Learning From the Past: Automated Rule Generation for Complex Event Processing

Alessandro Margara
Dept. of Computer Science
Vrije Universiteit Amsterdam
a.margara@vu.nl

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

Giordano Tamburrelli
Faculty of Informatics
Università della Svizzera Italiana Lugano
giordano.tamburrelli@usi.ch

ABSTRACT

Complex Event Processing (CEP) systems aim at processing large flows of events to discover situations of interest. In CEP, the processing takes place according to user-defined rules, which specify the (causal) relations between the observed events and the phenomena to be detected. We claim that the complexity of writing such rules is a limiting factor for the diffusion of CEP. In this paper, we tackle this problem by introducing iCEP, a novel framework that learns, from historical traces, the hidden causality between the received events and the situations to detect, and uses them to automatically generate CEP rules. The paper introduces three main contributions. It provides a precise definition for the problem of automated CEP rules generation. It discusses a general approach to this research challenge that builds on three fundamental pillars: decomposition into subproblems, modularity of solutions, and ad-hoc learning algorithms. It provides a concrete implementation of this approach, the iCEP framework, and evaluates its precision in a broad range of situations, using both synthetic benchmarks and real traces from a traffic monitoring scenario.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Rule-based databases*; I.2.6 [**Artificial Intelligence**]: Learning

Keywords

Complex Event Processing, Rule Generation, Learning

1. INTRODUCTION

Complex Event Processing (CEP) systems analyze large flows of *primitive events* received from a monitored environment to timely detect situations of interest, or *composite events*. In CEP, the processing takes place according to user-defined *rules*, which specify the relations between the observed events and the phenomena to be detected [19].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS '14, May 26-29, 2014, Mumbai, India.
Copyright 2014 ACM 978-1-4503-2737-4 ...$10.00.

CEP can be applied to several domains: sensor networks for environmental monitoring [13]; payment analysis for fraud detection [46]; financial applications for trend discovery [20]; RFID-based inventory management for anomaly detection [51]. More in general, as observed in [33], the information system of every company could and should be organized around an *event-based core* that acts as a nervous system to guide and control the other sub-systems.

While researchers and practitioners working on CEP focused mainly towards processing efficiency (achieving remarkable results) [12, 2, 17, 18], widespread adoption of CEP technologies depends on a correct and precise modeling of the application domain under analysis using CEP rules. Surprisingly, this aspect has received little attention so far and users are left alone in the hard task of rule definition.

To explain the reason of this difficulty, let us refer to the concrete case of vehicular traffic analysis. Rules must describe the *causal relations* between observations (e.g., the number of cars on a given road, the average speed on a given path) and the situation that must be detected (e.g., traffic congestion). A wide range of factors contribute in defining these causal relations: the events recently received, the information they carry, and their specific order. In many cases, these factors are partially or completely unknown.

To overcome this limitation and wield all the potential hidden in CEP, we need a set of techniques to support users in rule definition. More specifically, we have to identify the conceptual foundations and the key algorithms that allow to *learn*, using available *historical* information, those CEP rules that appropriately express the relevant causal relations of a given domain.

This is exactly the goal of this paper, which contributes to the research on CEP in three ways. First, it provides a precise but general definition of the problem of automated generation of CEP rules, concretely referring to the operators of CEP languages as illustrated in [19]. Second, it discusses a general approach to this research challenge that builds on three fundamental pillars: decomposition into subproblems, modularity of solutions, and ad-hoc learning algorithms. Finally, it provides a concrete implementation of this approach: the iCEP framework.

iCEP analyzes historical traces and learns from them. It adopts a highly modular design, with different components considering different aspects of the rule, such that users may decide, based on their knowledge of the domain, which modules to adopt and which hints to provide to guide the learning process and increase its precision. We evaluate iCEP in a

broad range of situations, using both synthetic benchmarks and real traces from a traffic monitoring scenario.

The rest of the paper is organized as follows. Section 2 introduces the event and rule models we adopt. Section 3 defines the problem of CEP rules learning, decomposes it into subproblems, and proposes a general approach to cope with it. Section 4 provides the details of the design and implementation of the iCEP framework. Section 5 evaluates iCEP using a number of synthetic and real traces. It highlights the accuracy of the framework and discusses potential limitations and improvements. Finally, Section 6 surveys related work and Section 7 provides some conclusive remarks.

2. BACKGROUND

Over the last few years, CEP received increasing attention as a mainstream approach for real-time monitoring and situation detection. By learning and generating rules for CEP we aim at exploiting the results that the research on CEP technologies produced so far: low latency processing of information, scalability in the volume of input events, in the number of event sources, and in the number of rules [17].

Despite the differences in the various CEP engines and rule languages both academia and industry proposed so far, it is possible to identify an abstract event model and a relatively small number of abstract operators that cover the functionalities of most of these systems [19]. In the next paragraphs, we introduce these event model and operators.

2.1 Event Model

We assume that each event notification is characterized by a *type* and a set of *attributes*. The event type defines the number, order, names, and types of the attributes that build the event itself. We also assume that events occur instantaneously. Accordingly, each notification includes a *timestamp*, which represents the time of occurrence of the event it encodes. As an example, the following notification:

```
Temp@10(room=123, value=24.5)
```

captures the fact that the air temperature measured inside room 123 at time 10 was 24.5°C.

2.2 CEP Operators

In most CEP languages, a composite event `CE` is defined using a *pattern* of primitive events. When such a pattern is identified the CEP engine derives that `CE` has occurred and notifies the interested components. For instance, composite event `Fire` could be derived from the presence of smoke and high temperature. When notifications about smoke and high temperature reach the engine it generates a `Fire` event. To describe patterns of events, CEP languages rely on some basic building block, or operators. In this paper we consider the most relevant ones, as envisaged in [19]:

- *Selection* filters relevant event notifications according to the values of their attributes.
- *Conjunction* combines event notifications together.
- *Parameterization* introduces constraints involving the values carried by different event notifications.
- *Sequence* introduces ordering relations among events.
- *Window* defines the maximum timeframe of a pattern.
- *Aggregation* introduces constraints involving some aggregated value.
- *Negation* prescribes the absence of certain events.

We are aware that specific CEP languages may implement additional operators or variations of those discussed above. Nevertheless, by focusing on the operators that are available on most CEP languages, our approach provides users with generic, ready to use rules. If necessary, these rules can be manually tuned to exploit system-specific features (e.g., customizable selection and consumption policies [19]) or additional, not yet supported operators. The interested reader may refer to Section 6 for a more detailed analysis of CEP languages and systems.

For improved readability, here and in the remainder of the paper we use an ad-hoc, simple and intuitive syntax for event patterns, which supports the seven operators above. The patterns that follow exemplify this syntax by introducing a few possible definitions for a `Fire` composite event.

```
Pattern P1
within 5min { Smoke(area=$a) and Temp(area=$a and value>50) }

Pattern P2
within 5min { Smoke() and Temp(value>50) and Wind(speed>20) }
where       { Temp->Smoke, Wind->Smoke }

Pattern P3
within 5min { Smoke() and Avg(Temp.value)>50 and not Rain(mm>2) }
where       { Temp->Smoke }
```

Pattern `P1` uses the *selection* operator to accept only `Temp` notifications whose value exceeds 50; it introduces a *window* of 5 minutes to find both `Smoke` and `Temp` (*conjunction*); finally, it imposes a common value for the `area` attribute in `Smoke` and `Temp` (using the *parameter* `$a`). Pattern `P2` shows how the *sequence* operator can be used to state that both `Temp` and `Wind` must precede `Smoke`. Finally, Pattern `P3` shows an *aggregate* constraint, imposing that the average value of all observed temperatures must be greater than 50; moreover, it introduces a *negation*, stating that `Rain` must not be detected within the window of observation.

3. ON AUTOMATED RULE GENERATION

This section defines our problem in details and discusses the approach we propose for dealing with it.

3.1 Problem Statement

The problem of automated rule generation involves learning the causal relations between primitive and composite events using historical traces. We distinguish between positive traces, in which the composite event occurs, and negative traces, in which the composite event does not occur.

More formally, the problem can be stated as follows. Given a set of event traces Θ, and a composite event `CE` such that, for each event trace $\varepsilon \in \Theta$, either `CE` is the last event in ε (i.e., ε is a *positive trace*) or it is missing from ε (i.e., ε is a *negative trace*), automated rule generation aims at deriving the pattern of primitive events whose occurrence leads to `CE`. As an example, from the three traces below:

```
A@0, A@5, B@10, C@15, CE@15
A@0, A@7, A@10, C@15
A@0, B@5, B@27, C@30, CE@30
```

one could infer that `CE` occurs when:

```
within 5s { B() and C() }
where     { B->C }
```

This is clearly a fictional example. In practice, capturing the many factors that contribute to the occurrence of a composite event requires a very large number of traces.

3.2 The Proposed Approach

To tackle the problem of automated rule generation, we propose an approach that builds on three fundamental pillars: decomposition into sub-problems, modularity of the solution, and ad-hoc learning algorithms.

We observe that the problem of automated rule generation can be decomposed into several learning sub-problems, each one considering a different aspect of the rule to discover. In particular, starting from the abstract CEP operators introduced in Section 2, we identified seven sub-problems: (*i*) determine the relevant timeframe to consider, i.e., the window size; (*ii*) identify the relevant event types and attributes; determine the (*iii*) selection and (*iv*) parameter constraints; (*v*) discover ordering constraints, i.e., sequences; identify (*vi*) aggregate and (*vii*) negation constraints.

We claim that a clear decomposition into sub-problems should not remain at the logical level only, but should drive the design and implementation of a concrete solution, with each sub-problem addressed by a different module. Indeed, a clear separation among these modules (see Section 4.1 and Figure 1) enables them to operate independently from each other, possibly using different strategies and technologies for their implementation. Moreover, it allows for ad-hoc, customized algorithms that address domain specific requirements. Some modules can be even guided or entirely substituted by human experts. Finally, an approach based on independent modules eases the extensibility to other CEP operators that may be introduced in the future.

Starting from this conceptual model, we designed, built, and evaluated different concrete tools for automated rule generation. Our first implementation heavily relied on supervised machine learning [36]. However, these traditional techniques shown some limitations in encoding the operators mentioned above. For example, expressing relations between attributes (i.e., encoding *parameter* constraints) was impossible without significantly hampering performance. To overcome these problems, we designed a novel algorithm based on a key intuition we describe in the next section.

3.3 Toward a New Learning Strategy for Automated Rule Generation

To solve the automated rule generation problem we can start from a relatively simple consideration: both a CEP rule r and an event trace ε can be associated to a set of constraints. These are the constraints *defined* by the rule and *satisfied* by the trace, respectively. As an example, consider rule r_1 characterized by the pattern:

```
within 5s { A() and B() }
```

Intuitively, it defines the set of constraints S_{r_1} :[1]

- `A`: an event of type A must occur;
- `B`: an event of type B must occur;

Similarly, the event trace ε_1`:A@0,B@2,C@3` satisfies the set of constraints S_{ε_1}:

- `A`: an event of type A must occur;
- `B`: an event of type B must occur;
- `C`: an event of type C must occur;
- `A→B`: the event of type A must occur before that of type B;
- `A→C`: the event of type A must occur before that of type C;
- `B→C`: the event of type B must occur before that of type C;

By looking at rules and traces from this viewpoint, we can assert that for each rule r and event trace ε, r fires if and only if $S_r \subseteq S_\varepsilon$.

Given these premises, the problem of learning an unknown rule r boils down to detecting the set of constraints S_r it includes. In this setting, given a positive trace ε, S_ε can be considered as an overconstraining approximation of S_r. In the example above, S_{ε_1} includes all the constraints defined by rule r_1 (`A`, `B`) as well as four additional constraints that are not required to satisfy r_1 (`C`, `A→B`, `A→C`, `B→C`).

To prune these additional constraints, we can look at the set of positive traces collectively. In particular, we can intersect the sets of constraints satisfied by each and every positive trace. For example, let us assume that an additional positive trace ε_2`:A@0,B@3,D@4` exists, which defines the following set of constraints S_{ε_2}:

- `A`: an event of type A must occur;
- `B`: an event of type B must occur;
- `D`: an event of type D must occur;
- `A→B`: the event of type A must occur before that of type B;
- `A→D`: the event of type A must occur before that of type D;
- `B→D`: the event of type B must occur before that of type D;

By intersecting S_{ε_1} with S_{ε_2}, we obtain a more precise approximation of S_{r_1}, which only includes the following three constraints: `A`, `B`, `A→B`. Although this approximation still contains an extra constraint (i.e., `A→B`), it shows the general idea behind our intuition: intersecting the constraints satisfied by positive traces enables to (at least partially) prune the additional constraints not included in the set of constraints defined by the rule, S_{r_1} in our example.

Even if this general idea is conceptually simple, its concrete application presents several challenges when it comes to effectively and efficiently support the CEP operators described in Section 2. Among these challenges, omitted in the example above, we have to learn the size of the window and consider the presence of negations, aggregates, and parameters. Next section shows how we detailed, tuned, and reified this approach into the iCEP framework.

4. THE iCEP RULE LEARNING SYSTEM

This section presents iCEP. Section 4.1 discusses the high level architecture of the tool, while Section 4.2 explores each component of this architecture in depth.

4.1 The iCEP Architecture

The problem decomposition described in Section 3.2 directly reflects on the architecture of iCEP, which consists of seven distinct modules, as shown in Figure 1:

- `Events and Attributes (Ev) Learner`: finds which event types and attributes are relevant for the rule;
- `Window (Win) Learner`: finds the minimal time interval that includes all relevant events;
- `Constraints (Constr) Learner`: finds the constraints that select relevant events based on the values of their attributes;

[1] For the sake of readability, this section considers only constraints on the presence and order of primitive events.

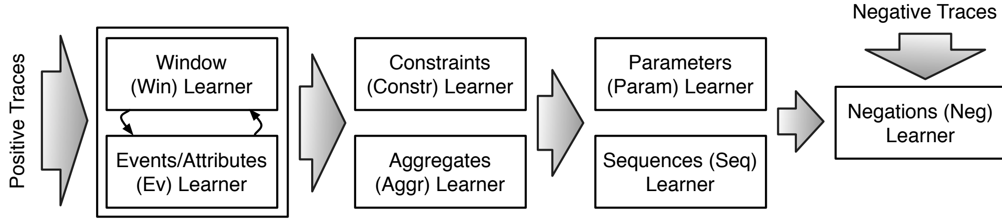

Figure 1: The High-Level Architecture of iCEP

- `Parameters (Param) Learner`: finds the parameters that bind the value of attributes in different events;
- `Sequences (Seq) Learner`: finds the ordering relations that hold among primitive events;
- `Aggregates (Agg) Learner`: identifies the presence and values of aggregate constraints;
- `Negations (Neg) Learner`: finds negation contraints;

As shown in Figure 1, each module takes the set of positive traces available and uses them to determine a specific aspect of the rule to learn. In particular, each module is in charge of learning a specific kind of constraints, leveraging the intuition given in Section 3.3. For instance, the `Ev Learner` looks for the event types and attributes that are relevant for the rule, while the `Constr Learner` finds the constraints on attribute values. Differently from the other modules, the `Neg Learner`, being in charge of learning the events that must *not* occur, also considers negative traces.

Modules operate in cascade as shown by the partially ordered chain depicted in Figure 1. In particular, some modules may proceed in parallel, while others require the results of the previous ones to operate. It is important to notice that the first two modules (the `Win Learner` and `Ev Learner`) require each other's output to perform their computation (this mutual dependency is indicated by the two arrows that join them in Figure 1). In absence of user-provided information that solves this circular dependency iCEP executes the two modules iteratively to learn both the window, the event types, and the relevant attributes for the rule to learn. This iterative process is explained in detail later on in this section.

Organizing modules in cascade provides an additional benefit. Each processing step removes elements from the traces, thus simplifying and speeding up the computation performed in the following steps. As an example, if the `Ev Learner` determines that only the events of types `A` or `B` are relevant for the rule to learn, it can simplify the traces by removing all events with a different type before passing them to the `Constr Learner`.

The modular architecture of iCEP concretely realizes the design pillars discussed in Section 3, by delegating subproblems to specific software components. This enables iCEP users to select and adopt only a subset of the modules mentioned above to integrate their (partial) knowledge of the domain. For instance, if the set of relevant events and attributes is known, it can be explicitly provided to the two modules `Constr Learner` and `Win Learner`, thus eliminating the need for the `Ev Learner`.

As a concluding remark, notice that no module (except for the `Neg Learner`) exploits the information contained in negative traces. All of them implement and refine the conceptual idea of intersecting constraints from all positive traces as described in Section 3.3. While developing iCEP, we tried to improve the performance it provides by also adopting negative traces to discard unrequired constraints, as usually done in traditional machine learning approaches. However, this proved to greatly increase the processing time without yielding significant improvements to the learned rules.

4.2 iCEP Modules Explained

Ev Learner. The role of the `Ev Learner` is to determine which primitive event types are required for the composite event to occur. It considers the size of the window as an optional input parameter. Let us first consider the simple case in which the window size `win` is indeed provided, e.g. by domain experts (the more general case in which the size of window is unknown will be covered in the next section). Under this assumption, the `Ev Learner` can discard events occurred outside the scope of the window. More precisely, it can cut each positive trace such that it ends with the occurrence `Occ` of the composite event to detect and starts `win` time instants before `Occ`.

For each positive trace, the `Ev Learner` simply extracts the set of event types it contains. Then, according to the general intuition described in Section 3.3, it computes and outputs the intersection of all these sets.

Finally, the `Ev Learner` is also responsible for discarding the attributes defined in each event type that are not relevant for the composite event. Indeed, in some application scenarios, it may be possible that event notifications specify a value only for a subset of the attributes defined in their types (leaving others empty). Detecting relevant attributes is conceptually identical to detecting relevant types: the `Ev Learner` simply looks at the set of attributes that are defined at least once in every positive trace.

Win Learner. The `Win Learner` is responsible for learning the size of the window that includes all primitive events relevant for firing a composite event `CE`.

Let us first assume for simplicity that the set of relevant primitive event types (and attributes) S_τ is known. In this case, the `Win Learner` analyzes past occurrences of `CE` and selects the minimum window `win` such that, for every occurrence `Occ` of `CE`, all elements in S_τ are contained in a timeframe ending with `Occ` and having size equal to `win`.

However, as we already observed, it is possible that both the window size and the set S_τ are unknown. This results in the mutual dependency between the `Ev Learner` and the `Win Learner` shown in Figure 1. In absence of external hints from domain experts, we address this dependency by using an iterative approach, which solves the two learning problems at once. In particular, we progressively increase the window size and, for each considered size w_s, we execute the `Ev Learner` assuming such a window w_s as the correct one. In

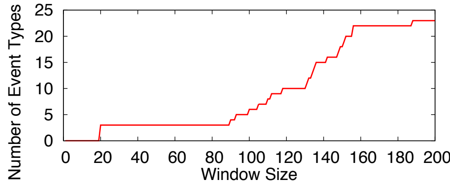

Figure 2: Executing the `Win Learner`. Size of S_τ with different window size. Positive traces generated from a rule requiring three specific event types to appear in a window od size 20.

doing so, we notice that initially the size of S_τ monotonically increases with w_s (as more events enter the window, more event types are detected as relevant), but this behaviour has an end. At a certain point this growth stops and the size of S_τ stabilizes in a plateau. In practice, this happens when w_s reaches the value of the window to learn. This behaviour is exemplified in Figure 2, which shows the size of S_τ during a concrete execution of the `Ev Learner` with a growing window size in a situation where the rule to learn requires three specific event types to appear in a window of size 20. As anticipated, we notice how the size of S_τ shows a plateau when w_s reaches a value of 20. With this window size, S_τ contains exactly the three primitive event types required by the rule.

Starting from this consideration, the algorithm implemented in the `Win Learner` works as follows: it iteratively invokes the `Ev Learner` with a growing window size and selects the first plateau having a size greater than a value p, which is provided as a parameter; the resulting window size is the starting point of such a plateau (20 in our example).

`Constr Learner`. With reference to the general architecture in Figure 1, the `Constr Learner` receives in input the positive traces after they have been filtered by the `Win Learner` and `Ev Learner`. The former resized traces based on the learned window size, while the latter removed irrelevant event types. The goal of the `Constr Learner` is to learn the set of constraints that select the relevant primitive events based on the value of their attributes.

Based on our experience, the `Constr Learner` is a critical component of iCEP since it deals with a learning subproblem characterized by a large solution space. Indeed, depending on the specific application as well as on the situation to detect, the `Constr Learner` should be capable of learning different classes of constraints.

The most elementary case is represented by *equality* constraints, which impose an equality between the value of an attribute a and a constant c. Detecting equality constraints involves detecting which constant values are associated to an attribute a in all positive traces. This can be easily implemented following the core principle described in Section 3.3, i.e., by extracting the set of values for a from all positive traces and computing their intersection.

In addition to equality, for numerical attributes iCEP supports constraints involving *inequality* relations (i.e., $\geq$, $\leq$, $\neq$). As an example, let us consider the following constraint for an event T that predicates on attribute a: $a \geq 0$. Intuitively, inequality constraints cannot be inferred by simply intersecting samples in positive traces. More specifically, learning these constraints involves two tasks: finding the specific relation to use (e.g., $\leq$ or $\geq$) and extracting the value of the constants (e.g., 0 in our previous example). Let us consider the following positive traces:

```
T(a=0)
T(a=2)
T(a=9)
```

First of all, iCEP looks for equality constraints, by intersecting the values of attribute a in the three positive traces. By doing this, iCEP deduces that no equality constraints exist for a. As a second step, iCEP looks for inequality constraints. Without any additional knowledge, iCEP extracts the minimum m and maximum M values for the attribute a in all the traces, and builds a constraint in the form $m \leq a \leq M$. Referring to the example above, iCEP would produce a $0 \leq a \leq 9$.

To improve the precision of this result, iCEP may exploit hints provided by domain experts. For example, they can suggest the kind of relation to be learned ($\geq$ in our example). This hint overwrites the default behavior of iCEP, which in our example would produces the correct constraint $a \geq 0$.

In the previous example we focused on constraints predicating on a single attribute. iCEP supports the general case of constraints involving multiple event attributes (e.g., $a \geq 10$ *and* $b \leq 20$): in this case we rely on existing techniques, i.e., Support Vector Machines [15], to learn the constraints that better describe the portions of the (multidimensional) attribute space observed in positive traces.

`Agg Learner`. As shown in Figure 1, the `Agg Learner` runs in parallel with the `Constr Learner`. The two components have many similarities, since they both are responsible for learning constraints on the content of events. However, differently from the `Constr Learner`, the `Agg Learner` does not consider the values carried by individual events. Instead, it considers the values computed by aggregation functions over all the events of a certaint type.

The `Agg Learner` we implemented natively supports the learning of the following aggregation functions: sum, count, minimum, maximum, and average. It is possible, for example, to learn constraints having the following form: `Avg(Temp.value)>50`, which demands the average `value` computed over all the events of type `Temp` to be greater than 50. To increase the customization and the adaptation of iCEP to different application domains, we also support user-defined aggregation functions.

After computing the values of aggregates in all positive traces, the `Agg Learner` implements the same algorithm already described for the `Constr Learner`. In particular, it first extracts possible aggregate constraints involving an equality relation; if it does not find any of them, it considers constraints involving inequality relations.

`Param Learner`. The `Param Learner` receives in input the positive traces after all events that do not match the content constraints identified by the `Constr Learner` have been removed. Its goal is to extract the parameter constraints among the remaining events. Recalling the definition in Section 2.2, parameters predicate on the relations between the value of attributes belonging to different events. iCEP supports equality as well as, in the case of numerical values, inequality relations.

Following the general principle of intersection among constraints that underpins iCEP, the `Param Learner` operates in two conceptual steps. (i) First, it considers each positive trace ε in isolation and extracts all possible relations among

the attribute of different events appearing in ε. Consider for example the following trace ε_1:

```
A@10(x=1, y=10), B@12(z=1), C@15(w=1)
```

The `Param Learner` extracts the following relations among attributes: `A.x = B.z`, `A.x = C.w`, `B.z = C.w`, `A.y > B.z`, `A.y > C.w`. (*ii*) Second, the `Param Learner` considers all positive traces together, and computes the intersection of the extracted relations among attributes.

Seq Learner. The `Seq Learner` works in parallel with respect to the `Param Learner` and receives the same input from the previous modules. It produces the set of ordering constraints, or sequences, that has to be satisfied to trigger the occurrence of a composite event. Once again, it implements the intersection approach described in Section 3.3. (*i*) First, it considers each trace in isolation and extracts all the ordering constraints among the events appearing in it. As an example, consider again the trace ε_1 illustrated in the previous section. It includes three sequence constraints: A→B, B→C, A→C. (*ii*) Second, the `Seq Learner` intersects all the sequence constraints extracted from individual traces, keeping only those that appear in all traces.

Neg Learner. As shown in Figure 1, the `Neg Learner`, differently from the other modules, also considers negative traces. Indeed, this module is in charge of finding primitive events that must *not* appear in a trace to trigger the occurrence of a composite event. To do so, the `Neg Learner` takes as input the set C of *all* the constraints generated by the other modules in previous steps. Then, it selects the negative traces that satisfy all the constraints in C. In absence of negations, all the traces satisfying the constraints in C should be positive traces, by definition. The fact that the selected traces are negative implies that they contain some additional primitive event e_n that prevented the occurrence of the composite event. In other words, e_n represents the negated event that iCEP needs to learn.

To identify the type and content of e_n, the `Neg Learner` applies the same algorithms adopted by the `Ev Learner` and the `Constr Learner` to the selected negative traces. In particular, it first looks for common event types appearing in all the selected traces, and then extracts the constraints that predicate on their content. The results produced by this are appended to the learned rule as negation constraints.

5. EVALUATION

This section evalutates the accuracy of iCEP in a number of different scenarios, based both on synthetic benchmarks and on real datasets. To enable the replicability of all the results discussed in this section, iCEP is publicly available[2].

5.1 Synthetic Benchmarks

To thoroughly explore the parameters that can influence the behavior of iCEP, we designed and implemented a framework for synthetic benchmarking, which generates events and rules based on a number of controllable parameters. Figure 3 shows the workflow of our synthetic benchmarking framework. First, it randomly generates a *training history* of primitive events. Then, it defines an *oracle* rule R, which we assume to perfectly represent the domain of interest. Next, the benchmarking framework uses R to detect all the composite events in the training history, splitting the training history into a set of positive and negative traces. Finally, it invokes iCEP to analyze positive and negative traces from the training history and learn a rule R^*.

[2] iCEP is written in Java. The source code, the datasets used in our experiments, and the documentation to replicate the experiments are available at http://www.inf.usi.ch/postdoc/margara/software/iCEP.zip

Parameter	Value
Number of event types	25
Distribution of type	Uniform
Number of attributes per event	3
Number of possible values for attributes	100
Number of constraints per event	3
Distr. of op. for select. constraints	= (20%) ≥ (40%) ≤ (40%)
Number of positive traces	1000
Number of primitive events in R	3
Average window size in R	10 s
Number of parameters in R	0
Number of sequence constraints in R	0
Number of aggregates in R	0
Number of negations in R	0
Distance between primitive events	1 s

Table 1: Parameters in the Default Scenario

Metric	Value
Recall (95% confidence interval)	0.9817 (0.0099)
Precision (95% confidence interval)	0.9419 (0.0168)

Table 2: Results of the Default Scenario

To quantitatively measure the performance of iCEP, the benchmarking framework generates an *evaluation history* of primitive events and uses both R and R^* to detect composite events over it. This allows us to measure the *recall* of our algorithms, which is the fraction of composite events captured by R that have been also captured by R^*; and the *precision*, which is the fraction of composite events captured by R^* that actually occurred, i.e., that were also captured by R. Notice that the benchmarking framework stores both rule R and rule R^* for each experiment it executes. This enabled us to study the syntactic differences between the two rules and to isolate the errors introduced by the different learning modules.

Experiment Setup. Several parameters influence the generation of: the rule R, the training history, and the evaluation history. For a broad analysis of this space, we defined a default scenario, whose parameters are listed in Table 1, and we investigated the impact of each parameter separately. Our default scenario considers 25 types of primitive events `T1`...`T25`, each with the same frequency of occurrence. Each primitive event contains three attributes, each one holding an integer value between 0 and 100. Rule R has the following form:

```
within 10 s { T1(c1 and c2 and c3) and
              T2(c4 and c5 and c6) and
              T3(c7 and c8 and c9) }
```

where `c1,...  c9` are elementary constraints on attribute values. In our default scenario, rule R does not include parameters, aggregates, sequences, or negations. Both the training and the evaluation histories include one primitive event per second, on average, which fire 1000 composite events (i.e., positive traces). All our experiments have been repeated ten times, using different seeds to generate random values. For each experiment, we plot the average value of recall and precision, and the 95% confidence interval.

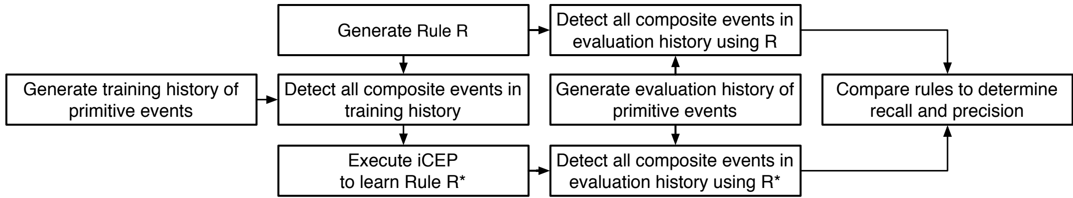

Figure 3: Workflow for Synthetic Benchmarking

Default Scenario. Table 2 shows the results we measured in our default scenario. First of all, the recall is very close to one, meaning that the rules generated by iCEP are capable of capturing almost all the composite events that occur in the evaluation history. Also the precision is above 0.94, meaning that the rule detects about 6% more events than those actually occurred (false positives). A remarkable result. Moreover, the results are stable over multiple runs: for both recall and precision the 95% confidence interval is lower than 2% of the measured value.

By analyzing the differences between the oracle rule R and the learned rule R^*, we observed that iCEP always correctly identifies the types and attributes of the primitive events in R, as well as the window size. Moreover, iCEP correctly identifies the absence of parameters, sequences, and negations. The only (limited) differences between R and R^* are represented by the constraints on the content of attributes, as learned by the `Constr Learner` and `Agg Learner`. In particular, the `Constr Learner` correctly identifies all equality constraints, while it sometimes over or under estimates the value in inequality constraints (i.e., constraints involving the $\leq$ or $\geq$ operators). Similarly, the `Agg Learner` produces aggregate constraints that are not present in rule R, typically introducing a lower (upper) bound for the minimum (maximum) value of an attribute. As explained in Section 4.2, we expect these imprecisions to be mitigated by the presence of hints from domain experts.

Time and Memory Requirements. We consider iCEP as an offline tool, whose results should be used to determine the rules to be subsequently deployed into a CEP engine. However, to provide an idea of the costs of the algorithms we propose, we also report here the time required to analyze the training history and determine rule R^*. Using our reference hardware (a Phenom II X6 1055T @2.8 GHz with 8 GB of RAM), a complete run of our default scenario (including the time to compute the recall and precision) requires less than 35s to conclude. Our experiments show that this time increases at most linearly with the number traces and with the number of events appearing in each trace. Even the most complex tests presented in the remainder of this section required less than ten minutes to run. Finally, we measured a maximum memory consumption of less than 1.5 GB. These numbers allow us to conclude that neither the processing time nor the memory consumption represent a limit for the adoption of iCEP.

Number of Primitive Events. Figure 4 shows how the results produced by iCEP change with the number of primitive events appearing in rule R. First of all, we measured a constant recall, above 0.97, meaning that the learned rule R^* correctly identifies almost all the composite events in the evaluation history, independently from the number of primitive events in R. On the other hand, we measure a drop in precision when R includes only one or two primitive events. Indeed, we observed that the `Constr Learner` usually generates selection constraints that are less selective (i.e., easier to satisfy) than those present in the oracle rule R. This results in producing false positives, i.e., in generating more composite events than those actually occurring in the evaluation history. As Figure 4 shows, the impact of this evaluation error decreases with the number of primitive events. Indeed, selecting a wrong event has fewer chances to trigger a false occurrence when other events are required by the rule.

Size of Window. Figure 5 shows how the size of the window in R impacts on the performance of iCEP. Also in this case, the recall remains almost constant. Conversely, the precision slightly decreases with the window's size. This is not a consequence of an error in learning the window size itself. Indeed, the `Win Learner` always identifies the correct size of window. On the other hand, a large window has a negative impact on the `EV Learner`. This can be explained by noticing that a large window w will include a large number of events, thus increasing the chances of finding events in w that appear in all positive traces and become part of the learned rule R^* even if they are not part of R.

Number of Event Types. Figure 6 analyzes the recall and precision of iCEP while changing the overall number of event types appearing in the training and in the evaluation histories. Since types are uniformly distributed, the frequency of occurrence of each type `Ti` decreases with their number. Thus, as the number of different types decreases, it becomes more and more difficult for iCEP (and in particular for the `Ev Learner`) to discriminate between relevant and irrelevant types for the rule R. This explains the behavior of Figure 6: with a small number of types (less than five) we measure a recall of 0.71. However, this value rapidly increases, up to 0.98, with seven or more types.

Number of Selection Constraints. This section analyzes how the results of iCEP change when increasing the number of selection constraints for each primitive event appearing in R. As Figure 7 shows, the number of constrains has virtually no impact on the recall. On the other hand, a small number of constraints (less than three) negatively impacts on the precision. We already observed a similar behavior in Figure 4: when rule R becomes less selective the difficulty of the `Constr Learner` in predicting the correct selection constraints become more evident.

Number of Parameters. This section explores the accuracy of the `Param Learner` in detecting parameter constraints. As shown in Figure 8, introducing parameter constraints does not introduce any visible impact on precision (constantly above 0.94) and on recall (constantly above 0.96).

Number of Sequences. This section analyses the impact

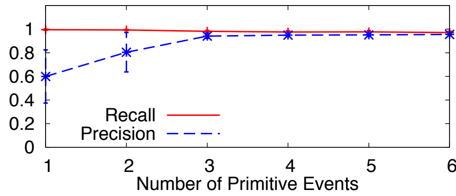

Figure 4: Number of Primitive Events in R

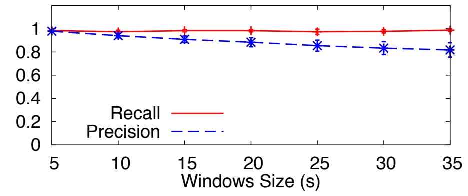

Figure 5: Size of Window in R

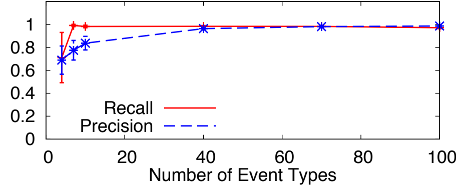

Figure 6: Number of Event Types

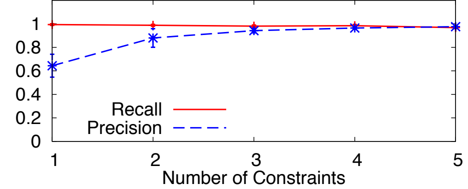

Figure 7: Number of Selection Constraints in R

of sequence constraints on iCEP. To do so, we repeated the experiment shown in Figure 4, by measuring the recall and precision of iCEP while increasing the number of primitive events involved in rule R. However, in this case we also introduced sequence constraints, forcing all the primitive events in R to appear in a precise and predefined order (i.e., in a sequence). Figure 9 shows the results we measured: both precision and recall are almost identical to those measured in Figure 4 (without ordering constraints). iCEP, and in particular the `Seq Learner`, was always capable to identify sequence constraints correctly.

Number of Aggregates. This section explores the behavior of iCEP in presence of aggregates. We considered five different aggregation functions: minimum, maximum, average, count, and sum. Figure 10 shows the results we measured. As in most previous experiments, the recall remains constant and above 0.98. However, we register a drop in precision, which moves from 0.94 (with no aggregate constraints) to 0.78 (with one aggregate constraint). This is due to an inaccurate detection of aggregate constraints. In particular, in presence of an aggregate constraint involving type `Ti`, the `Ev Learner` always recognizes that `Ti` is required to fire the occurrence of a composite event. In some cases, however, the `Agg Learner` fails to recognize the presence of an aggregate constraints involving `Ti`, or, more commonly, generates an under-constraining aggregate. Nevertheless, precision remains constant as the number of aggregates increases.

Presence of Negations. This section investigates the impact of negations on the accuracy of iCEP. As in the case of aggregates, we started from our default scenario and we added one or more negation constraints, each of them predicating over a different event type, and including three selection constraints. Figure 11 shows the results we measured. When moving from zero to one negation constraint, we notice that the precision remains almost constant, while the recall decreases from 0.98 to 0.85. We already observed in previous experiments that the selection constraints generated by iCEP are usually less selective than the ones in the oracle rule R. This also occurs when estimating the selection constraints for the negated event: iCEP correctly identifies the type and attributes of the negation, but it generates selection constraints that capture more events than required, thus reducing the recall.

More interesting is the behavior of iCEP in presence of two negations. In this case, the presence of *at least one* negated event is sufficient to prevent the firing of a composite event. Because of this, the algorithm implemented in the `Neg Learner`, which extracts common elements from *all* negative traces, leads to a lower accuracy. Indeed, in presence of two negated primitive event types, `Tn1` and `Tn2`, some negative traces may include only `Tn1`, while others may include only `Tn2`; thus, the intersection of all negative traces may yield to an empty set. This is what happened in our experiment: the `Neg Learner` fails to correctly identify negation constraints when more than one event is negated. In such cases the precision drops below 0.5. As a positive side effect, the recall returns to 0.98. We further discuss this issue in Section 5.3.

Number of Traces. Finally, this section investigates how the performance of iCEP changes with the number of positive traces available (i.e., composite events). This is a crucial parameter for a learning algorithm. Moreover, in some real situations, composite events can occur quite infrequently (e.g., monitoring of an exceptional system's behavior), and collecting a large number of positive traces can be difficult.

As expected, by looking at Figure 12, we observe that a very small number of positive traces (below 40) results in poor recall and precision. In this situation, almost all the modules of iCEP provide inaccurate results. However, 40 positive traces are already sufficient to overcome a value of 0.9 in both precision and recall. With 100 positive traces, they both overcome 0.95. After this point, iCEP provides stable results.

5.2 Traffic Monitoring Scenario

The use of synthetic benchmarks enabled us to extensively investigate the accuracy of iCEP. To shed light on the concrete applicability of iCEP, we applied it to a real world dataset including timestamped information about the position and status of buses in the city of Dublin[3]. We used the dataset to generate primitive events about the traffic. In particular, we defined a different event type for each bus line (75 in total), where each type defines five attributes including the bus number and the delay w.r.t. schedule, which are all part of the dataset. Each bus approximately generates

[3] `http://dublinked.com/datastore/datasets/dataset-304.php`

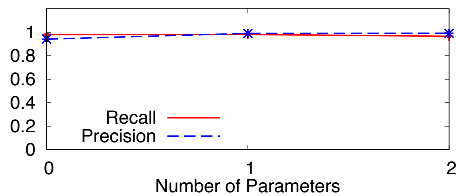

Figure 8: Number of Parameter Constraints in R

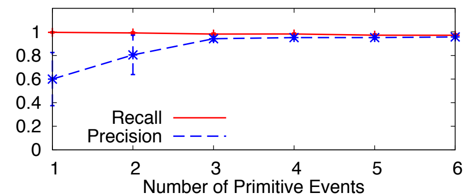

Figure 9: Presence of Sequence Constraints in R

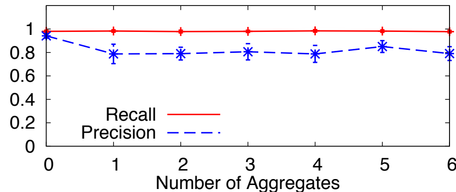

Figure 10: Number of Aggregate Constraints in R

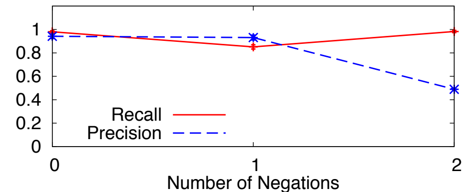

Figure 11: Number of Negation Constraints in R

an update every two seconds; thus, every minute contains several events of the same type. This is a very challenging scenario for iCEP, since even windows of limited size are likely to include multiple occurrences for all event types.

To validate iCEP, we defined a rule R similar to the rule illustrated in the default scenario. In particular, the rule fires a composite event `Alert` whenever at least one bus from n different (and pre-defined) lines has more than one minute of delay. We evaluate the accuracy of iCEP by progressively increasing the number of lines n involved in the rule. We extracted from the dataset information concerning two different days of operation. We use the first day to train iCEP and the second one to evaluate the precision and recall of the learned rule R^*. Figure 13 shows the results we measured. We obtained a very high recall and precision, independently from the number of event types (i.e., bus lines) appearing in R. Nevertheless, if we look at the learned rule R^*, we observe some differences w.r.t. R, with the former often including additional event types not contained in R. Indeed, the high density of events made it difficult for iCEP to discriminate between relevant and non-relevant event types. On the other hand, this does not impact the ability of the learned rule to detect all and only those composite events that appear during the second day (the evaluation history).

5.3 Discussion

This section presented a detailed evaluation of iCEP. In almost all our tests, we measured a high recall and precision, even when considering relatively large evaluation windows and heterogeneous constraints including parameters, sequences, and aggregates. Although iCEP may receive external hints to improve its accuracy, we never exploited this feature: the learning process was completely automated.

During our tests, we observed the impact of the size and quality of the training dataset. Precision and recall decreased with a (very) limited number of traces (see Figure 12). However, for our default scenario, 100 traces were already enough to obtain a recall and precision above 0.95. Similarly, the accuracy of results decreased in presence of a small number of primitive event types (see Figure 6) or with large windows (see Figure 5). Indeed, in both cases, the evaluation window includes multiple events of each type, which may bias the learning process.

From the observation above, we can derive a more general conclusion concerning the characteristic of input data: since iCEP starts filtering positive traces based on the event types they include, it provides the best accuracy in all the scenarios where each type encodes the occurrence of an exceptional fact. Conversely, as discussed in our case study of traffic monitoring, when events in traces encode periodic updates about the state of the environment, we observe some differences between rule R and R^*. Nevertheless, also in this case, both precision and recall remain close to one.

Another important aspect that emerged from our analysis is the crucial role of the `Constr Learner`. On the one hand, almost all modules in iCEP rely on the knowledge of selection constraints. On the other hand, detecting them is very challenging since it requires to generate selection criteria for events based only on available observed values. As observed in different experiments, this complexity sometimes translates in results that are not fully accurate. We believe that domain specific techniques and optimizations could contribute to enhance the accuracy of the `Constr Learner`. We plan to apply iCEP to specific scenarios (e.g., system security) to validate this hypotesis.

Finally, there are cases in which the general idea of intersecting constraints may generate less accurate results. We experienced this limitation in the case of multiple negations, where the firing of a rule can be prevented by the presence of different negated events. Despite our previous investigations with alternative solutions (e.g., machine learning techniques [36]) this remains an open issue that we plan to investigate further in future work.

6. RELATED WORK

This section revises related work. First, it presents existing CEP systems, with the purpose of showing the applicability of iCEP. Second, it introduces research efforts in the CEP area that are related to rule learning. Finally, it presents existing approaches that deal with the general problem of extracting knowledge from data traces.

Stream and Event Processing Systems. The last few years have seen an increasing interest in technology for stream and event processing, and several systems have been proposed both from the academia and from the industry [33, 24]. Despite all existing solutions have been designed to accomplish

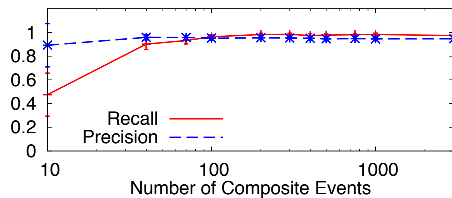

Figure 12: Number of Composite Events

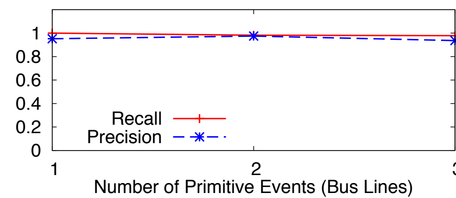

Figure 13: Number of Bus Lines

the same goal, i.e., to process large volumes of flowing data on-the-fly, they present different data models and rule definition languages, as well as processing algorithms and system architectures. For example, several kinds of languages have been proposed that range from extensions of regular expressions [32, 12], to logic languages [5, 16, 17, 8], and declarative languages [2, 46]. According to the analysis presented in [19] we can roughly identify two main classes of systems. On the one hand, the database community gave birth to *Data Stream Management Systems (DSMSs)* [10] to process generic information streams. On the other hand, the community working on event-based systems focused on a form of data, event notifications, with a specific semantics, in which the time (of occurrence) plays a central role [37].

DSMSs usually rely on languages derived from SQL, which specify how incoming data have to be transformed, i.e., selected, joined together, and modified, to produce one or more output streams. Processing happens in three steps [6]: first, *Stream-to-Relation* (or *windows*) operators are used to select a portion of a stream and to implicitly create traditional database tables. The actual computation occurs on these tables, using *Relation-to-Relation* (mostly SQL) operators. Finally, *Relation-to-Stream* operators generate new streams from tables, after data manipulation. Despite several extensions have been proposed [23, 49, 39], they all rely on the general processing schema described above.

At the opposite side of the spectrum, CEP systems are explicitly designed to capture composite events (or situations of interests) from patterns of primitive ones [24]. CEP systems often trade simplicity and performance for expressiveness, providing a reduced set of operators: for example, some languages force sequences to capture only adjacent events [12]; negations are rarely supported [32, 12] or they cannot be expressed through timing constraints [3]. iCEP mainly targets this second class of systems, with the aim of learning the causal relations between the presence of primitive events and the occurrence of a composite one. Nevertheless, most of the results we presented can be applied to both classes of systems. Our long term goal in the development of iCEP is to exploit all the operators offered in the most expressive rule definition languages, thus enabling the derived rules to capture causal relationships present in the observed environment as precisely as possible.

In this perspective, it is important to mention that some CEP systems adopt an interval-based, as opposed to point-based, time semantics: time is modeled using two timestamps that indicate the interval in which an event is considered valid [1]. In addition, some CEP systems define iteration operators (Kleene+ [28]) to capture a priori unbounded sequences of events. This is typically exploited to detect trends (e.g., constantly increasing value of temperature). Trend detection as well as the extension of iCEP to interval-based semantics are part of our future work.

Rule Learning and Related Efforts in CEP. Concerning the specific problem of learning CEP rules we can mention some relevant existing research efforts. For example, [38] describes a rule learning approach based on an extension of Hidden Markov models [42] called Noise Hidden Markov models that can be trained with existing, low-level, event data. This work precisely addresses the same research goal of iCEP, however, it differs in several fundamental aspects. For example, differently from iCEP, it only processes and learns elementary events (i.e., attributes are not supported) and does not provide the same level of expressiveness as iCEP. Indeed, it does not support some operators commonly offered in CEP languages (e.g., aggregates and parameters).

Worth to mention is also [50], which discusses an iterative approach for automating both the initial definition of rules and their update over time, combining partial information provided by the domain experts with machine learning techniques. This work differs from iCEP in two distinct aspects: it explicitly requires the human intervention in its tuning phase and it does not provide the same level of expressiveness as iCEP in terms of learned operators (e.g., negations aggregates). Similarly, [47] discusses an iterative approach to support the domain expert in designing new rules and identifying missing ones.

Proactive event processing [22] is an additional research goal strictly related to the idea of automated rule generation. It aims at predicting the occurrence of future events, thus enabling actions that can mitigate or eliminate undesired situations. Connected to proactive processing is the idea of computing the degree of uncertainty (often modeled as probability) associated to composite events. The first model proposed for dealing with uncertainty in CEP is described in [52], and extended in [53, 54], where the authors introduce a general framework for CEP in presence of uncertainty. Similar approaches have been studied in [44, 21, 43]. Worth to mention is [9], where the authors tackle the issue of uncertainty in transportation systems and explore a logic-based event reasoning tool to identify regions of uncertainty within a stream. Finally, a tutorial on event processing under uncertainty has been presented in the DEBS (Distributed Event Based Systems) 2012 conference [7]. We plan to explore these models to understand a potential integration with iCEP, to offer some indication about the confidence one can put on the learned rules and their derivations.

Related Learning Approaches. The general problem of knowledge learning and mining from (time-annotated) data traces has been extensively studied in the past (e.g., [27]). The interested reader can refer to [45, 31] for an extensive description and classification of the solution proposed. The techniques adopted in iCEP are close to the algorithms described in [35, 40]. Worth to mention among them are the approaches that adopt multi-dimensional variables to sup-

port events with multiple attributes. Such techniques are considered one of the most important future trends in data mining [30], and several algorithms have already been proposed [26]. In contrast to the approaches mentioned above, iCEP adopts a totally different viewpoint explicitly conceived for CEP. Indeed, it solves the learning problem by modelling rules and traces as set of constraints and systematically computing their intersections. Moreover, existing solutions lack the expressiveness of iCEP, which is designed to automatically discover multiple rule features.

As mentioned in Section 5, the algorithms currently implemented in iCEP are not well suited to deal with application fields in which primitive events are produced by periodic readings of a certain value, e.g., the price of a stock, or the temperature read by a sensor. Indeed, these settings often require mechanisms to learn temporal trends. We plan to integrate iCEP with trend detection mechanisms, such as those described in [11, 55].

Several approaches have been proposed to construct temporal rules in noisy environments. For example, [29] adopts Markov logic networks to construct knowledge bases, while [4] proposes a data mining algorithm for inducing temporal constraint networks. Interestingly, [56] addresses the problem of unusual event detection and discusses an approach based on a semi-supervised Hidden Markov Models combined with Bayesian theory. Finally, the problem of learning rules from data traces has been also addressed in various domains, such as medical applications [41, 14], detection malicious intrusions in software systems [48, 34], and mining frequent patterns in alarm logs [25].

7. CONCLUSIONS

In this paper, we addressed the problem of automated rule generation for CEP systems. We precisely defined the problem and proposed a general approach that builds on three fundamental pillars: decomposition into sub-problems, modularity of the solution, and ad-hoc learning algorithms. Moreover, we presented iCEP, a concrete implementation of our approach, which effectively trades off performance and expressiveness. On the one hand, iCEP supports all the operators commonly offered by the state of the art CEP systems. On the other hand, it can analyze thousands of traces in a few minutes. In addition, the highly modular architecture of iCEP enables extensibility and customization. An extensive evaluation of iCEP, based on synthetical as well as real data, demonstrated its benefits in terms of recall and precision. As future work we plan to deploy iCEP in real a real world application to further challenge its concrete applicability and flexibility.

To conclude, while CEP systems have proved their effectiveness and efficiency in a wide range of scenarios, the complexity of rule definition still represents a challenge that received little attention. We hope that, by building on top of our contribution, and by exploiting and refining our framework, future research could simplify the use of CEP systems and promote their adoption.

Acknowledgment

This research has been funded the Dutch national program COMMIT and by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom and by Programme FP7-PEOPLE-2011-IEF, Project 302648-RunMore.

8. REFERENCES

[1] R. Adaikkalavan and S. Chakravarthy. SnoopIB: Interval-based event specification and detection for active databases. *Data & Know. Eng.*, 59(1):139 – 165, 2006.

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

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

[4] M. R. Álvarez, P. Félix, P. Cariñena, and A. Otero. A data mining algorithm for inducing temporal constraint networks. In *Computational Intelligence for Knowledge-Based Systems Design*, pages 300–309. Springer, 2010.

[5] D. Anicic, P. Fodor, S. Rudolph, R. Stühmer, N. Stojanovic, and R. Studer. Etalis: Rule-based reasoning in event processing. *Reasoning in Event-Based Distributed Systems*, pages 99–124, 2011.

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

[7] A. Artikis, O. Etzion, Z. Feldman, and F. Fournier. Event processing under uncertainty. In *DEBS*, 2012.

[8] A. Artikis, G. Paliouras, F. Portet, and A. Skarlatidis. Logic-based representation, reasoning and machine learning for event recognition. In *ACM DEBS*, pages 282–293. ACM, 2010.

[9] A. Artikis, M. Weidlich, A. Gal, V. Kalogeraki, and D. Gunopulos. Self-adaptive event recognition for intelligent transport management. 2013.

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

[11] D. J. Berndt and J. Clifford. Advances in knowledge discovery and data mining. chapter Finding patterns in time series: a dynamic programming approach, pages 229–248. American Association for Artificial Intelligence, 1996.

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

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

[14] G. Carrault, M.-O. Cordier, R. Quiniou, and F. Wang. Temporal abstraction and inductive logic programming for arrhythmia recognition from electrocardiograms. *AI in Medicine*, 28(3):231–263, 2003.

[15] C. Cortes and V. Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 2013.

[16] G. Cugola and A. Margara. Tesla: a formally defined event specification language. In *DEBS*, pages 50–61. ACM, 2010.

[17] G. Cugola and A. Margara. Complex event processing with t-rex. *Journal of Systems and Software*, 85(8):1709 – 1728, 2012.

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

[19] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, 2012.

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

[21] Y. Diao, B. Li, A. Liu, L. Peng, C. Sutton, T. T. L. Tran, and M. Zink. Capturing data uncertainty in high-volume stream processing. In *CIDR*, 2009.

[22] Y. Engel and O. Etzion. Towards proactive event-driven computing. In *DEBS*, pages 125–136. ACM, 2011.

[23] Esper, http://esper.codehaus.org/, 2013.

[24] O. Etzion and P. Niblett. *Event Processing in Action*. Manning Publications Co., 2010.

[25] F. Fessant, F. Clérot, and C. Dousson. Mining of an alarm log to improve the discovery of frequent patterns. In *Advances in Data Mining*, pages 144–152. Springer, 2005.

[26] T. Gärtner, P. A. Flach, A. Kowalczyk, and A. J. Smola. Multi-instance kernels. In *ICML*, pages 179–186, 2002.

[27] V. Guralnik and J. Srivastava. Event detection from time series data. In *ACM SIGKDD*, pages 33–42. ACM, 1999.

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

[29] S. Kok and P. Domingos. Learning markov logic networks using structural motifs. *Fürnkranz, J., Joachims, T.(eds.)*, 951:551–558, 2010.

[30] H.-P. Kriegel, K. M. Borgwardt, P. Kröger, A. Pryakhin, M. Schubert, and A. Zimek. Future trends in data mining. *Data Mining and Knowledge Discovery*, 15(1):87–97, 2007.

[31] S. Laxman and P. Sastry. A survey of temporal data mining. *Sadhana*, 31:173–198, 2006.

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

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

[34] M. V. Mahoney and P. K. Chan. Learning rules for anomaly detection of hostile network traffic. In *IEEE ICDM*, pages 601–604. IEEE, 2003.

[35] H. Mannila, H. Toivonen, and A. Inkeri Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Know. Disc.*, 1:259–289, 1997.

[36] A. Margara, G. Cugola, and G. Tamburrelli. Towards Automated Rule Learning for Complex Event Processing, 2013.

[37] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer-Verlag, 2006.

[38] C. Mutschler and M. Philippsen. Learning event detection rules with noise hidden markov models. In *NASA/ESA Conf. on AHS*, pages 159–166. IEEE, 2012.

[39] Oracle cep. http://www.oracle.com/technologies/soa/complex-event-processing.html, 2013.

[40] B. Padmanabhan and A. Tuzhilin. Pattern discovery in temporal databases: A temporal logic approach. In *KDD*, pages 351–354, 1996.

[41] R. Quiniou, L. Callens, G. Carrault, M.-O. Cordier, E. Fromont, P. Mabo, and F. Portet. Intelligent adaptive monitoring for cardiac surveillance. In *Computational Intelligence in Healthcare 4*, pages 329–346. Springer, 2010.

[42] L. Rabiner and B. Juang. An introduction to hidden markov models. *ASSP Magazine, IEEE*, 3(1):4–16, 1986.

[43] M. Rajesh Khanna and M. Dhivya. A generic framework for deriving and processing uncertain events in rule-based systems. In *ICICES*, pages 398–403. IEEE, 2013.

[44] C. Ré, J. Letchner, M. Balazinksa, and D. Suciu. Event queries on correlated probabilistic streams. In *SIGMOD*, pages 715–728. ACM, 2008.

[45] J. Roddick and M. Spiliopoulou. A survey of temporal knowledge discovery paradigms and methods. *IEEE TKDE*, 14(4):750–767, 2002.

[46] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *DEBS*, pages 4:1–4:12. ACM, 2009.

[47] S. Sen, N. Stojanovic, and L. Stojanovic. An approach for iterative event pattern recommendation. In *ACM DEBS*, pages 196–205. ACM, 2010.

[48] C. Sinclair, L. Pierce, and S. Matzner. An application of machine learning to network intrusion detection. In *IEEE ACSAC*, pages 371–377. IEEE, 1999.

[49] Streambase, http://www.streambase.com/, 2013.

[50] Y. Turchin, A. Gal, and S. Wasserkrug. Tuning complex event processing rules using the prediction-correction paradigm. In *ACM DEBS*, page 10. ACM, 2009.

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

[52] S. Wasserkrug, A. Gal, and O. Etzion. A model for reasoning with uncertain rules in event composition systems. In *UAI*, pages 599–606, 2005.

[53] S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin. Complex event processing over uncertain data. In *DEBS*, pages 253–264. ACM, 2008.

[54] S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin. Efficient processing of uncertain events in rule-based systems. *IEEE Trans. on Knowl. and Data Eng.*, 24(1):45–58, 2012.

[55] A. Weigend, F. Chen, S. Figlewski, and S. Waterhouse. Discovering technical traders in the t-bond futures market. In *KDD*, pages 354–358. Citeseer, 1998.

[56] D. Zhang, D. Gatica-Perez, S. Bengio, and I. McCowan. Semi-supervised adapted hmms for unusual event detection. In *IEEE CVPR*, volume 1, pages 611–618. IEEE, 2005.