Efficient Analysis of Event Processing Applications

Gianpaolo Cugola
Politecnico di Milano
gianpaolo.cugola@polimi.it

Alessandro Margara
Università della Svizzera italiana (USI)
alessandro.margara@usi.ch

Mauro Pezzè
Università della Svizzera italiana (USI)
mauro.pezze@usi.ch

Matteo Pradella
Politecnico di Milano
matteo.pradella@polimi.it

ABSTRACT

Complex event processing (CEP) middleware systems are increasingly adopted to implement distributed applications: they not only dispatch events across components, but also embed part of the application logic into declarative rules that detect situations of interest from the occurrence of specific pattern of events. While this approach simplifies the development of large scale event processing applications, writing the rules that correctly capture the application domain arguably remains a difficult and error prone task, which fundamentally lacks consolidated tool support.

Moving from these premises, this paper introduces CAVE, an efficient approach and tool to support developers in analyzing the behavior of an event processing application. CAVE verifies properties based on the adopted CEP ruleset and on the environmental conditions, and outputs sequences of events that prove the satisfiability or unsatisfiability of each property. The key idea that contributes to the efficiency of CAVE is the translation of the property checking task into a set of constraint solving problems. The paper presents the CAVE approach in detail, describes its prototype implementation and evaluates its performance in a wide range of scenarios.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.1.3 [**Software**]: Concurrent Programming—*Distributed Programming*

Keywords

Complex Event Processing, Middleware, Program Analysis, Constraint Solving

1. INTRODUCTION

Modern software systems are more and more often built from loosely coupled, distributed components that observe stimuli generated from the other components in the form of *events*, process them, and react by producing new events. We refer to this class of systems as *event processing applications.* Examples of event processing applications come from different fields, such as sensor networks for environmental monitoring [10], payment analysis for fraud detection [31], financial applications for trend discovery [17], RFID-based inventory management for anomaly detection [34].

Event-based middleware systems [29] have been widely adopted as a communication substrate to inter-connect the components of event processing applications. Traditionally, they were used to dispatch events from producers to consumers, based on the interests that the latter registered in the system. More recently, event-based middleware systems have evolved from mere communication layers to intelligent services that administer part of the logic of an event processing application. So called complex event processing (CEP) systems [15, 25, 20] detect situations of interest —or *composite events*— from the observation of specific patterns of event notifications. This behavior is programmed in a declarative way through *rules* that embed part of the application logic by expressing how composite events are derived.

Using CEP systems, programmers offload part of the event management to the middleware, which becomes responsible for filtering, combining and aggregating low level events to offer a higher level vision of the application.

This new breed of event-based middleware has received significant attention from both researchers and practitioners: for instance, CEP systems have been adopted to manage the workflow of large companies [26] and to support financial trading[1]. Particular effort has been devoted to define suitable languages to express event processing rules [12, 22, 2] and to design and implement efficient event processing tools [13, 14, 9, 31].

Despite these achievements, designing, analyzing and evaluating event processing applications still remains a difficult and error prone task. Developers have to check the correctness of the rules they write with respect to the desired system behavior, taking into account how each rule interacts with the external environment and with the other rules. Some interesting research efforts address this problem [30, 19, 35]. Nevertheless, there exist no established analysis and support tools to help the developers in the complex task of analyzing event processing applications.

We believe that devising techniques and tools to support the definition, validation, analysis and evolution of event processing applications based on CEP middleware has be-

DEBS'15, June 29 - July 3, 2015, OSLO, Norway.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3286-6/15/06 ...$15.00.
http://dx.doi.org/10.1145/2675743.2771834.

[1]http://fixglobal.com/home/secrets-revealed-trading-tools-uncover-hidden-opportunities/

come fundamental to promote the diffusion and usability of CEP systems, reduce the time and cost for developing event processing applications, and potentially improve their quality.

Starting from these premises, this paper introduces CAVE (Constraint-based Analysis of eVEnts), a novel methodology and tool to support developers in analyzing the behavior of an event processing application. CAVE aims to identify potential errors in event processing applications by checking the satisfiability of properties that indicate whether a situation of interest can possibly occur, based on the set of rules written by the developers and on the assumptions they have about the environment in which the application operates. Moreover, CAVE automatically generates sequences of events that prove a given property true or false. In this way, the developers can easily understand the implications of the rules they write, and, in case of property violation, they gain some insight about the specific conditions that lead to such violation.

CAVE operates by decomposing rules into the set of content and temporal constraints that event notifications must satisfy in order to trigger each rule, and relies on efficient constraint solving tools to prove the satisfiability of properties. The efficiency of CAVE stems from generating constraint solving problems tailored towards the verification of single properties instead of relying on comprehensive models of the entire ruleset. This contributes in making CAVE scalable and suitable to dynamic contexts, in which the developers need to frequently change or adapt the ruleset and thus require a timely verification of the effects of their choices.

Contributions. This paper contributes to the research on event processing applications and CEP middleware in several ways: (*i*) it proposes a novel and efficient approach to the analysis of rules for event processing applications, which helps the developers in verifying relevant properties about the application under specific conditions; (*ii*) it presents a concrete implementation of the approach into a prototype tool; (*iii*) it provides a detailed evaluation of the performance of the tool under a wide range of scenarios.

Outline. The rest of the paper is organized as follows: Section 2 presents background knowledge on CEP middleware systems and discusses the motivations for our work; Section 3 presents the CAVE approach for the analysis of event processing applications in detail. Section 4 presents the implementation of CAVE into a prototype tool and evaluates its performance in a wide range of scenarios; finally, Section 5 surveys related work and Section 6 concludes the paper and suggests future work in the area.

2. BACKGROUND AND MOTIVATIONS

According to the definition introduced in [15], a CEP system analyzes a stream of input *primitive* events to detect occurrences of situations of interest, or *composite events*, based on declarative *rules* that define composite events in terms of *patterns* of primitive (or composite) ones.

Several CEP systems and rule definition languages have been proposed by both the academia and the industry. This section identifies an abstract event model and a rule language that cover the functionalities of most of these systems. We use them in the remainder of the paper, while we defer to Section 5 a discussion of their generality.

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 and names 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, in a hardware monitoring system, the following notification:

```
Temp@10(component=GPU, value=28.5)
```

captures the fact that the temperature measured at time `10` on component `GPU` is `28.5`°C.

2.2 Rule Model

A CEP rule defines a composite events from a *pattern* of events. When such a pattern is detected within the stream of input events, the CEP system knows that the corresponding composite event has occurred and notifies the interested components. We say that the stream of input events *satisfies* the pattern. We also take the earliest time at which the pattern is satisfied as the timestamp of the pattern itself and consequently the timestamp of the corresponding composite event.

For the sake of generality, in this paper we assume an ad-hoc CEP language that includes the most common operators found in existing languages: selection, sequence, conjunction, disjunction, parameterization, window, aggregation and negation [15]. In this language, rules assume the general form:

```
CE(a1=f1(..), ..., an=fn(...)) := pattern
```

where the symbol `:=` separates the *head* of the rule from the *pattern*. The former specifies the type `CE` of the composite event captured by the rule and how its attributes `a1,...,an` are functionally defined from the attributes of the events that appear into the pattern.

Selection. The simplest pattern we consider *selects* a single event from the input stream through its type and content, the latter being specified using predicates on its attributes. For instance, rule:

```
Overheat() := Host(cpu_load>80 and gpu_load>80 or
                   cpu_load=100)
```

is satisfied if and only if an event of type `Host` is detected whose attributes `cpu_load` and `gpu_load` are both greater than `80`, or the `cpu_load` alone equals `100`. The time at which a selected event `Host` occurs becomes the occurrence time of the pattern and consequently the timestamp of `Overheat`. We admit predicates that combine comparison relations with logical conjunctions (`and`) and disjunctions (`or`).

Sequence. The *sequence* operator introduces an ordering relationship between two events (more specifically, two patterns). In particular, pattern `P1 then P2` is satisfied if and only if pattern `P1` is followed by pattern `P2`, i.e., the timestamp of `P1` precedes the timestamp of `P2`. As an example, rule:

```
Overheat() := CPU(load>80) then GPU(load>80)
```

states that composite event Overheat happens if and only if a CPU event with attribute load greater than 80 is followed (immediately or after other events) by a GPU event with attribute load greater than 80. The pattern is detected when event GPU occurs, so the timestamp associated to the pattern coincides with the timestamp of event GPU.

Logical connectives. Two patterns can be combined using *logical* operators and and or. The former demands both patterns to be satisfied, while the latter demands at least one of them to be satisfied. As in the general case, the timestamp associated with the resulting pattern is the timestamp associated with the latest pattern being satisfied. For instance, rule:

```
Overheat() := CPU(load>80) and GPU(load>80)
```

states that we have a Overheat composite event if and only if we detect both a CPU event with load>80 and a GPU event with load>80. The CPU event may happen before the GPU event or vice-versa. In any case, the timestamp of the pattern is the timestamp of the last of the two events.

Windows. The scope of patterns can be limited in time by using the *window* operator, which forces a specific timespan for the events that satisfy the pattern. As an example, the rule:

```
Overheat() := [CPU(load>80) and
               GPU(load>80)] 5m..10m
```

constrains the interval that may separate the CPU and GPU events: between five and ten minutes.

Parameters. In the presence of patterns that include multiple events, *parameters* impose relationships among the values of attributes within different events. For example the following rule:

```
Overheat() := CPU(load>80 and host_id=$h) and
              GPU(load>80 and host_id=$h)
```

introduces parameter $h to relate attribute host_id of events CPU and GPU.

Aggregates. *Aggregates* are only used as part of a conjunction and constrain the result of an aggregation function computed over the attributes of events that meet certain conditions. For instance, the following rule:

```
Overheat() := (CPU(load>80) then GPU(load>80)) and
              avg(CPU().load)>50
```

states that composite event Overheat is triggered when event CPU with load>80 is followed by event GPU with load>80 and the average value of the load attribute in *all* the CPU events that occur between the two is greater than 50.
The general syntax for an aggregate is:

```
P and f(A(c)) op v
```

where f is an aggregation function, c is a selection constraint, op a relational operator and v a value. We call f_v the value of the aggregation function f computed over all the events of type A captured by the selection constraint c and occurring in the time interval that includes all the events used to satisfy P. The pattern is satisfied if and only if P is satisfied and the value f_v satisfies the relation f_v op v.

Negation. The last operator we consider is *negation.* It only applies to selection patterns and can only be used as part of a conjunction, as in the following rule:

```
Overheat() := (CPU(load>80) then GPU(load>80)) and
              not CPU(load<=80)
```

which states that composite event Overheat happens if and only if event CPU with load>80 is followed by event GPU with load>80 and no event CPU with load<=80 happens between the two. In general, pattern P and not E is satisfied if and only if pattern P is satisfied and event E never occurs in the time interval that includes all the events used to satisfy P.

Hierarchies. The last aspect we consider in our language is the ability of organizing rules into hierarchies by allowing composite events to be defined from other composite events. In other words, patterns may include not only primitive events, but also composite events defined by other rules.

2.3 Problem Statement

CEP middleware systems aim to simplify the development of event processing applications by enabling developers to express part of the application logic using rules. On the one hand, rules become a critical part of the application; on the other hand, writing rules is a complex and error prone task that involves checking the correctness of rules with respect to the desired application behavior while taking into account possible assumptions on the environment in which the application operates and dependencies and mutual interactions of different rules.

To the best of our knowledge, there exist no established methodologies and tools to support the development of event processing applications through the analysis and verification of CEP rules. As we better argument in Section 5, the few existing proposals in this area either lack the support for checking application specific properties or rely on expensive techniques that limit their applicability and scalability.

In this paper we aim to overcome such limitations and provide a methodology and tool for checking a ruleset against application specific properties that is capable of scaling to large sets of mutually interacting rules. The approach considers assumptions about the application environment and enables root cause analysis in the case of property violation, by constructing a concrete sequence of events that leads to the violation.

3. THE CAVE APPROACH

This section motivates the need for a new approach and then presents an overview and a detailed description of the CAVE approach.

3.1 The need for a new solution

Possible approaches to the analysis of CEP rules include (*i*) model checking techniques, which build a finite state model of the event processing application and use it to check properties, and (*ii*) direct translation of rules and properties into logic formulas, whose satisfiability can be verified using SAT/SMT solvers.

Model checking techniques such as REX [19] translate rules into temporal automata and then use model checkers,

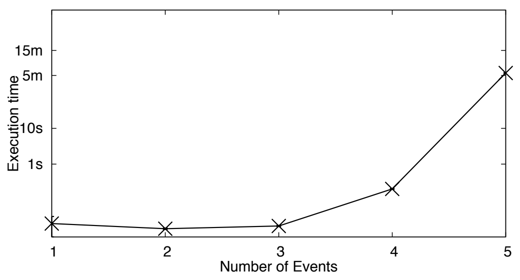

Figure 1: Cost of satisfiability with a FOL model with Z3.

like the UPPAAL model checker in the case of REX, to verify the properties of interest. While these proposals show that model checking techniques can be used to analyze CEP rules, the literature lacks detailed data about the performance of these approaches and thus about the cost and the feasibility of the approaches. Our preliminary experiments indicate the limited applicability of the existing model checking tools to the problem we want to solve. Indeed, in several cases the state space explosion leads to prohibitive computational costs and memory occupancy.

Approaches that translate rules and properties into logical formulas to be processed through SAT/SMT solvers present similar feasibility problems. In a preliminary study, we started by adopting a naïve solution that exploits the formalization of CEP rules into First Order Logic (FOL) formulas, following the semantics proposed in [12]. We used predicates to denote the occurrence time of events and the values of event attributes, and universal quantifiers and implications to model the occurrence of a composite event when its defining pattern is detected.

Figure 1 shows the time required by the Z3 SMT solver [16] to check the satisfiability of a simple property, asking for the possible occurrence of a composite event e in the presence of a single CEP rule that defines e.

This rule requires the occurrence of n primitive events, each of them having a specific type and satisfying five constraints (numerical equalities and disequalities) on the content of its attributes. The rule does not impose any constraint on the time of occurrence of such events. Figure 1 shows that the time required to find a solution grows exponentially with the number n of primitive events used by the rule. Checking the property with only five events required more than five minutes of execution. In the case of properties with six events, the SMT solver could not generate an output after several hours of execution.

Starting from this result, we recognized the need of providing a simpler and ad-hoc translation of the rules and properties to check, with the aim of achieving a level of performance that is amenable to practical usage.

CAVE moves from this consideration and builds a simplified representation of the property checking problem. This is obtained by analyzing the ruleset and the property to check and translating them into a finite set of *configurations*, each of them including conjunctions and disjunctions of basic constraints over a number of variables.

The problem of checking the property is translated to the problem of verifying that at least one configuration admits a solution, i.e., to a set of constraint satisfiability problems. Although checking the satisfiability of a (potentially large) set of constraints remains a computationally hard problem in general, our evaluation proves that it is tractable in practice using specialized constraints solvers.

To better understand how the translation implemented in CAVE helps simplify the problem, let us consider a property `P`, which asks whether rule `A():=[X then Y(y>0)] 0..2m` can be triggered at least once.
Formally, such rule states that:

$$\forall X, Y \left(\begin{array}{c} Y.y > 0 \land X.ts < Y.ts \\ \land \\ Y.ts - X.ts < 2 \end{array} \Rightarrow \exists A(A.ts = Y.ts) \right)$$

where ts represents the timestamp of the event. As we observed above, the presence of quantifiers increases the complexity of the model and makes the verification of properties too expensive to be computed by SMT solvers.

CAVE uses a different approach that starts from property `P` to simplify the logic representation of the rule and to remove universal quantifiers. In our example, CAVE recognizes that a sequence of two event occurrences of type `X` and `Y` is *sufficient* to satisfy property `P`. Moreover, the presence of these two events is also *necessary* to satisfy `P`. Accordingly, CAVE builds a single configuration that entails these two events as a set of constraints on numeric variables that represent their timestamps and attributes values. In this way, checking the satisfiability of `P` only involves finding an assignment of values to those variables that satisfy the constraints expressed by the rule.

3.2 CAVE in a Nutshell

Figure 2 shows an overview of the CAVE approach. CAVE takes in input a *property* to be checked, a set of CEP *rules* and a number of *assumptions* on the environment under analysis. The property represents a desired or undesired situation of interest, defined as a pattern of events adopting the same language used for expressing rules. CAVE computes whether the property can be satisfied or not with the given ruleset and environmental conditions, i.e., whether it is possible to find at least one sequence of events that satisfies the requirements expressed through the property.

CAVE operates in two steps. In the first step, it *translates* the property into a finite number of *configurations*. Each configuration represents a constraint solving problem, with a set of variables and a set of constraints over such variables. In the second step, CAVE uses a constraint solver to find a solution for at least one configuration. A solution for one configuration identifies a sequence of events that satisfy the property. If none of the configurations can be satisfied, then also the original property cannot be satisfied.

To better describe the main building blocks in Figure 2, let us consider again a ruleset composed of a single rule `R`, which is the same rule we introduced in the previous section:

```
Rule R
A() := [X then Y(y>0)] 2m
```

Let us further assume that the application domain imposes that two events of type `Y` can never occur within five minutes from each other. Finally, consider a simple property `P1` that asks whether it is possible to observe (at least) one occurrence of an event of type `A`.

```
Property P1
A()
```

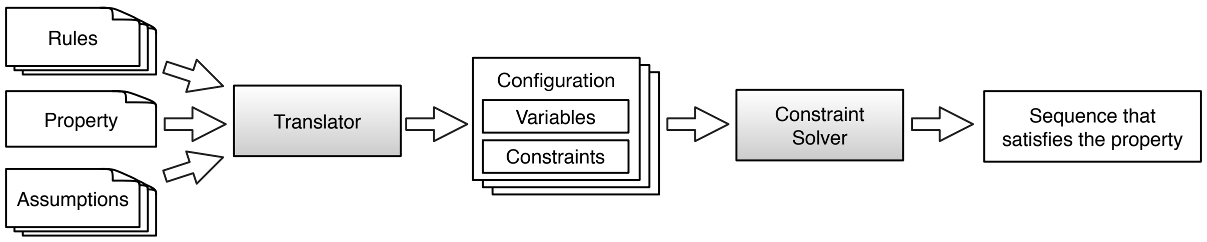

Figure 2: An overview of CAVE

This is expressed by the very simple pattern above, which requires the occurrence of an event of type A without any additional constraint.

CAVE observes that event A can be generated only from rule R. Thus, property P1 can be satisfied only if rule R can be triggered at least once. This brings CAVE to create a single configuration which includes an event of type X and an event of type Y. The configuration defines three variables: X.timestamp, Y.timestamp, and Y.y. Intuitively, rule R defines three basic constraints over such variables:

```
Y.y>0 and
X.timestamp < Y.timestamp and
Y.timestamp - X.timestamp < 2
```

In general, CAVE adds to each configuration also the constraints derived from the assumptions on the application environment, if any. In this case, the environment only limits the distance in time between any two events of type Y. Since property P1 can be satisfied with a single occurrence of an event of type Y, CAVE does not add any constraint.

This terminates the translation phase. In the subsequent phase, the constraint solver recognizes the generated configuration as satisfiable and produces a concrete instantiation of the value of each variable, such as X.timestamp = 1, Y.timestamp = 2, Y.y = 1.

Let us now consider a second property P2 asking for two events of type A to occur between three and four minutes from each other: [A() then A()] 3m..4m

Satisfying property P2 requires triggering rule R twice. Accordingly, CAVE instantiates a set of variables for each of the two occurrences, and imposes the constraints expressed in rule R on both sets of variables, as shown below:

```
Y_1.y > 0 and
X_1.timestamp < Y_1.timestamp and
Y_1.timestamp - X_1.timestamp < 2 and
Y_2.y > 0 and
X_2.timestamp < Y_2.timestamp and
Y_2.timestamp - X_2.timestamp < 2 and
3 < Y_2.timestamp - Y_1.timestamp < 4 and
Y_2.timestamp - Y_1.timestamp > 5
```

The first three constraints predicate over the events required for the first occurrence of A. Similarly, the next three constraints predicate over the events required for the second occurrence. The last two constraints encode the requirement of property P2, i.e., the minimum and the maximum distance between the two occurrences of A, and the assumption on the environment, i.e., the minimum distance between any two occurrences of Y. The constraint solver recognizes these last two constraints as conflicting and determines that the property cannot be satisfied.

3.3 CAVE in Detail

This section discusses the translation process of CAVE in detail. It starts from the CEP operators identified in Section 2 and shows how they contribute to the definition of variables and constraints in configurations.

For ease of exposition, in the first part of this section we assume that: (*i*) the ruleset includes a single rule r that generates a composite event e; (*ii*) the property always checks whether the composite event e can occur at least once, i.e., whether rule r can be triggered; (*iii*) there are no additional constraints coming from the application environment. Under these assumptions, the goal of CAVE is to prove that the pattern of r can be satisfied by some sequence of event occurrences.

In the last part of the section we will remove these assumptions, by showing how multiple rules can be used together to verify the satisfiability of a property and how it is possible to add assumptions on the environment in which the system works.

Event instances. CAVE generates a set of variables for each event instance that explicitly appears in the pattern of the rule to satisfy. Each variable represents one attribute of the event instance, including its timestamp.
Consider for example the following rule[2]:

```
A() and A() and B()
```

The rule includes three event instances, two of them having type A and one having type B. Assuming that events of type A include a single attribute named a and that events of type B include a single attribute named b, CAVE builds a single configuration with six variables: A_1.a, A_1.timestamp, A_2.a, A_2.timestamp, B.b, B.timestamp.

The rule above does not introduce any restriction on the value of the attributes and the time of occurrence of the event instances. Thus, CAVE does not include any constraint in the configuration, meaning that any value of the variables satisfies the property.

Content constraints. *Selection* and *parameter* operators constrain the values of attributes appearing in event instances. In particular, selection operators constrain the value of an individual event attribute, while parameter constraints express mutual relations between multiple event attributes. CAVE encodes them as constraints on the corresponding variables.
Consider for example the following rule:

```
A(x>10 and y<5 and z=$p) and B(k=12 or w=$p)
```

[2]In the following, we represent a rule through its pattern and we omit its head, if not strictly necessary.

As discussed in the previous section, CAVE builds a single configuration with seven variables, A.x, A.y, A.z, A.timestamp, B.w, B.k and B.timestamp and generates the following constraints:

```
A.x>10 and A.y<5 and (B.k=12 or A.z=B.w)
```

where the first two constraints derive from the selection operators used inside the event of type A, while the third and the last constraints refer to the disjunction of the selection and parameter constraints that predicate on the attributes of the event of type B.

Temporal constraints. *Sequence* operators and temporal *windows* impose constraints on the time of occurrence of event instances. CAVE translates these operators accordingly. Consider for example the following rule:

```
[A(x>10) then B(k>2 or w<3)] 5m
```

The rule introduces two constraints that limit the temporal order and the distance in the time in which the event occur. These constraints are applied in conjunction to the selection and parameter constraints defined in the pattern, as follows:

```
A.timestamp<B.timestamp and
B.timestamp-A.timestamp<5 and
A.x>0 and (B.k>2 or B.w<2)
```

where the first constraint encodes the order between the event of type A and the event of type B imposed by the sequence operator. The second constraint encodes the maximum distance in time between the two events, as imposed by the window operator.

Aggregates. *Aggregates* impose constraints to the result of a function computed over the value of the attributes of a number of events. Usually, the set of events considered in the computation of an aggregate is not explicitly specified within the rule, but rather characterized by means of some membership criteria: all and only the events that satisfy the criteria are included into the computation of aggregates. For example the following rule:

```
(C() then B()) and Avg(A(y>0).x)<10
```

requires the computation of the average value of attribute x over all events of type A that include an attribute y>0 and occur between an event of type C and an event of type B.

In general, the number of event instances included in the computation of an aggregation function is potentially unbounded. Thus, it is not possible to determine a finite number of configurations that represent all possible sets of event instances that may satisfy the aggregate constraint.

CAVE tackles this problem by asking the developers to specify a minimum and a maximum number of events that can participate in each aggregate. Then, it generates a different configuration for each number between the minimum and the maximum.
For instance, a configuration that considers two events of type A participating in the aggregate is shown below:

```
C.timestamp < B.timestamp and
C.timestamp < A_1.timestamp < B.timestamp and
A_1.y>0 and
C.timestamp < A_2.timestamp < B.timestamp and
A_2.y>0 and
Avg(A_1.x, A_2.x)<10
```

where the second and the third constraints impose that the first event of type A satisfies the requirements for being part of the aggregate, i.e., having a value for attribute y greater than zero and occurring between the event of type C and the event of type B. Similarly, the fourth and the fifth constraints impose that the second event of type A satisfies the requirements for taking part in the aggregate. The last constraint encodes the constraint on the result of the aggregation function.

Finally, notice that other events having the type required by the aggregate may explicitly appear in the rule. CAVE uses separate configurations to encode the case in which such events participate in the computation of the aggregate and the case in which they do not. Consider for example the following rule:

```
(A(y<5) then B()) and Avg(A(y>0).x)<10
```

which introduces an aggregate constraint over some events of type A, but also requires the presence of at least one event of type A having attribute y lower then five.

Let us consider the problem of building a configuration with a single event occurrence that participates in the aggregate. There are two options, that CAVE encodes into two separate configurations. The first one assumes that the event having attribute y lower than five does not participate in the aggregate, i.e., does not have a value for attribute y that is greater than zero:

```
A_1.y>0 and A_1.timestamp<B.timestamp and
Avg(A_1.x)<10 and
A_2.y<=0 and A_2.timestamp<B.timestamp and
A_2.y<5 and
```

where A_1 represents the event used for the aggregation, while A_2 represents the event that satisfies the constraint y<5.

The second configuration assumes that the event having attribute y lower than five also participates in the aggregate.

```
A.y>0 and A.y<5 and
A.timestamp<B.timestamp and
Avg(A.x)<10
```

Negations. *Negations* require that events with certain characteristics do not occur in a given interval. For example, the following rule:

```
[A(x=$p) then B()] 5m and not C(y=$p and z>0)
```

requires that no events of type C with attribute z greater than zero and attribute y equals to A.x occur between events A and B.

Configurations assume a *closed world* semantics: configuration variables represent all and only the occurring event instances. Because of this, CAVE translates a negation by simply omitting the corresponding event from a configuration. For instance, the previous rule would be translated as follows:

```
A.timestamp < B.timestamp and
B.timestamp - A.timestamp < 5
```

where only the events of type A and B are represented, while no event of type C appears.

However, there are cases in which the property explicitly requires one or more occurrences of an event having the same type as the negated event. In this case, CAVE adds a set of constraints for each event having the same type as the negated one, forcing it to violate at least one of the requirements expressed in the negation. Consider for example the following rule:

```
C() and
( [A(x=$p) then B()] 5m and
  not C(y=$p and z>0) )
```

which explicitly requires the occurrence of at least an event of type C. CAVE forces the event of type C to violate at least one of the conditions expressed in the negation constraint. In particular, it forces the event of type C to occur either before A or after B, or to violate one of the selection and parameter constraints.

```
A.timestamp<B.timestamp and
B.timestamp-A.timestamp<5 and
( C.timestamp<A.timestamp or
  C.timestamp>B.timestamp or
  A.x!=C.y or
  C.z<=0 )
```

Hierarchies of events. As discussed in Section 2, CEP systems often enable the events generated by one rule to be used within the pattern of other rules, thus generating *hierarchies* of rules and events. In presence of hierarchies of events, CAVE adopts a rewriting mechanism that substitutes each composite event e appearing in the pattern of a rule with the sets of constraints required for the generation of e.
Consider for example the following rules:

```
X := A(a>0) then B(b>0)
A(a=$p) := C(c=$p) or D(d=$p)
B(b=$p) := E(e=$p)
```

When translating the first rule, CAVE substitutes the occurrence of A and B with all their possible triggering configurations. Since there are two different ways for generating events of type A, CAVE derives two configurations for obtaining events of type X, as shown below:

```
C.c>0 and E.e>0 and C.timestamp<E.timestamp

D.d>0 and E.e>0 and D.timestamp<E.timestamp
```

The first configuration uses an event of type C to generate A, while the second configuration uses an event of type D.

CAVE substitutes the variables that refer to event attributes and adapts content constraints accordingly. In the example above, CAVE recognizes that the value of A.a is derived from C.c in the first configuration and from D.d in the second configuration. In both cases the value of the attribute B.b is derived from the value of E.e. Variables substitution is also used for temporal constraints. In our example, the order between the occurrence of A and B is translated into constraints on the order of the events that cause the occurrence of A and B, that is to say C and E in the first configuration, and D and E in the second configuration.

Verifying properties. After describing how CAVE deals with hierarchies of events, we can remove two of the simplifying assumptions introduced at the beginning of this section, and precisely describe how CAVE checks properties identified by (*i*) generic patterns of events, (*ii*) in presence of multiple rules.

CAVE always starts the analysis from the property to be checked. This is expressed with a pattern of events in the same language used to specify the CEP rules. If the pattern only contains primitive events, then CAVE simply checks the satisfiability of the pattern. Otherwise, it substitutes each composite event appearing in the pattern recursively, to obtain a set of configurations that include only primitive events.

Using this approach, CAVE allows developers to check two kinds of properties, which answer these questions: (*i*) determine whether a certain condition can be possibly satisfied; (*ii*) determine whether a certain condition always holds.

Properties of the first kind are used to check if the system can reach a given state, for instance if a specific rule can indeed be triggered. They can also be used to check if some undesired situation can manifest, for instance if two events may occur within a predefined amount of time, while they should not. Properties of the second kind are used to capture invariants, for instance whether events of a given type always include certain attribute values, or whether they are always followed by an event of another type.

CAVE verifies properties of the second type —which ask whether a condition always holds— by negating them, thus converting them into a satisfiability problem. If the negation cannot be satisfied, then the property is verified.

Assumptions on the application environment. CAVE allows the developers to introduce additional constraints that are not explicitly encoded into the ruleset. This is useful to (*i*) express assumptions regarding the environment in which the event processing application operates and (*ii*) check how the application behaves when specific conditions are met.

Instead of defining an ad-hoc language for expressing these assumptions, we decided to use the same language of patterns adopted to encode rules and properties, thus enabling the developers to exploit a single language to specify all the aspects involved in the analysis.

However, patterns are designed to describe situations of interest and not to indicate invariants of the environment, i.e., conditions that must necessarily hold. We overcome this issue by representing assumptions in a negative form, as the set of situations (patterns) that should never occur, according to the knowledge that developers have about the environment.

For example, in an environmental monitoring application, developers may assume that the value of attribute temp of events T (which represent temperature readings from sensors), is always between 0 and 35 degrees. This assumption can be encoded using the following pattern:

```
T(temp<0 or temp>35)
```

which models the situations that developers know should never happen: those where temperature readings are lower than 0 or greater than 35 degrees.

By knowing that assumptions are expressed as situations that should never occur, CAVE first translates them in terms of constraints, as it does for properties to be checked; then it negates them to derive the constraints that should actually hold. In the example above, CAVE translate the pattern that encodes the assumption on temperature reading by adding two constraints to the temp attribute of every event of type T.

```
T_n.temp>=0 and T_n.temp<=35
```

Similarly, developers can specify the assumption presented in Section 3.2, which forces a minimum distance of five minutes to any two occurrences of events of type Y, as follows:

```
[Y() and Y()] 5m
```

CAVE searches within configurations for every couple of variables `Y_i.timestamp` and `Y_j.timestamp`, representing the time of occurrence of two events of type Y, and adds the following constraint:

```
abs(Y_i.timestamp - Y_j.timestamp) > 5
```

Finally, developers may restrict the maximum variation between two temperature values coming from the same room and occurring within one minute from each other with the following pattern:

```
[T(value=$v and room=$r) and
 T(value>$v+5 and room=$r)] 1m
```

Also in this case, CAVE searches for every couple of events of type T and imposes that either their distance in time is greater than one or it violates at least one of the parameter constraints.

```
abs(T_i.timestamp - T_j.timestamp) > 1 or
T_i.room != T_j.room or
T_i.value - T_j.value < 5
```

4. EVALUATION

We experimentally evaluated CAVE with a prototype implementation written in Java that supports all the CEP operators presented in Section 2. Using such tool with an existing CEP system only requires writing an adapter that translates the rules written in the system-specific language into the CAVE operators.

The current implementation includes three main components: (*i*) a *translator* from properties, assumptions, and rules to the constraint solver language; (*ii*) a *rewriter* that addresses hierarchies of events by substituting the composite events that appear in a pattern with their definitions; (*iii*) a *solver* that interfaces with a constraint solving tool.

We evaluated CAVE with several popular constraint solvers. In this paper, we present the results obtained with JACOP 4.2 [23], which proved to be the most efficient solver for our purpose.

4.1 Experimental setting

Given the large amount of factors that may impact on the performance of CAVE, we consider several different experiments that aim to cover the analysis space as broadly as possible. During each experiment, we check the satisfiability of a property that involves a given number of composite events, joined together by selection and parameter constraints, temporal constraints, aggregate constraints and negation constraints. We measure the total time required to analyze such property against a given ruleset, under a given set of environmental assumptions. This total time includes the time to translate and rewrite the property into a set of constraints, and the time required by JACOP to solve such constraints.

In general, not all rules in the ruleset contribute to the problem addressed by CAVE. Indeed, only the composite events (and hence the rules) that are directly or indirectly referenced by the property to check contribute to the set of constraints that, after the rewriting and translation phase, have to be checked by the constraint solver. The same is true for the assumptions on the environment. Thus, we cannot use the size and structure of the ruleset as a measure of the complexity of the problem to solve, which we capture as the number of primitive events and operators that build the pattern to check after the rewriting phase. These are the numbers we refer to in the remainder of the section with the terms: "number of event involved in the property" or "number of operators involved in the property".

In our experiments, we define a default scenario, in which the property to check involves 20 primitive events, including five attributes each. From those events we build 20 selection patterns, each using a predicate with five constraints on the event attributes. These selection patterns are finally joined together in a single conjunction that represents the property to check after the rewriting phase.

Starting from this default scenario, we build a number of experiments, each one varying a single aspect of the problem to solve, in some cases introducing additional operators, in other cases changing the number of events, attributes or operators that are already part of the default scenario.

We repeat each measure ten times, using different seeds to randomly generate the ruleset and the property to check. We plot the average value over these measures and the 95th confidence interval. Furthermore, for each experiment, we separately consider the case in which the property to check is satisfiable and the case in which it is not satisfiable. All the experiments have been conducted on a Intel Core i7-4850HQ machine with 4 cores and 16 GB of DDR3 RAM, running Java 1.7.0.51.

4.2 Performance analysis

Event instances. Our first experiment studies how the number of primitive event instances involved in the analysis of the property impacts on the performance of CAVE. We consider a range between 1 and 1000 events.

After the rewriting and translation phase, each event instance introduces six variables, one to encode the timestamp of the event and five to encode the value of its attributes. Moreover, given the way our default scenario is organized, five selection constraints per event instance are defined on those variables. This means that the number of variables and constraints passed to the constraint solver is directly proportional to the number of primitive events involved into the property to check, which ultimately determines the time JACOP requires to find a solution. This consideration explains the trends in Figure 3.

As shown in Figure 3, CAVE is much more efficient than the naïve approach discussed in Section 3.1, with an overall processing time that is below 500 ms even when considering properties and rulesets involving 1000 primitive events.

Moreover, we notice that proving a property unsatisfiable is significantly faster than finding a model for a satisfiable property. Finally, Figure 3 shows that the processing time remains stable across multiple runs, with a maximum confidence interval of less than 10 ms.

Selection constraints. Figure 4 shows how the number of selection constraints defined for each event impacts on the performance of CAVE. Each selection constraint predicates

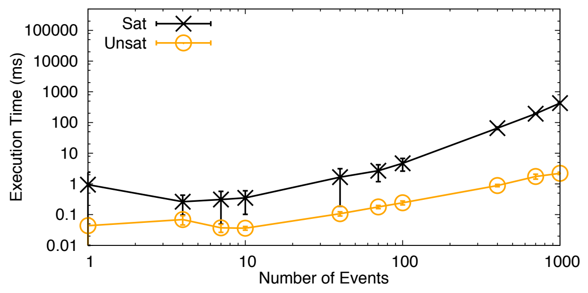

Figure 3: Number of events

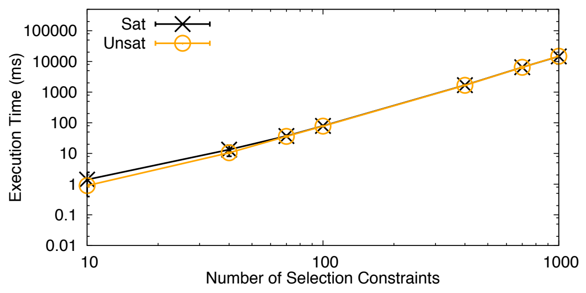

Figure 5: Number of selection constraints (conj. and disj.)

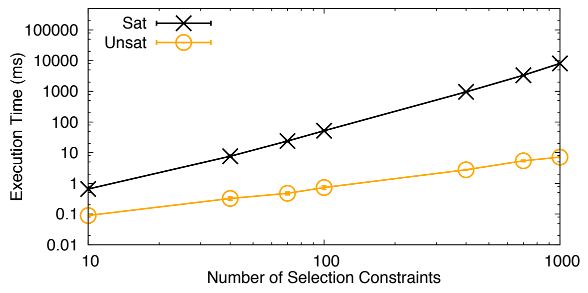

Figure 4: Number of selection constraints (conj. only)

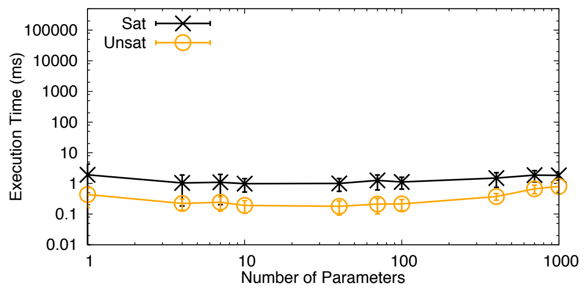

Figure 6: Number of parameters

on a different attribute. Thus, increasing the number of selection constraints leads to an increase in the number of variables and constraints that are passed to the constraint solver, and thus to the overall complexity of the problem.

Because of this, the results we measured are similar to those presented in the previous section. The overall execution time increases with the number of selection constraints, but remains within eight seconds even when considering the satisfiability of a property that involves the conjunction of 1000 selection constraints. Also in this case, proving the unsatisfiability of a property is up to two order of magnitude faster.

In the previous experiment we considered only conjunction connectives to join the constraints of each selection predicate. Next, we present the results obtained when considering also disjunctions. In particular, for each event instance we create several conjunctions of two selection constraints, and we combine all such conjunctions using disjunction connectives.

Figure 5 shows the experimental results. With respect to the previous experiment, the time for proving the satisfiability of a pattern increases by a factor of two. Interestingly, in the presence of disjunctions, proving the unsatisfiability of a property requires about the same time as proving the satisfiability. This is due to the need of proving the unsatisfiability of all the conjunctions of constraints within a disjunction before determining that a property cannot be satisfied.

Parameters. Figure 6 shows how the performance of CAVE changes with the number of parameters defined in the property to check. Differently from the previous cases, increasing the number of parameter constraints does not increase the overall number of variables passed to the constraint solver. Because of this, the processing time does not increase significantly when moving from 1 to 1000 constraints.

Parameter constraints are combined with conjunction connectives. As already observed in the previous experiments, in this setting, understanding that a property cannot be satisfied is computationally less expensive than verifying a satisfiable property. In the first case, CAVE requires at most 2 ms to produce an answer, while in the second case it requires less than 1 ms.

Temporal constraints. Figure 7 shows how the performance of the analysis changes when introducing temporal constraints. In particular, we force all the events involved in the property to occur in a specific order, and we increase the number of such events. As a consequence, the overall number of variables (and constraints) passed to the constraint solver increases as well.

We observe that the processing time increases up to a maximum of about 650 ms. In comparison to the experiment in Figure 3, where we increased the number of events without introducing temporal constraints, we observe an average 50% increase in the processing time.

Aggregates. As discussed in Section 3, the constraints that result from the use of aggregates can be potentially satisfied by sets of event instances with different cardinalities. CAVE limits the analysis to specified finite ranges. We study the cost of analyzing aggregates by considering the default scenario, introducing a single aggregate, and measuring the time required to verify or falsify a property with a specific number of event instances participating in the aggregate.

Figure 8 shows the performance we measured under this scenario. Even in this case the execution time is very low, with the maximum processing time for the analysis below 200 ms, even when considering a set of 1000 events.

Negations. Figure 9 shows how the performance of CAVE changes when dealing with negations. In this experiment, we consider 20 different primitive events. The property re-

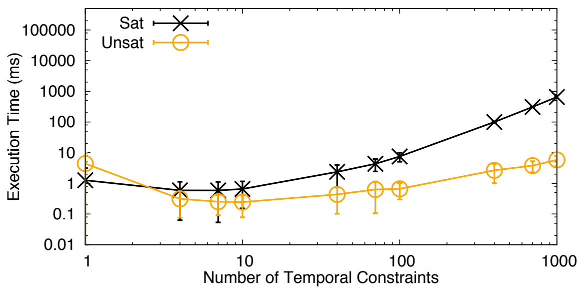

Figure 7: Number of temporal constraints

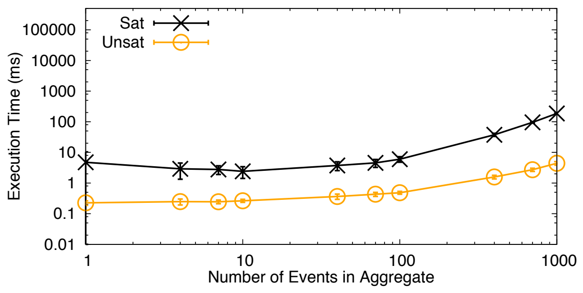

Figure 8: Number of events in aggregate

quires the occurrence of an event of each type, in a specific order. Each event instance is selected using five selection constraints. We introduce an increasing number of negations, each referring to a different event type and each using five constraints as part of its selection predicate.

As Figure 9 shows, the presence of negations does not introduce a visible overhead in the case of satisfiable properties. Conversely, the performance of CAVE decreases with the number of negations when considering non satisfiable properties. Indeed, as explained in Section 3, negations introduce disjunctions of constraints within the constraint solving problem. This increases the complexity of the analysis, since the constraint solver needs to evaluate the possible combinations of events enabled by the logic disjunctions before determining that the property cannot be satisfied.

It is also worth noticing that the additional complexity significantly increases the variance of the execution time across different runs. This is visible in the confidence intervals of Figure 9. In most cases, the time required for the analysis in the case of unsatisfiable properties was comparable to the time observed in the case of satisfiable properties. However, in a few cases the execution time of the constraint solver increased by two orders of magnitude.

We can conclude that the presence of a large number of logical disjunctions as introduced by negation constraints represents one of the factors with the highest impact on the performance of CAVE. Nevertheless, CAVE could provide an answer in less than two minutes in all the tests performed during our experiment.

5. RELATED WORK

We organize related work into three main parts: first, we present relevant research in the area of CEP systems; second, we survey existing approaches to analyze the behavior of event processing applications; third, we discuss state of the art methodologies and approaches to support the design of event-based and reactive applications.

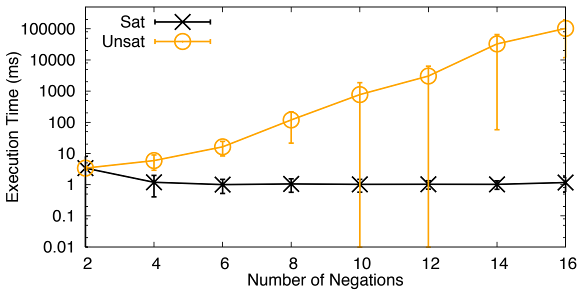

Figure 9: Number of negations

Complex Event Processing systems. We already presented CEP systems in Section 2. Here, we put them in context and discuss the generality of our approach. Complex Event Processing [25, 20] is a form of information flow processing [15] specifically devoted to the definition and detection of high level situations of interest from the observation of low level event notifications. In particular, CEP systems build on the detection of patterns that predicate over the content and the temporal relations of event notifications.

Different, complementary forms of information flow processing systems exist. Data stream management systems [3] extend the relational model to enable queries over streaming data. They provide operators to manipulate input flows of data and generate continuous streams of results. Many modern products embed CEP pattern detection capabilities and stream management capabilities in a single solution. Examples are the Esper system[3] and the Oracle CEP system[4]. Event Processing Networks (EPNs) [32, 20] have been proposed as a graph-based formalism to describe the flow of data in an event processing application.

CAVE focuses specifically on CEP and pattern detection rules. We will investigate possible extensions to capture more general forms of information flow processing as future work. In the context of CEP, several languages have been proposed for rule definition: they range from temporal extensions of regular expressions [9, 22], to logic-based programming approaches [2], to temporal logic [12]. While these languages present significant differences in terms of syntax and semantics, they all define patterns that constrain the time and content of event occurrences. Because of this, the CAVE approach based on constraint analysis can be adopted with all of them.

Analysis of event processing applications. The idea of providing analysis tools for event processing applications has been investigated in a few work. REX [18, 19] proposes a formal analysis of CEP rules based on model checking. Similar to CAVE, REX allows developers to write and verify application specific properties. In REX, rules are translated into temporal automata and properties are encoded as computational tree logic (CTL) formulas; the UPPAAL model checker [8, 7] is used to verify properties. To the best of our knowledge, there exist no evaluation of the performance of REX. At the same time, as we mentioned in Section 3.1,

[3]`http://esper.codehaus.org`
[4]`http://www.oracle.com/technetwork/middleware/complex-event-processing/`

some preliminary experiments we made, convinced us about the limited applicability of the existing model checking tools to the problem we want to solve, and suggested to follow a different path in developing CAVE.

Rabinovich et al. [30] propose a framework for the analysis of event processing applications based on the Event Processing Network formalism. The framework includes tools for static and dynamic analysis, and the paper also discusses possible implementations of analysis based on formal methods. The static analysis tool works at the level of event types, while CAVE also considers the satisfiability of content constraints. The rule language considered is less expressive than the one we are using. Moreover, the proposed dynamic analysis tools aim to analyze properties during the processing of a sequence of input events. For instance, they capture the set of rules that a specific sequence of events triggers. However, they do not automatically generate input events to stress a particular property, as CAVE does.

Weidlich et al. [35] present a tool that translates Event Processing Networks into Coloured Petri Nets for analysis and simulation. The proposed approach targets the verification of general properties, such as the presence of unused transitions in the Event Processing Network graph, and provides tools to assess the behavior of the processing network through simulation. This work is complementary to CAVE. Indeed, CAVE enables the verification of application specific properties. Moreover, CAVE does not consider simulation, but it can generate sequences of events that satisfy or violate a given property: such sequences constitute interesting inputs to validate a concrete implementation.

Finally, work related to the analysis of rule-based reactive systems can be found in the context of Active DBMSs [36]. Most of the work in this area targets general properties, such as termination [6], detection of upper limits in the number of triggered rules [4], or confluence [1, 11]. Only a few works consider application specific properties [24, 21].

Design of event-based and reactive applications. Recently, the design and implementation of (distributed) event-based and reactive applications has received significant attention. In particular, reactive programming [5] has been proposed as a programming paradigm to ease the development of such applications. It relies on the explicit definition of data dependencies and on the automated propagation of changes. The reactive programming paradigm is supported in several modern programming languages through language extensions or libraries. Similar to CEP, reactive programming delegates part of the application logic to the runtime environment or middleware, which takes care of identifying and propagating changes in variables.

The work on reactive programming targets the same problem as ours: providing suitable abstractions and methodologies to develop complex event-based and reactive applications. We believe that the results presented in this paper could be extended to support reactive programming. The interested reader can refer to [28] for an analysis of the commonalities and differences between the CEP and the reactive programming approaches.

Finally, previous work on automated generation of CEP rules [27, 33] complements the approach presented in this paper by proposing a tool that analyzes historical data to automatically or semi-automatically learn rules that model the application domain under analysis.

6. CONCLUSIONS

Complex Event Processing middleware systems are becoming increasingly popular for developing event processing, reactive applications. In such systems, part of the application logic is encoded as a set of rules that capture the relevant aspects of the application domain. Writing such rules is a critical and difficult step of the development process, since it requires the developers to take into account complex rules interactions and dependencies.

In this paper, we presented CAVE, a novel approach that assists developers and domain experts in defining a reliable set of rules for their application, by automatically checking rulesets against correctness properties, and generating concrete sequences of events that prove the validity of the properties.

CAVE transforms rules into basic constraints that can be efficiently verified by means of constraint solving techniques. The paper shows how CAVE captures the main operators used in modern CEP languages and presents a relevant set of experiments to demonstrate the efficiency and scalability of CAVE, whose execution times range from milliseconds to few minutes.

Our short term research plans include a detailed analysis of the relation between the expressiveness of the language used to define properties and assumptions, and the efficiency of the analysis. For instance, we will investigate the cost for a fully automated analysis of aggregates. We also plan to consider additional operators found in some modern CEP rule languages, such as flexible selection and consumption policies and recursive definitions [15]. Finally, we plan to integrate performance analysis and data mining techniques, to automatically learn the relevant characteristics of the environment in which the application operates.

Acknowledgments

This work was partially supported by the Swiss National Science Foundation, project DyStaCCo (grant n. 146831) and by the Italian Ministry of Education, University, and Research (MIUR), project SHELL.

7. REFERENCES

[1] A. Aiken, J. M. Hellerstein, and J. Widom. Static analysis techniques for predicting the behavior of active database rules. *ACM Trans. on Database Systems*, 20(1):3–41, 1995.

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

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 1–16, New York, NY, USA, 2002. ACM.

[4] J. Bailey, G. Dong, and K. Ramamohanarao. On the decidability of the termination problem of active database systems. *Theoretical computer science*, 311(1):389–437, 2004.

[5] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. A survey on reactive

programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, Aug. 2013.

[6] E. Baralis, S. Ceri, and S. Paraboschi. Compile-time and runtime analysis of active behaviors. *IEEE Trans. on Knowledge and Data Engineering*, 10(3):353–370, 1998.

[7] G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In M. Bernardo and F. Corradini, editors, *4th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[8] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, Oct. 1995.

[9] 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 *Proc. of the 2007 ACM SIGMOD Int. Conf. on Management of Data*, pages 1100–1102, New York, NY, USA, 2007. ACM.

[10] K. Broda, K. Clark, R. Miller, and A. Russo. Sage: A logical agent-based environment monitoring and control system. In *Proc. of the European Conf. on Ambient Intelligence*, pages 112–117, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] S. Comai and L. Tanca. Termination and confluence by rule prioritization. *IEEE Trans. on Knowledge and Data Engineering*, 15(2):257–270, 2003.

[12] G. Cugola and A. Margara. Tesla: A formally defined event specification language. In *Proc. of the Fourth ACM Int. Conf. on Distributed Event-Based Systems*, pages 50–61, New York, NY, USA, 2010. ACM.

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

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

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

[16] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[17] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Proc. of the 10th Int. Conf. on Advances in Database Technology*, pages 627–644, Berlin, Heidelberg, 2006. Springer-Verlag.

[18] A. Ericsson. *Enabling Tool Support for Formal Analysis of ECA Rules.* PhD thesis, Linköping University, 2009.

[19] A. Ericsson and M. Berndtsson. Rex, the rule and event explorer. In *Procs. of the 2007 Int. Conf. on Distributed Event-based Systems*, DEBS '07, pages 71–74, New York, NY, USA, 2007. ACM.

[20] O. Etzion and P. Niblett. *Event processing in action.* Manning Publications Co., 2010.

[21] E. Falkenroth and A. Törne. How to construct predictable rule sets. In *Procs. of the joint 24th IFAC/IFIP workshop on real time programming and 3rd international workshop on active and real-time database system*, pages 33–40, 1999.

[22] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman. On supporting kleene closure over event streams. In *Proc. of the 2008 IEEE 24th Int. Conf. on Data Engineering*, pages 1391–1393, Washington, DC, USA, 2008. IEEE Computer Society.

[23] K. Kuchcinski and R. Szymanek. Jacop-java constraint programming solver. In *Procs. of CP Solvers: Modeling, Applications, Integration, and Standardization*, 2013.

[24] S.-Y. Lee and T.-W. Ling. Verify updating trigger correctness. In *Database and Expert Systems Applications*, pages 382–391. Springer, 1999.

[25] D. C. Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.

[26] D. C. Luckham. *Event processing for business: organizing the real-time enterprise.* John Wiley & Sons, 2011.

[27] A. Margara, G. Cugola, and G. Tamburrelli. Learning from the past: Automated rule generation for complex event processing. In *Proc. of the 8th ACM Int. Conf. on Distributed Event-Based Systems*, pages 47–58, New York, NY, USA, 2014.

[28] A. Margara and G. Salvaneschi. Ways to react: Comparing reactive languages and complex event processing. In *1st Workshop on Reactivity, Events, and Modularity (REM '13)*, 2013.

[29] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed event-based systems.* Springer, Heidelberg, 2006.

[30] E. Rabinovich, O. Etzion, S. Ruah, and S. Archushin. Analyzing the behavior of event processing applications. In *Procs. of the Fourth ACM Int. Conf. on Distributed Event-Based Systems*, DEBS '10, pages 223–234, New York, NY, USA, 2010. ACM.

[31] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. In *Proc. of the Third ACM Int. Conf. on Distributed Event-Based Systems*, pages 4:1–4:12, New York, NY, USA, 2009. ACM.

[32] G. Sharon and O. Etzion. Event-processing network model and implementation. *IBM Systems Journal*, 47(2):321–334, 2008.

[33] Y. Turchin, A. Gal, and S. Wasserkrug. Tuning complex event processing rules using the prediction-correction paradigm. In *Procs. of the Third ACM Int. Conf. on Distributed Event-Based Systems*, page 10. ACM, 2009.

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

[35] M. Weidlich, J. Mendling, and A. Gal. Net-based analysis of event processing networks–the fast flower delivery case. In *Application and Theory of Petri Nets and Concurrency*, pages 270–290. Springer, 2013.

[36] J. Widom and S. Ceri. *Active database systems: Triggers and rules for advanced database processing.* Morgan Kaufmann, 1996.