Static Analysis of Complex Event Processing Programs

Adrián García-López
Universidad de Málaga, Spain
agl@lcc.uma.es

Loli Burgueño
Universidad de Málaga, Spain
UOC, Barcelona, Spain
CEA-List, Paris, France
loli@lcc.uma.es

Antonio Vallecillo
Universidad de Málaga, Spain
av@lcc.uma.es

ABSTRACT

Complex Event Processing (CEP) provides a mechanism to efficiently correlate and infer conclusions about systems by means of analyzing the events they process. In areas such as the Internet of Things (IoT), Cyber Physical Systems (CPS), system monitoring or data streaming analytics, CEP is able to read events from a data stream and to generate complex events that represent situations of interest to the system owner by means of event patterns. Every time a sequence of events matches a pattern, a complex event is created and added to the data stream. The dependencies among the rules and the possibility of non-confluent behavior of CEP rule-based systems may lead to unexpected outputs when executing CEP programs. In this work, we show how to statically check and correct two particular properties of CEP systems: rule acyclicity and rule race conditions. We use Esper EPL as a CEP language, and present a tool we have developed to perform these analyses.

KEYWORDS

Complex Event Processing; Esper EPL; Static Analysis; Rule Acyclicity; Rule Race Conditions

1 INTRODUCTION

Domains such as Internet of Things (IoT), Cyber-Physical Systems (CPS), social networks, or system monitoring, generate gigabytes of data every second. In order to take advantage of that situation, approaches for data analytics have arisen and are used to extract useful information and knowledge from it.

Complex Event Processing (CEP) [9, 15] addresses the issue of analyzing a specific kind of data: events about facts or situations occurring in real-time. CEP systems deal with the tasks of processing streams of events and the identification of significant patterns by means of techniques such as detection of relationships among events, event correlation, and aspects such as causality and timing.

CEP systems are developed using Event Pattern Languages (EPL), which are normally rule-based languages that define rules for each pattern. Patterns are triggered when their matching conditions are met (e.g., the relevant source event occurs) and are in charge of generating the resulting complex events.

As with any other rule-based language, CEP programs are easy to develop and very efficient when they are composed of a small set of rules. However, as the number of patterns (rules) grow, the complexity of CEP programs becomes unmanageable, their behavior can be unpredictable, and checking their correctness becomes a very difficult task.

In this work, we present a tool for the static analysis of CEP programs, which addresses the issues of detecting and fixing acyclicity and rule race conditions. This static analyses help improve and maintain the code quality and give developers immediate feedback in the early development phases of the CEP software system.

The rest of the paper is structured as follows. First, Section 2 presents the background of our work. Then, we present our main contribution in Section 3, and our tool in Section 4. Section 5 discusses a validation exercise we have used to evaluate our approach, using a real CEP application. Finally, Section 6 compares our approach with other related works, and Section 7 concludes and outlines our future lines of work.

2 BACKGROUND

Complex Event Processing offers a form of data processing [6] that aims at defining and detecting situations of interest, from the analysis of low-level event notifications [7].

There are two types of events in a CEP system: *simple* events, which contain the information received by the sensors; and *complex* events, which are generated by the CEP system and inserted into the data stream [10]. Each event has a type and a set of attributes associated. CEP programs are composed of sets of patterns that analyze and match sequences of events (both *simple* and *complex*) taking into account their content and temporal relations. Each time a match occurs, a complex event is created.

Although several CEP systems and languages exist, they all share the same basic concepts, mechanisms and structure. In this paper we write the rules in one particular EPL, called Esper EPL [8].

2.1 CEP by example

In order to explain CEP and to illustrate our proposal, we will use the Smart House case study we introduced in [18]. In this system, a house contains a set of devices and services that cooperate to simplify the lives of its residents.

For the particular case of ensuring security in situations of fire, we know that when a fire starts, the temperature increases at a rate higher than a given value; and carbon monoxide (CO), which weights less than air, increases and accumulates in the ceiling. For this reason, temperature and carbon monoxide (CO) sensors have been installed in the ceiling. They record the absolute value of the temperature and CO in their respective position every second and send signals to the central system.

In case of fire, another key aspect is to determine whether or not someone is at home. For this purpose, our system uses the mobile phones of its occupants as motion detectors that inform of their geographical coordinates at all times.

Using the measurements produced by these sensors, and using Esper EPL, we have defined the set of CEP rules that we show in Listing 1. Lines 1–3 show the types of the simple events, in the form of templates that specify their name and attributes. In this example,

Listing 1: Esper EPL patterns for the Smart House

```
create schema Home(id int, ts int, x int, y int, sqre int,
                   temp int, co int, dopen boolean);
create schema Person(pid int, ts int, x int, y int);

@Name("TempIncrease")
insert into TempIncrease
select h2.ts as ts, h1.id as id, h2.temp as temp,
       h2.temp-h1.temp as incr
from pattern [(every (h1 = Home() ->
                   h2=Home(h2.temp-h1.temp>=2 and h2.id=h1.id)) )
where timer:within(1 minutes)];

@Name("TempWarning")
insert into TempWarning
select t4. ts as ts t1. id as id , t4.temp
from pattern [(every (t1 = TempIncrease(t1.temp >= 33))
  -> (t2 = TempIncrease(t2.temp > t1.temp and t2.id = t1.id))
  -> (t3 = TempIncrease(t3.temp > t2.temp and t3.id = t1.id))
  -> (t4 = TempIncrease(t4.temp > t3.temp and t4.id = t1.id)))
where timer: within(5 minutes)];

@Name("COHigh")
insert into COHigh
select h1.ts as ts, h1.id as id
from pattern [(every (h1 = Home(h1.co >= 5000)))];

@Name("FireWarning")
insert into FireWarning
select tw.id as id, coh.ts as ts
from pattern [(every (coh = COHigh()) ->
               every (tw = TempWarning(tw.id = coh.id)))
where timer:within(5 seconds)];

@Name("NobodyHome")
insert into NobodyHome
select p.ts as ts, h.x as x, h.y as y, h.id as id
from pattern [(every h =Home(not dopen) -> every (p=Person(
     (p.x <= (h.x - Math.sqrt(h.sqre)/4)) or
     (p.x >= (h.x + Math.sqrt(h.sqre)/4)) or
     (p.y <= (h.y - Math.sqrt(h.sqre)/4)) or
     (p.y >= (h.y + Math.sqrt(h.sqre)/4))) ))
where timer:within(3 seconds)];

@Name("CallFireDepartment")
insert into CallFireDepartment
select fw.id as id, fw.ts as ts
from pattern [(every (nh = NobodyHome()) ->
                  fw = FireWarning(fw.id = nh.id)))
where timer:within(5 seconds)];
```

there are two types of simple events that serve as inputs to the CEP program: `Home` and `Person`. Each `Home` event keeps information of the house identifier, the event timestamp, the house location given its geographical coordinates `x` and `y`, its size measured in square meters, the temperature degrees and CO accumulated in the environment and whether the main door is open or not. Regarding the simple events of type `Person`, they are represented by a tuple with information about the person identifier, her corresponding geographical location given by its `x` and `y` coordinates, and the timestamp in which the event is produced.

Our example contains six event patterns. The pattern called `TempIncrease` matches `Home` events in a time window of one minute and checks whether the temperature of the house has increased in 2 or more degrees. If so, it generates a complex event with the same name. Similarly, the pattern `COHigh` creates a complex event each time the sensors detect CO levels above 5000 ppm.

Pattern `TempWarning` detects if four events of type `TempIncrease` occur in a five-minute window where the first one starts from a temperature equal to or greater than 33 degrees, and the rest of them detect a temperature increase of two or more degrees.

A `FireWarning` event is created every time an event of type `COHigh` is detected, followed by another of type `TempWarning`, everything within less than 5 seconds.

A `NobodyHome` event is created from the primitive events `Home` and `Person` when the main door of the house is closed and there is nobody whose geographic coordinates are within the perimeter of the house.

Finally, pattern `CallFireDepartment` creates an event of the same name when there is nobody at home and a fire warning has been detected. In these cases, the central system must react and call the fire department.

In Esper EPL, as can be inferred from previous patterns, each complex event created has a type and set of attributes indicated in the `select` clause of the pattern.

2.2 Problems with patterns

Due to the nature of rule-based declarative languages such as Esper EPL, their patterns (rules) could be executed in any order. Pattern execution of course depends on the arrival of the events and thus the event arrival would constrain the order in which patterns will execute. However, CEP systems can generate complex events that are added to the event stream, and there are rules that depend on these complex events to produce further complex events. The order in which these rules are triggered may have a significant effect on the final output of the program. This makes the execution of CEP programs non-deterministic and hard to maintain, test and debug.

The two main source of problems we detect in this work are *rule acyclicity conditions* and *rule race conditions*. Rule acyclicity takes place, for instance, when a pattern A generates events of type a, which are consumed by another pattern B, which at the same time generates events of type b that are consumed by A.

Rule race conditions occur when there are input-output dependencies between rules in uncontrollable or non-deterministic situations. In the case of CEP, rule race conditions happen when a pattern consumes complex events that have not been produced by the time it is being executed, but afterwards. Hence, the pattern misses them and its match and execution never take place although they should. For example, pattern A produces complex event a; pattern B produces complex event b; and pattern C requires the occurrence of the events a and b to produce the event c. What happens if pattern C is checked before patterns A or B are triggered?

3 STATIC ANALYSIS

In order to address the problems presented in Section 2.2, we represent CEP programs as directed graphs where the patterns are nodes and the dependencies between patterns are the graph edges. Note that two patterns can generate the same complex event. In the case that that complex event is consumed by a third pattern, this third pattern has dependencies with the two former patterns.

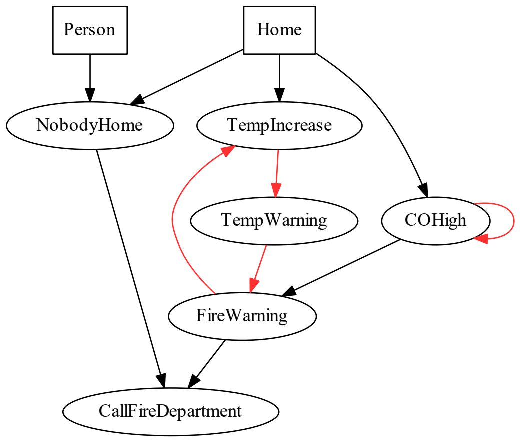


Figure 1: Rule dependencies

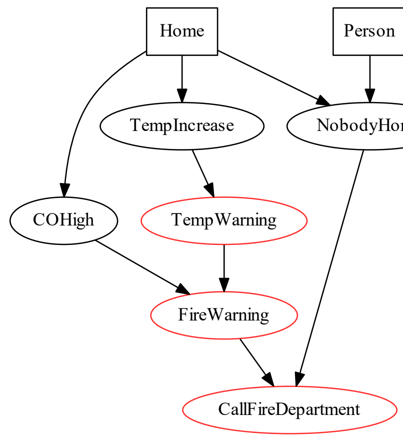


Figure 2: Priority problems

3.1 Rule acyclicity detection

Rule acyclicity is not a problem in itself but it can cause problems, such as infinite loops that flood the system with useless events. Thus, we decided to detect those cycles and inform the engineer about them.

First of all, we build the graph corresponding to the Esper EPL code. As mentioned before, a node is created for each pattern. Edges are created as follows. For each pattern R, we analyze its *from* pattern to detect the events involved in its matching phase, track the patterns in which these events are produced, and create an edge in the graph associating the pattern R with each one with which it has dependencies. In a second step, once the dependency graph is built, we apply the Kosaraju-Sharir algorithm [21] to detect cycles.

Imagine that due to copy-paste, the code of our example contains errors. Imagine that the rule `TempIncrease` stated in its from pattern that depends on `FireWarning` events, and the `COHigh` rule consumed its own events. The generated graph following our approach would be the one presented in Figure 1. Note how cycles are depicted in red to catch the engineer's attention.

3.2 Rule race detection and solving

In Esper EPL, priorities can be assigned to patterns by using the label `@Priority(n)` where n is a non-negative integer. By default, if the priority is not specified, the CEP engine assumes that the pattern has the highest priority, which is 0. Users may assign priorities to rules in order to avoid the rule race conditions described in Sect. 2.2.

The lack of priority assignation, or errors in their assignments, can lead to race conditions between patterns—specially those that consume complex events. In order to detect these conditions, we also rely on the graph representation of the Esper EPL programs.

Firstly, the acyclicity analysis has to be performed. If it detects cycles, we cannot make any conclusion about the correctness of the priorities. On the contrary, if the graph does not contain cycles, we are able to generate the priority of each node given their dependencies. In order to do so, we make use of the graph topological order of their nodes. The priority of each node is the maximum of the priorities of its predecessors plus one.

Once we calculate the priority that each pattern should have, we compare them with the priorities assigned by the user. With the result of the analysis we generate two files. One with the program dependency graph that, when interpreted and visualized, shows in red every node for which the priorities do not coincide, and one with the Esper EPL code containing the same code as the original file in which the priorities have been fixed.

As an example, Figure 2 shows the graph generated from the code in Listing 1. We can see how, due to the lack of priorities (which the Esper engine assumes as priority 0), patterns `TempWarning`, `FireWarning` and `CallFireDepartment` are suggested for review. Listing 2 shows the excepts of the new generated Esper file in which the *right* priorities are explicitly defined. In this context, *right* means that the rule priorities ensure the correct order of triggering, in order to avoid rule race conditions.

4 THE CEPA TOOL

We have created an Eclipse plug-in called CEPA[1,2] to give support to our approach. We have used Xtext,[3]which allows the definition of Domain-Specific Languages (DSL) and offers a parser, typechecker and compiler for free; and GEF, [4] which permits the integration of the Graphviz[5] tool for graph visualization within Eclipse.

Using Xtext, we have defined a simplified grammar that parses any valid Esper EPL program, and builds an Abstract Syntax Tree (AST) that focuses only on the parts of interest needed to extract the pattern dependencies. In this way, the AST only contains information about the `select` and `from pattern` parts of the CEP programs, ignoring the rest of the language syntax.

Given the AST provided by Xtext, we have built a Java program that creates a graph containing the primitive events and pattern dependencies and perform the analyses mentioned in Section 3.

If the acyclicity analysis detects cycles, we use the Xtend[6] code generator to automatically generate the textual code that represent

[1]https://github.com/Garlo13/CEPStaticAnalysis
[2]CEPA stands for *CEP Analysis*. Cepa is also the Spanish word for *strain*.
[3]https://www.eclipse.org/Xtext/
[4]https://www.eclipse.org/gef/
[5]http://www.graphviz.org/
[6]http://www.eclipse.org/xtend/

Listing 2: Esper EPL patterns with priorities

```
@Name("TempIncrease")
insert into TempIncrease
@Priority(0)
select ...

@Name("TempWarning")
insert into TempWarning
@Priority(1)
select ...

@Name("COHigh")
insert into COHigh
@Priority(0)
select ...

@Name("FireWarning")
insert into FireWarning
@Priority(2)
select ...

@Name("NobodyHome")
insert into NobodyHome
@Priority(0)
select ...

@Name("CallFireDepartment")
insert into CallFireDepartment
@Priority(3)
select ...
```

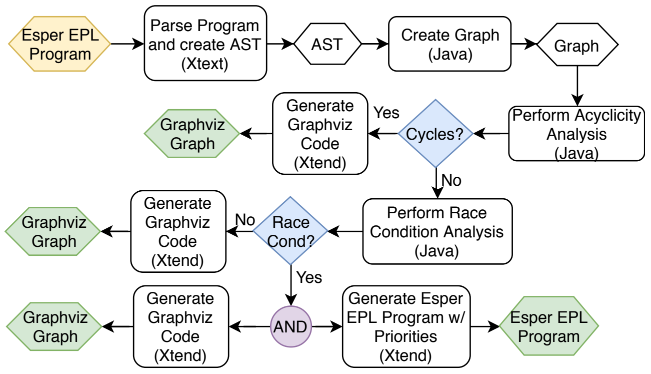

Esper EPL Program
Parse Program and create AST (Xtext)
AST
Create Graph (Java)
Graph
Perform Acyclicity Analysis (Java)
Cycles?
Yes
Generate Graphviz Code (Xtend)
Graphviz Graph
No
Perform Race Condition Analysis (Java)
Race Cond?
No
Generate Graphviz Code (Xtend)
Graphviz Graph
Yes
AND
Generate Graphviz Code (Xtend)
Graphviz Graph
Generate Esper EPL Program w/ Priorities (Xtend)
Esper EPL Program

Figure 3: CEPA workflow

the pattern dependency graph showing the cycles with red arrows. This graph is interpreted by Graphviz and shown to the user.

In case the acyclicity analysis does not report the presence of cycles, the race condition analysis is carried out. If rule race conditions are detected, Xtend is used to generate two files: one containing the Graphviz graph with the result of the analysis, and another one with the Esper code with the right priority for each pattern. If no rule race condition is detected, only one file is generated with the Graphviz graph.

For clarity, Figure 3 depicts the CEPA workflow diagram, with the main steps of its internal behavior. For illustration purposes, Figure 4 presents a screenshot of our tool, showing its graphical interface to the user.

5 VALIDATION

In order to validate our approach, we used a real CEP application that currently runs across the Andalusian region in southern Spain, and controls the quality of the air in real time, using the index proposed by the U.S. Environmental Protection Agency (EPA) [17]. The raw data is obtained from the Andalusian Regional Government's sensor network, which is composed of 61 sensor stations. Each station measures every 10 minutes six air pollutants: carbon monoxide, ozone, nitrogen dioxide and sulfur dioxide. The EPA proposes the analysis of these pollutant every 1, 8 or 24 hours and express the results in a 6-grade scale, from Good to Hazardous.

The definition of this CEP program[7], which is written in Esper EPL [8], contains 43 patterns. The information about each pollutant measurement is aggregated by one pattern, and other six patterns per pollutant determine the air quality grade for it. This makes seven patterns for every pollutant, and therefore the CEP program uses 42 patterns to analyze the six pollutants of interest. A final pattern, `AirQualityLevel`, calculates the global air quality level by computing the maximum of the grades obtained for each pollutant.

Interestingly, we discovered that, although there are no cycles in this program, these 43 patterns suffered from rule race conditions due to dependencies between the patterns. These dependencies may cause that some patterns are triggered before the patterns on which they depend, hence leading to erroneous results and conclusions about the quality of the air in the region.

Our tool was able to detect this error and to propose the correct assignment of priorities to the application CEP patterns that permitted solving the rule-race condition problem.

6 RELATED WORK

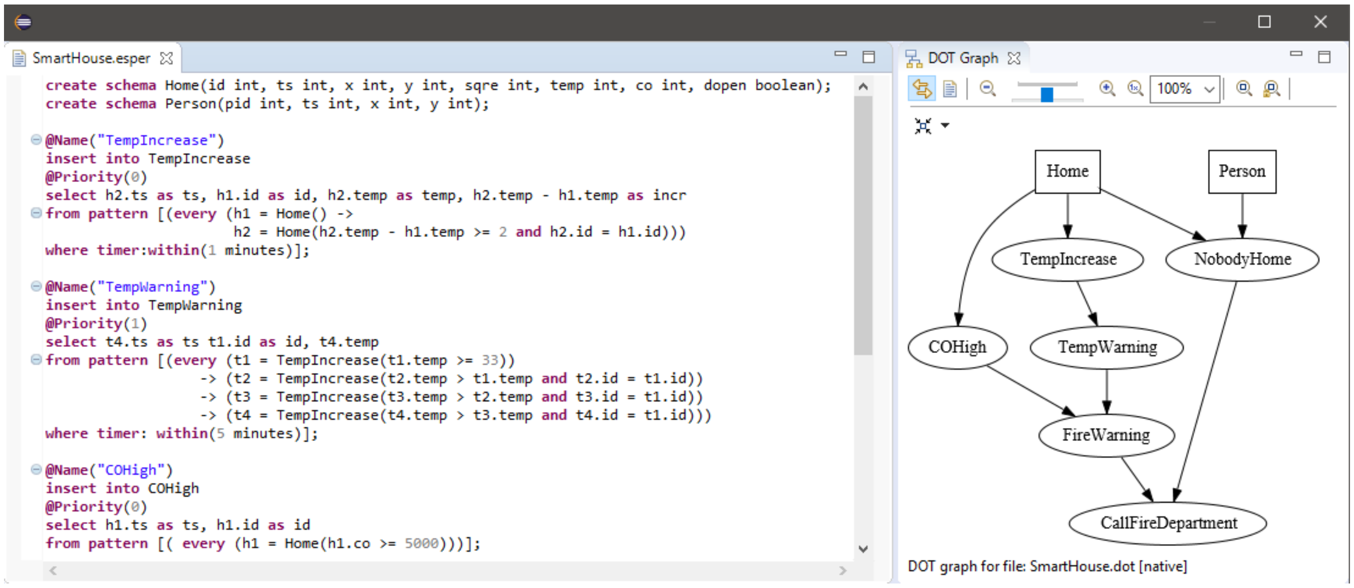
Rabinovich et al. [19] analyze the behavior of CEP applications using static and dynamic techniques using formal methods. Cugola et al. [7] transform the property checking task into a set of constraint solving problems. Although these approaches offer more kinds of analyses than we do, for the ones we provide support, given the exploratory nature of CSP solutions, their approaches are not as efficient as ours.

In the context of Active DBMSs [22] there is a group of works related to the analysis of rule-based reactive systems, including confluence [2, 5], termination [3], and correctness [11, 14]. Our work is similar to those, applied to the context of CEP systems.

Race detection in event-driven systems have been previously studied [12, 20]. The novelty of our approach is that here we deal with race conditions between complex events generated by the CEP rules, and therefore the order in which the rules are triggered has effect on the final output of the program. This is not the same as possible race conditions among the simple events that arrive to the data stream.

There are other works modeling CEP systems using Petri Nets. An an example, Ahmad et al. [1] model CEP applications using Timed Net Condition Event Systems (TNCES), a formalism based on Timed Petri Nets. This approach checks whether the system satisfies certain properties or not. However, a simple EPL pattern such as `every (A -> B)`, is transformed into a vast and hardly readable TNCES model that is difficult to understand and debug.

[7] Available from http://atenea.lcc.uma.es/projects/FormalizingCEP.html



```
create schema Home(id int, ts int, x int, y int, sqre int, temp int, co int, dopen boolean);
create schema Person(pid int, ts int, x int, y int);

@Name("TempIncrease")
insert into TempIncrease
@Priority(0)
select h2.ts as ts, h1.id as id, h2.temp as temp, h2.temp - h1.temp as incr
from pattern [(every (h1 = Home() ->
                      h2 = Home(h2.temp - h1.temp >= 2 and h2.id = h1.id)))
where timer:within(1 minutes)];

@Name("TempWarning")
insert into TempWarning
@Priority(1)
select t4.ts as ts t1.id as id, t4.temp
from pattern [(every (t1 = TempIncrease(t1.temp >= 33))
                 -> (t2 = TempIncrease(t2.temp > t1.temp and t2.id = t1.id))
                 -> (t3 = TempIncrease(t3.temp > t2.temp and t3.id = t1.id))
                 -> (t4 = TempIncrease(t4.temp > t3.temp and t4.id = t1.id)))
where timer: within(5 minutes)];

@Name("COHigh")
insert into COHigh
@Priority(0)
select h1.ts as ts, h1.id as id
from pattern [( every (h1 = Home(h1.co >= 5000)))];
```

Figure 4: Screenshot of the CEPA tool showing its graphical interface to the user.

Finally, some of the issues discussed here were identified in our previous work [4]. This paper reports on the tool we have developed to implement the analyses hinted in that work, and how we have realized it.

7 CONCLUSIONS

In this contribution, we present our approach to statically analyze Esper EPL programs, and to automatically check two properties: rule acyclicity and rule race conditions. Furthermore, when rule race conditions are detected, we suggest how to repair the Esper code to avoid non-confluence problems. We have built an Eclipse tool, called CEPA, that offers support for conducting such tests.

In the future we plan to increase the list of analyses supported by our tool, such as for example dead-end detection. Furthermore, we would like to extend our current scope, by integrating into our tool the dynamic analyses we explored in [4]. Finally, although the performance of the algorithm currently used for detecting cycles in the graph is acceptable, we would also like to evaluate other algorithms [16], in particular the one by Johnson [13].

Acknowledgements. We would like to thank the reviewers of this paper for their helpful suggestions and comments. This work has been funded by Spanish Research Project TIN2014-52034-R.

REFERENCES

[1] Waheed Ahmad, Andrei Lobov, and Jose L. Martinez Lastra. 2012. Formal modelling of Complex Event Processing: A generic algorithm and its application to a manufacturing line. In *Proc. of INDIN'12*. IEEE, 380–385.
[2] Alexander Aiken, Joseph M. Hellerstein, and Jennifer Widom. 1995. Static Analysis Techniques for Predicting the Behavior of Active Database Rules. *ACM Trans. Database Syst.* 20, 1 (1995), 3–41.
[3] Elena Baralis, Stefano Ceri, and Stefano Paraboschi. 1998. Compile-Time and Runtime Analysis of Active Behaviors. *IEEE Trans. Knowl. Data Eng.* 10, 3 (1998), 353–370.
[4] Loli Burgueño, Juan Boubeta-Puig, and Antonio Vallecillo. 2018. Formalizing Complex Event Processing Systems in Maude. *IEEE Access* 6 (2018), 23222–23241. https://doi.org/10.1109/ACCESS.2018.2831185
[5] Sara Comai and Letizia Tanca. 2003. Termination and Confluence by Rule Prioritization. *IEEE Trans. Knowl. Data Eng.* 15, 2 (2003), 257–270.
[6] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.* 44, 3 (2012), 15:1–15:62.
[7] Gianpaolo Cugola, Alessandro Margara, Mauro Pezzè, and Matteo Pradella. 2015. Efficient analysis of event processing applications. In *Proc. of DEBS'15*. ACM, 10–21.
[8] EsperTech. (accessed 18 November 2017). Esper - Complex Event Processing. http://www.espertech.com/esper/.
[9] Opher Etzion and Peter Niblett. 2010. *Event Processing in Action*. Manning Publications.
[10] Event Processing Technical Society. 2011. *Event Processing Glossary, Version 2.0.* http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf.
[11] Esa Falkenroth and Anders Törne. 1999. How to construct predictable rule sets. In *Proc. of Real Time Programming'99*. 33–40.
[12] Chun-Hung Hsiao, Cristiano Pereira, Jie Yu, Gilles Pokam, Satish Narayanasamy, Peter M. Chen, Ziyun Kong, and Jason Flinn. 2014. Race detection for event-driven mobile applications. In *Proc. of PLDI'14*. ACM, 326–336.
[13] Donald B. Johnson. 1976. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.* 4, 1 (1976), 77–84.
[14] Sin-Yeung Lee and Tok-Wang Ling. 1999. *Verify Updating Trigger Correctness.* Springer, 382–391.
[15] David C. Luckham. 2002. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley.
[16] Prabhaker Mateti and Narsingh Deo. 1976. On Algorithms for Enumerating All Circuits of a Graph Related Databases. *SIAM J. Comput.* 5, 1 (1976), 90–99.
[17] D. Mintz. 2016. *Technical Assistance Document for the Reporting of Daily Air Quality - the Air Quality Index (AQI).* Technical Report EPA-454/B-16-002. U.S. Environmental Protection Agency. http://www.epa.gov/airnow/aqi-technical-assistance-document-may2016.pdf
[18] Nathalie Moreno, Manuel F. Bertoa, Gala Barquero, Loli Burgueño, Javier Troya, and Antonio Vallecillo. 2018. Managing Uncertain Complex Events in Web of Things Applications. In *Proc. of ICWE'18 (LNCS)*. Springer, 349–357. http://atenea.lcc.uma.es/projects/UCEP.html
[19] Ella Rabinovich, Opher Etzion, Sitvanit Ruah, and Sarit Archushin. 2010. Analyzing the behavior of event processing applications. In *Proc. of DEBS'10*. ACM, 223–234.
[20] Veselin Raychev, Martin T. Vechev, and Manu Sridharan. 2013. Effective race detection for event-driven programs. In *Proc. of OOPSLA'13*. ACM, 151–166.
[21] M. Sharir. 1981. A strong-connectivity algorithm and its application in data flow analysis. *Computers and Mathematics with Applications* 7 (1981), 67–72. Issue 1.
[22] J. Widom and S. Ceri. 1996. *Active database systems: Triggers and rules for advanced database processing.* Morgan Kaufmann.