of information items. Moreover, each operator can access *synopses* (Arasu et al., 2003), i.e., state information that may be required for future processing.

Since a large number of operators may be deployed concurrently, it becomes important for DSMSs to define good scheduling policies for allocating operators to available resources: an accurate selection of the operators to be executed can reduce the amount of time spent by data inside queues and increase the system throughput.

Usually, scheduling strategies are defined according to some Quality of Service (QoS) policy, which can be either system defined or user defined (Abadi et al., 2003).

Different systems have proposed different algorithms or implementation techniques to increase the QoS perceived by users. They include sharing of synopses among operators (Arasu et al., 2003), query plan rewriting based on the characteristics of operators or on traffic information (Krishnamurthy et al., 2010), and load shedding techniques (Tatbul et al., 2003; Babcock et al., 2004; Srivastava and Widom, 2004b; Chandrasekaran and Franklin, 2004; Tatbul and Zdonik, 2006; Chi et al., 2005), which deal with system overload and sacrifice the completeness of the output to reduce the load of the system and the latency of processing.

Most of the systems designed to detect patterns of events, as T-Rex, translate rules into automata (Brenna et al., 2007; Agrawal et al., 2008; Li and Jacobsen, 2005; Pietzuch et al., 2003). Even in this area, T-Rex represents an advance with respect to other proposals as it is, to the best of our knowledge, the only one that allows to efficiently capture complex patterns composed of several sequences of events in a single rule (i.e., automaton), including customizable event selection and consumption policies.

In this field, Cayuga (Brenna et al., 2007) is often cited for its processing performance. Similarly to T-Rex, it creates indexes for providing fast access to relevant automata, and duplicates them during processing. Cayuga, however, trades expressiveness for performance, providing an extremely simple rule language, with a small number of operators. Interestingly, this allows Cayuga to take advantage of Multi Query Optimization (MQO) techniques, designed to share automata instances across multiple rules.

Sase (Wu et al., 2006) adopts similar techniques, but relies on non-deterministic automata (NFAs). Similar to some advanced DSMSs, Sase allows the detection of patterns inside time-based windows: for this reason, most of the optimization techniques proposed for this system rely on anticipating window evaluation as much as possible to reduce the number of events to consider during pattern detection. Also, SASE shares with T-Rex the idea of duplicating automata as new events enter the system.

Beside plan-based and automata-based processing algorithms, other techniques have been proposed in the past for pattern detection, mainly in the area of Active DataBase System; they include structures based on trees (Chakravarthy et al., 1994), and on Petri Nets (Gatziu and Dittrich, 1993). Finally, Etalis (Anicic et al., 2011, 2010) translates rules in Prolog, decomposing each rule into simple statements that involve only two events.

A few CEP systems exist, like (Schultz-Møller et al., 2009; Li and Jacobsen, 2005; Rabinovich et al., 2011), which support distributed processing of events to optimize resource usage. In particular, by moving the detection of composite events closer to the sources, these systems allow to minimize the bandwidth usage, while they leverage the availability of different engines to share load. In general, this problem is orthogonal w.r.t. the way the engine operates and in principle the same mechanisms for process distribution proposed there could be used by our T-Rex engine to realize a fully distributed CEP middleware.

7. Conclusions

In this paper we presented T-Rex, a CEP middleware that combines an expressive language, explicitly designed to target the needs of applications that have to cope with event notifications and their relationships, with an efficient engine capable of processing large volume of incoming events in an efficient and effective way.

A comparison with Esper, one of the most widely used commercial solutions for CEP, known for its expressiveness and efficiency, shows how T-Rex performs better in different scenarios. Moreover, we analyzed the behavior of T-Rex in a large number of situations using synthetic workloads, showing how it provides good performance even in extremely challenging scenarios.

The analysis of the behavior of T-Rex in a multi-core machine, presented in Section 5, suggested us to further explore this area to increase the processing performance of our engine. Accordingly, we are currently working on an implementation of the engine that leverage the parallel processing capabilities of modern Graphical Processing Units through the CUDA library (Cugola and Margara, 2012).

At the same time, we are integrating T-Rex with RACED (Cugola and Margara, 2009), our protocol for distributed event processing, to further increase its scalability for widely distributed scenarios.

Acknowledgment

This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom.

References

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

Adi, A., Etzion, O., 2004. Amit—the situation manager. The VLDB Journal 13 (2), 177–203.

Agrawal, J., Diao, Y., Gyllstrom, D., Immerman, N., 2008. Efficient pattern matching over event streams. In: SIGMOD, ACM, New York, NY, USA, pp. 147–160.

Anicic, D., Fodor, P., Rudolph, S., Stuhmer, R., Stojanovic, N., Studer, R., 2010. A rule-based language for complex event processing and reasoning. In: Hitzler, P., Lukasiewicz, T. (Eds.), Web Reasoning and Rule Systems. Vol. 6333 of Lecture Notes in Computer Science. Springer, Berlin/Heidelberg, pp. 42–57.

Anicic, D., Fodor, P., Rudolph, S., Stuhmer, R., Stojanovic, N., Studer, R., 2011. Etalis: Rule-based reasoning in event processing. In: Helmer, S., Poulovassilis, A., Xhafa, F. (Eds.), Reasoning in Event-Based Distributed Systems. Vol. 347 of Studies in Computational Intelligence. Springer, Berlin/Heidelberg, pp. 99–124.

Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Nishizawa, I., Rosenstein, J., Widom, J., 2003. Stream: the stanford stream data manager. IEEE Data Engineering Bulletin 26, 2003.

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

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

Babcock, B., Datar, M., Motwani, R., 2004. Load shedding for aggregation queries over data streams. In: ICDE '04: Proceedings of the 20th International Conference on Data Engineering, IEEE Computer Society, Washington, DC, USA, p. 350.

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

Broda, K., Clark, K., Miller, R., Russo, A., 2009. Sage: a logical agent-based environment monitoring and control system. In: Aml. pp. 112–117.

Carzaniga, A., Wolf, A.L., 2003 Aug. Forwarding in a content-based network. In: Proceedings of ACM SIGCOMM 2003, Karlsruhe, Germany, pp. 163–174.

Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.-K., 1994. Composite events for active databases: Semantics, contexts and detection. In: Proceedings of the 20th International Conference on Very Large Data Bases, VLDB '94. Morgan Kaufmann Publishers Inc., San Francisco, CA. USA, pp. 606–617.

Chandrasekaran, S., Franklin, M., 2004. Remembrance of streams past: overload-sensitive management of archived streams. In: VLDB '04: Proceedings of the Thirtieth International Conference on Very Large Data Bases. VLDB Endowment, pp. 348–359.

Chi, Y., Wang, H., Yu, P.S., 2005. Loadstar: load shedding in data stream mining. In: VLDB '05: Proceedings of the 31st International Conference on Very Large Data Bases. VLDB Endowment, pp. 1302–1305.

Cugola, G., Margara, A., 2009. Raced: an adaptive middleware for complex event detection. In: ARM, ACM, New York, NY, USA, pp. 1–6.

Cugola, G., Margara, A., 2010. Tesla: A formally defined event specification language. In: DEBS, pp. 50–61.

Cugola, G., Margara, A. Processing flows of information: from data stream to complex event processing. ACM Computing Survey, in press.

Cugola, G., Margara, A., 2012. Low latency complex event processing on parallel hardware. Journal of Parallel and Distributed Computing 72 (2), 205–218.

Demers, A.J., Gehrke, J., Hong, M., Riedewald, M., White, W.M., 2006. Towards expressive publish/subscribe systems. In: EDBT, pp. 627–644.

Esper 2011. Esper. http://www.eventzero.com/solutions/environment.aspx.

Etzion, O., Niblett, P., 2010. Event Processing in Action. Manning Publications Co.

Event Zero, 2011. Event zero. http://esper.codehaus.org/.

Gatziu, S., Dittrich, K.R., 1993. Events in an active object-oriented database system. Tech. rep., University of Zurich.

Gyllstrom, D., Agrawal, J., Diao, Y., Immerman, N., 2008. On supporting kleene closure over event streams. In: ICDE, pp. 1391–1393.

Krishnamurthy, S., Franklin, M.J., Davis, J., Farina, D., Golovko, P., Li, A., Thombre, N., 2010. Continuous analytics over discontinuous streams. In: Proceedings of the 2010 International Conference on Management of Data, SIGMOD '10, ACM, New York, NY, USA, pp. 1081–1092.

Li, G., Jacobsen, H.-A., 2005. Composite subscriptions in content-based publish/subscribe systems. In: Middleware. Springer-Verlag New York, Inc., pp. 249–269.

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

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

Oracle CEP, 2011. Oracle cep. http://www.oracle.com/technologies/soa/complex-event-processing.html.

Pietzuch, P.R., Shand, B., Bacon, J., 2003. A framework for event composition in distributed systems. In: Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware. Middleware '03. Springer-Verlag New York, Inc., New York, NY, USA, pp. 62-82. http://dl.acm.org/citation.cfm?id=1515915.1515921.

Progress-Apama, 2011. http://web.progress.com/it-need/complex-event-processing.html (accessed November 2011).

Rabinovich, E., Etzion, O., Gal, A., 2011. Pattern rewriting framework for event processing optimization. In: Proceedings of the 5th ACM International Conference on Distributed Event-Based System, DEBS '11. ACM, New York, NY, USA, pp. 101–112.

Schultz-Møller, N.P., Migliavacca, M., Pietzuch, P.R., 2009. Distributed complex event processing with query rewriting. In: DEBS. pp. 4:1–4:12.

Srivastava, U., Widom, J., 2004. Flexible time management in data stream systems. In: PODS '04. ACM, New York, NY, USA, pp. 263–274.

Srivastava, U., Widom, J., 2004b. Memory-limited execution of windowed stream joins. In: VLDB '04: Proceedings of the Thirtieth International Conference on Very Large Data Bases. VLDB Endowment, pp. 324–335.

Streambase, 2011. Streambase. http://www.streambase.com/.

Tatbul, N., Çetintemel, U., Zdonik, S., Cherniack, M., Stonebraker, M., 2003. Load shedding in a data stream manager. In: VLDB '2003: Proceedings of the 29th International Conference on Very Large Data Bases. VLDB Endowment, pp. 309–320.

Tatbul, N., Zdonik, S., 2006. Window-aware load shedding for aggregation queries over data streams. In: VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases. VLDB Endowment, pp. 799–810.

Tibco, 2011. Tibco BusinessEvents. http://www.tibco.com/software/complex-event-processing/businessevents/default.jsp (accessed November 2011).

Wang, F., Liu, P., 2005. Temporal management of RFID data. In: VLDB. VLDB Endowment, pp. 1128-1139.

Wu, E., Diao, Y., Rizvi, S., 2006. High-performance complex event processing over streams. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. SIGMOD '06, ACM, New York, NY, USA, pp. 407–418.

Gianpaolo Cugola is Associate Professor at Politecnico di Milano where he teaches several courses in the area of Computer Science. In 1998 he received the Prize for Engineering and Technology for his PhD thesis on Software Development Environments, while in 2007 his paper on "The JEDI event-based infrastructure and its application to the development of the OPSS WFMS" was awarded as "Most cited Software Engineering Paper in 2001". He has been involved in several projects financed by the EU commission and by the Italian government. He is co-author of tens of scientific papers published in international journals and conference proceedings. His research interests are in the area of Software Engineering and Distributed Systems. In particular, his current research focuses on middleware technology for largely distributed and highly reconfigurable distributed applications, with a special attention to the issue of Content Based Routing and Complex Event Processing.

Alessandro Margara is currently working as a Post-Doc in the DEEP-SE group at the Dipartimento di Elettronica e Informazione (DEI) of the Politecnico di Milano, Italy. As part of his research, he is currently exploring solutions for bringing Complex Event Processing (CEP) to large scale scenarios, with focus on language abstractions, processing techniques, and communication protocols. His main research interests are in distributed systems, and more specifically in the area of event-based middleware, but also in parallel systems for high performance computing and in programming languages and abstractions.