IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 26, 2018, accepted April 24, 2018, date of publication April 27, 2018, date of current version May 16, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2831185

Formalizing Complex Event Processing Systems in Maude

LOLI BURGUEÑO[1], JUAN BOUBETA-PUIG[2], AND ANTONIO VALLECILLO[1]
[1]Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, 29071 Málaga, Spain
[2]Department of Computer Science and Engineering, University of Cádiz, 11519 Puerto Real, Spain

Corresponding author: Loli Burgueño (loli@lcc.uma.es)

This work was supported by the Spanish MINECO/FEDER Research Projects under Grant TIN2014-52034-R, Grant TIN2015-65845-C3-3-R, and Grant TIN2016-81978-REDT.

ABSTRACT Complex event processing (CEP) is a cutting-edge technology for analyzing and correlating streams of information about events that happen in a system, and deriving conclusions from them. CEP permits defining complex events based on the events produced by the incoming sources, to identify complex meaningful circumstances and to respond to them as quickly as possible. Such event types and patterns are defined using event processing languages. However, as the complexity of CEP programs grows, they become difficult to understand and to prove correct. This paper proposes a formal framework for the specification of CEP applications, using rewriting logic and Maude, to allow developers to formally analyze and prove properties of their CEP programs. Several case studies are presented to illustrate the approach, as well as a discussion on the benefits of using Maude and its toolkit for modeling and analyzing CEP systems.

INDEX TERMS Formal modeling, complex event processing, event processing language, rewriting logic, Maude.

I. INTRODUCTION

Complex Event Processing (CEP) is gaining acceptance in real-time distributed environments as a powerful technology for analyzing and correlating streams of information about events that happen, and deriving conclusions from them [1]–[4]. CEP is becoming very relevant in many different contexts such as the Internet of Things [5], [6], where many applications should process and react to events arriving from various kinds of sources including distributed sensors, wireless sensor and RFID networks, GPS, social media, etc. Other examples of these kinds of applications include monitoring systems for critical infrastructures [7], health care systems [8], inventory control and manufacturing applications [9], environmental monitoring [10], [11], stock market analysis [12], network analysis and surveillance [13], or social media data aggregation [14], [15]. Unlike other stream processing systems, CEP permits defining complex events or patterns on top of the basic primitive events, to identify complex meaningful circumstances and respond to them as quickly as possible. Such event types and event patterns are defined using Event Processing Languages (EPLs).

The wide adoption of CEP systems has also introduced some challenges to these kinds of approaches. In the first place, the complexity of CEP programs is significantly growing, and hence they are becoming more difficult to understand, maintain and prove correct. There is the need to check for the occurrence of semantic errors in the event patterns definitions, and to validate that CEP programs behave as expected—e.g., they properly identify the events of interest to the system developer and no others, or that they do not miss any relevant event. The design of CEP programs still remains a challenging and error-prone task, since it requires developers to consider complex pattern dependencies and interactions [16]. The composition of CEP programs represents another challenge: what is the expected behavior of a CEP application when two or more programs are integrated to form a bigger one, or when new patterns are added to an existing CEP system? Finally, we also need to conduct some behavioral analysis on a CEP application prior to its deployment, such as simulation, to detect design errors and any other semantic anomalies so frequent in systems that exhibit stochastic behaviors.

In this paper, we investigate the use of rewriting logic [17], and its implementation in Maude [18], for giving semantics to CEP programs. Using Maude as a target semantic domain brings remarkable benefits, since Maude specifications are executable and permit multiple analyses,

including simulation. In particular, we propose a formal encoding of CEP concepts and mechanisms in Maude, which allows developers to formally analyze and prove properties of their CEP systems using the Maude toolkit. Several kinds of analyses are presented, both covering the static properties of the CEP patterns (acyclicity and confluence) and the statistical simulation of such systems.

Our approach is illustrated with a running example from the automotive industry, which process events coming from sensors of a motorbike (tire pressure, speed, location, etc.). We also discuss other systems that we have used to validate our proposal.

The structure of the paper is as follows. After this introduction, Sections II and III introduce CEP systems and Maude, respectively. Section IV describes our encoding of CEP concepts and mechanisms in Maude, and how to specify CEP applications using this representation. Section V shows the kinds of analyses that our proposal enables, and how to conduct them. This section also describes the validation experiments we have performed to assess the expressiveness, strengths and limitations of our proposal. Finally, Section VI relates our work to other similar approaches and Section VII concludes and outlines some future lines of work.

II. COMPLEX EVENT PROCESSING

CEP [2], [3] is a form of Information Processing [1] whose goal is the definition and detection of situations of interest, from the analysis of low-level event notifications (also called raw events) [16]. Following the Event Processing Technical Society [19], we use the *simple* events term to refer to the low level primitive event occurrences, and *complex* events to those that summarize, represent, or denote a set of other events. *Derived* events are a particular kind of complex events, which are generated as a consequence of applying a process or method to one or more other events. In summary, CEP systems analyze streams of *simple events* to detect occurrences of *complex events*, which represent the high-level situations of interest to the CEP modeler, using declarative *patterns* that define the complex (derived) events from other simple or complex events, their content, and temporal relations.

Although several CEP systems and languages exist, they all share the same basic concepts, mechanisms and structure. These will be briefly introduced below, using a running example. The description of CEP concepts and mechanisms is made in general, but for clarity we will write the rules in one particular EPL, called Esper EPL [20], which is a representative example of the EPLs that extend SQL to define CEP events and patterns. This section is not intended to serve as an Esper tutorial; the interested reader can consult the language documentation for particular details.

A. RUNNING EXAMPLE

Let us assume a fleet of motorbikes equipped with sensors that produce real-time information about their state, including the pressure of their two tires, their location and speed, and whether the driver is on the seat or not. We are interested in monitoring them, being able to detect in real time flat tires, vehicle crashes or accidents. In particular, we are interested in the following complex events:

- **BlowOutTire**: The pressure of one of the tires of a moving motorbike goes down from more than 2.0 BAR to less than 1.2 BAR in less than 5 seconds.
- **Crash**: The speed of a motorbike goes from more than 50 km/h to 0 km/h in less than 3 seconds.
- **DriverLeftSeat**: The seat sensor detects that the motorbike driver has left the seat.
- **Accident**: A moving motorbike suffers a blow out of one of its tires, then a `Crash` event is detected, and the driver is thrown out, everything within less than 3 seconds.
- **AccidentsReport**: A complex event with the number of `Accidents` per day and location.
- **DangerousLocation**: This event is raised every time 100 events of type `Crash` are detected in a given location.

B. EVENTS

In CEP, every event (simple or complex) has a *type* and a set of *attributes*. Events happen instantaneously, and they all have an attribute *timestamp* with information about the moment in time at which they occur.

For example, simple events of type `Motorbike` with the information received from the sensors are represented by tuples with the information about the timestamp, the motorbike id, the name of the current location, the speed in Km/h, the pressure of the two tires measured in BARs, and whether the driver is on the seat or not. Listing 1 shows examples of these simple events. Timestamps are expressed using the POSIX time convention, which is roughly the number of seconds that have elapsed since January 1, 1970 [21].

```
Motorbike(1488326400,1,"Cadiz",100,3.1,3.1,true)
Motorbike(1488326401,1,"Cadiz",90,3.1,3.1,true)
Motorbike(1488326402,2,"Malaga",62,3.01,3.01,true)
Motorbike(1488326402,1,"Cadiz",0,3.1,3.1,true)
Motorbike(1488326403,2,"Malaga",70,3.01,3.01,true)
Motorbike(1488326403,1,"Cadiz",0,3.07,3.07,false)
```

LISTING 1. Examples of `Motorbike` simple events.

C. PATTERNS

A CEP pattern defines a complex event, by means of a pattern that combines other events. Whenever the pattern is detected in the stream (i.e., it is *satisfied* by the stream events), the complex event is created. In the rest of this section, we will identify and describe the most basic and representative types of CEP patterns.

1) SELECTION PATTERNS

The simplest pattern permits creating complex events every time a given simple event is detected. For example, the pattern in Listing 2, named `GhostRider`, creates a `GhostRider` event every time a motorbike is detected to be moving and with no driver on top.

```
@Name('GhostRider')
insert into GhostRider
select e.timestamp as timestamp,
       e.motorbikeId as motorbikeId
from pattern [every
       e = Motorbike(e.seat=false and e.speed>0)]
```

LISTING 2. `GhostRider` **pattern.**

The use of the `every` operator ensures that a `GhostRider` event is created every time a `Motorbike` event satisfies the pattern. Otherwise only one complex event will be created the first time a simple event satisfies the pattern. In the pattern, label `e` is an alias that refers to an expression (in this case, the `Motorbike` event that has occurred) that can be used in other sub-expressions. Event `GhostRider` defines two attributes, `timestamp` and `motorbikeId`, whose values are taken from those of the `Motorbike` event that has triggered the creation of the `GhostRider` event.

2) WINDOWS

We can assign *windows* to patterns, restricting their scope. Windows could refer to specific time intervals or number of occurrences of particular events. For example, Listing 3 shows the pattern `AccidentsReport`, which creates an event with the number of accidents that have happened daily in one location.

```
@Name('AccidentsReport')
insert into AccidentsReport
select current_timestamp() as timestamp,
       a1.location as location,
       count(*) as count
from pattern [every a1 = Accident].
         win:time_batch(86400 seconds)
group by a1.location
```

LISTING 3. `AccidentsReport` **pattern.**

It uses a *batch time window* whose starting and ending points are fixed. In this case, it is triggered every day (86400 seconds). As in SQL, the query is grouped according to its location by means of the `group by` operator in the pattern.

We can also define *sliding time windows* whose ending time is the `T` timestamp of the event being considered, and its starting time is `T-L`, being `L` the duration of the window.

Similarly, *event windows* permit referring to sets of particular events of a given size (the *window size*), e.g. every 100 accidents. Event windows can be either *batch* or *sliding*, too. For example, in Listing 4, the pattern

```
@Name('DangerousLocation')
insert into DangerousLocation
select current_timestamp() as timestamp,
       a1.location as location
from pattern [every a1 = Crash(a1.location="Cadiz")]
       .win:length_batch(100)
group by a1.location
```

LISTING 4. `DangerousLocation` **pattern.**

`DangerousLocation` uses a batch window of size 100 to raise this kind of event, signaling a location as dangerous.

3) TEMPORAL SEQUENCING OF EVENTS

An important CEP operator is *followedBy* ("`->`"), which introduces a temporal ordering between two events. Events related by this operator do not need to be consecutive: "`A -> B`" only implies that event `A` occurs some time before `B`, i.e., the timestamp of `A` precedes that of `B`.

For instance, the pattern in Listing 5 creates an event of type `DriverLeftSeat` every time the seat of a motorbike is left by the driver, i.e., its state changes from true to false.

```
@Name('DriverLeftSeat')
insert into DriverLeftSeat
select current_timestamp() as timestamp,
       a2.motorbikeId as motorbikeId,
       a2.location as location
from pattern [every
      (a1 = Motorbike(a1.seat = true)
       -> a2 = Motorbike((a2.seat = false)
      and (a1.motorbikeId = a2.motorbikeId))]
```

LISTING 5. `DriverLeftSeat` **pattern.**

4) PATTERN COMBINATION

Event patterns can be combined in different ways by using logical operators (OR, AND, etc.) and temporal connectors (Until, While, etc.), among others. In addition, windows can be combined restricting their scope. This is needed, for instance, to specify the pattern `Crash` shown in Listing 6.

```
@Name('Crash')
insert into Crash
select current_timestamp() as timestamp,
       a2.motorbikeId as motorbikeId,
       a2.location as location,
       a1.speed as initialSpeed,
       a2.speed as finalSpeed
from pattern [every
       a1 = Motorbike(a1.speed >= 50)
       -> a2 = Motorbike((a2.speed = 0) and
       (a1.motorbikeId = a2.motorbikeId))
       where timer:within(3 seconds)]
```

LISTING 6. `Crash` **pattern.**

The `timer:within` expression permits limiting the pattern lifetime, requiring the second event to follow the first one within 3 seconds.

5) HIGHER-ORDER EVENTS

Higher-order complex events can be defined when the pattern that specifies the event makes use of other previously defined complex events. This is needed, for instance, to produce events of type `Accident` (Listing 7), which are defined using three complex events (`BlowOutTire`, `Crash`, and `DriverLeftSeat`) that were added to the stream by previous patterns.

The pattern in Listing 7 uses the `every-distinct` operator, which behaves like `every` but selecting only one unique occurrence of the events that fulfill the condition. It also shows a sequence of *followedBy* operators.

```
@Name('Accident')
insert into Accident
select current_timestamp() as timestamp,
       a3.motorbikeId as motorbikeId,
       a3.location as location
from pattern [every-distinct
       (a1.motorbikeId, a1.timestamp)
       a1=BlowOutTire
       -> (a2=Crash(a1.motorbikeId=a2.motorbikeId)
       -> a3=DriverLeftSeat(a1.motorbikeId=
       a3.motorbikeId))
       where timer:within(3 seconds)]
```

LISTING 7. `Accident` **pattern.**

III. MAUDE

Maude [18] is a high-level language that integrates an equational style of functional programming with rewriting logic computation. It supports membership equational logic and rewriting logic specification of systems [22], and provides an efficient interpreter able to execute different kinds of specifications. This section briefly describes those features needed for understanding the paper; the interested reader is referred to [18] for further details.

Rewriting logic [17] is a logic very appropriate for the specification of concurrent computations. Within this scope, a system is axiomatized by a *rewrite theory* (Σ, E, R), where (Σ, E) is an equational theory describing its set of *states* in terms of an algebraic data type associated to an initial algebra (Σ, E), and R is a collection of rewrite rules. Maude's underlying equational logic is membership equational logic [23], a Horn logic whose atomic sentences are equalities $t = t'$ and *membership assertions* of the form $t : S$, stating that a term t has sort S.

For example, Maude module `NAT` in Listing 8 defines the natural numbers (with sorts `Nat` of natural numbers and `NzNat` of nonzero natural numbers), using the Peano notation. Operations zero (`0`) and successor (`s_`) act as constructors for the type (note the `[ctor]` attribute). The behavior of the sum operation (`_+_`) is specified by two equational axioms on the constructors. Note the use of the *mixfix* syntax in the definition of operators `s_` and `_+_` (underscores indicate placeholders for arguments).

```
fmod NAT is
  sorts NzNat Nat .
  subsort NzNat < Nat .
  op 0 : -> Nat [ctor] .
  op s_ : Nat -> NzNat [ctor] .
  op _+_ : Nat Nat -> Nat [assoc comm] .
  vars M N : Nat .
  eq 0 + N = N .
  eq s M + s N = s (M + N) .
endfm
```

LISTING 8. Example of a module.

If a functional specification is terminating, confluent, and sort-decreasing, then it can be executed [24]. Maude uses the equations as simplification rules from left to right, until a canonical form is found. Some operator attributes can be used to express some common properties of the operations being specified, such as associativity (`assoc`), commutativity (`comm`), identity (`id`), and idempotence (`idem`). Maude performs simplification using the equational theories provided by such attributes. The above specifications must therefore be understood in the more general context of simplification *modulo* such equational theories.

Rewrite rules are written as "`crl [L] : T => T' if Cond`", where L is the rule label, T and T' are terms, and *Cond* is a condition. These rules describe the local, concurrent transitions that are possible in the system. Then, whenever a part of the system state fits the pattern T, then it can be replaced by the corresponding instantiation of T'. The guard *Cond* serves as a blocking precondition, i.e., a conditional rule can be fired only if its condition holds.

If more than one rule can be triggered at one moment in time, the system can nondeterministically select any of them.

Conditions are written as "$EqCond_1$ /\ ... /\ $EqCond_n$" where each of the $EqCond_i$ is either an ordinary equation $t = t'$, a *matching equation* $t := t'$, a sort constraint $t : s$, or a term t of sort `Bool`, abbreviating the equation t `= true`. In the execution of a matching equation $t := t'$, the variables of the term t become instantiated by *matching* it against the canonical form of term t'. These conditions are commonly used in our approach to define the guards of the Maude rules, which represent the corresponding CEP patterns.

A. OBJECT-ORIENTED SPECIFICATIONS: FULL MAUDE

Maude also supports the specification of object-oriented concurrent systems. Classes are defined using the syntax `class` C `|` $a_1 : S_1$, ..., $a_n : S_n$, where C is the class name, a_i are the attribute names, and S_i their types. Objects of a class C are structures of the form $< O : C \mid a_1 : v_1, \ldots, a_n : v_n >$, where O is the name of the object, and v_i are the values of its attributes.

With this, the state of a system has the structure of a bag of objects. Maude's predefined sort `Configuration` represents these bags of Maude objects, with `none` as empty configuration and the empty syntax operator "`__`" as union of configurations, as can be seen in Listing 9.

```
sort Configuration .
subsort Object < Configuration .
op none : -> Configuration [ctor] .
op __ : Configuration Configuration ->
          Configuration [ctor assoc comm id: none] .
```

LISTING 9. Maude's sort Configuration.

The state of the system evolves by the effect of the rewriting rules, which describe the permitted transitions between configurations.

For example, Listing 10 shows a module `BANK` that specifies a class `Account`, with an attribute `balance` of sort integer (`Int`) that represents the current balance. Another class `Withdraw` models the action of a money withdrawal; its attributes are the object identifier (of sort `Oid`) of the subject account, and the amount to withdraw. A conditional rule describes the behavior of the system. Rule `debit` specifies a local transition when the system has an object `A` of class `Account` and a `Withdraw` object requesting to withdraw

```
(omod BANK is
  class Account | balance : Int .
  class Withdraw | acc : Oid, amount : Int .
  vars A W : Oid .
  vars M Bal  : Int .
  crl [debit] :
    < W : Withdraw | acc : A, amount : M >
    < A : Account  | balance : Bal >
    => < A : Account | balance : Bal - M >
  if M <= Bal .
endom)
```

LISTING 10. Example of rewriting rules.

an amount smaller or equal than the balance of A; as a result of the application of such a rule, the object representing the action is consumed, and the balance of the account is modified.

B. TIMED SPECIFICATIONS: REAL-TIME MAUDE

Real-Time Maude [25] is an extension of Maude for the formal specification of distributed object-oriented real-time systems. Sort Time specifies time domain, which can be considered a commutative monoid (Time, 0, +, <). Some predefined modules specifying useful time domains, namely natural numbers and nonnegative rational numbers, are provided too. These modules define a supersort TimeInf that extends the sort Time with an infinity value INF.

The state of a system is represented in Real-Time Maude by terms of sort System. The user may specify the data type of time values as considered most appropriated. Therefore, time can be either discrete, which is recommended to specify real-time systems, or dense, which is often used to model hybrid systems.

One common way of specifying how the system evolves with time, is in terms of operations delta and mte [25].

The delta operation determines, for each object in the given configuration, its evolution after a period of time has passed. The second operation, mte (maximum time elapse), determines the maximum amount of time within which no timed action occurs in the system.

Given delta and mte operations, a unique [tick] rule is enough to manage time (see Section IV-B). The user needs to give equations for the delta and mte operators on those objects affected by the passage of time.

IV. ENCODING CEP APPLICATIONS IN MAUDE

This section describes how CEP concepts and mechanisms are encoded in Maude, and how to specify CEP applications using this representation. These encodings will be explained in the following subsections, and illustrated with the concrete Motorbike example.

A. STRUCTURAL ASPECTS

1) EVENTS

Given the nature of CEP events, they can be naturally modeled by objects instances, which also have a type (given by the class they belong to) and a set of attributes (defined in their classes, too). In our approach, all events (simple and complex) will inherit from class Event (Listing 11), which also provides an operation ts() to get its timestamp.

```
class Event | ts : Time .
op ts : Object -> Time .
eq ts(< O : Event | ts : T >) = T .
```

LISTING 11. Class Event.

Example: The (simple) events produced by a motorbike can be simply modeled by class Motorbike (Listing 12). This class inherits from Event and defines several attributes, which faithfully correspond to the ones defined for the CEP event it represents. The rest of the events are defined similarly.

```
class Motorbike |
  id : Int,
  location : String,
  speed : Int,
  seat : Bool,
  tirePressure1 : Rat,
  tirePressure2 : Rat .
subclass Motorbike < Event .
```

LISTING 12. Class Motorbike.

2) EVENT STREAMS

Class Stream (Listing 13) represents streams (an application can deal with more than one stream), as a list of events. Operation insert() inserts objects in that list, ordered by their timestamps.

```
class Stream | events : List{Object} .
op insert : Object List{Object} -> List{Object} .
eq insert (OB, nil) = OB .
eq insert (OB1, (L ; OB2)) =
     if ts(OB1) > ts(OB2) then L ; OB2 ; OB1
     else insert(OB1, L) ; OB2
     fi .
```

LISTING 13. Class Stream.

3) EVENT FACTORIES

One significant feature of our proposal is that all kinds of events have an associated *factory* object. In case of simple events, their factories can be used to read them from external sources and to add them to the stream of Maude objects that represent them; or to generate their instances in case we want to simulate their occurrences (see Section V). In the case of complex events, factory objects are used to specify the metadata used in the rules that create them: window duration or length, etc.

There are three kinds of factory objects depending on the kind of window defined for the event: HistoryFactory, SizeFactory and TimedFactory. The first two are used for (resp. time and size) sliding windows. The third one is used for batch time windows. Listing 14 shows their specifications.

4) CEP SYSTEM

The complete system is modeled by a configuration of objects whose behavior is dictated by the Maude rules. In that

```
class HistoryFactory |
  startTime : Time,     *** starting time
  windowLength : Time,  *** duration of the window
  lastEvent : Time .    *** last event detected
class SizeFactory |
  startTime : Time,     *** starting time
  windowLength : Int,   *** # events in the window
  lastEvent : Time .    *** last event detected
class TimedFactory |
  wakeUpAt : Time .     *** when to wake up again
```

LISTING 14. Factory classes.

configuration, there is always one `Counter` object, used to create fresh object identifiers, and one `Clock` object. Their specification is shown in Listing 15.

```
sort System .
op `{_`} : Configuration -> System .
class Counter | n : Int .    *** To create fresh ids
class Clock | time : Time . *** Global clock
```

LISTING 15. Sort `System`.

B. TIME MODEL

Time is represented using the Real-Time Maude approach. Rule `[tick]` advances the clock to the next moment in time (defined by `NOW + T`) when an action is scheduled (see Listing 16 below). The value of `T` is calculated by the `mte` operation. As we can see, `mte` only needs to consider `TimeFactory` objects, calculating the next time one of these factory objects has to be awakened. The rest of the objects are not affected by time, as we shall later see, and therefore operation `delta` is not needed.

```
crl [tick] : { Conf < C : Clock | time : NOW > }
             => { Conf < C : Clock | time : NOW + T > }
             if T := mte(Conf, NOW) /\ T > 0 .
op mte : Configuration Time -> Time .
eq mte (< O : TimedFactory | wakeUpAt : T1 > Conf, T2)
       = min( T1 - T2 , mte(Conf, T2) ) .
eq mte (Conf, T) = INF [owise] .
```

LISTING 16. Rule `tick`.

Note that CEP modelers using our encoding do not need to worry about the time infrastructure, only use the `Clock` object in the Maude rules that represent their CEP patterns.

C. CEP PATTERNS

In our approach, every CEP pattern is mapped to one or more Maude rules, each one in charge of identifying the events that trigger the pattern—as specified in the left-hand side (LHS) of the rule—and the effects of such a pattern—as described in the right-hand side (RHS) of the Maude rule.

The concrete structure of the Maude rule will depend on the kind of CEP pattern being represented, as discussed next.

1) CEP PATTERNS WITH BATCH WINDOWS

These are the simplest rules, which follow the general structure described in Listing 17.

The rule is triggered when the wakeup time indicated by the event factory object coincides with the current time (`NOW`), and the conditions of the event (as indicated in the `if` clause) are met. Note the use of variable assignment in the condition

```
crl [Event-X] :
< C : Clock | time : NOW >
< CO : Counter | n : N >
< S : Stream | events : L >
< F : X-Factory | ..., wakeUpAt : NOW >
=> < C : Clock | time : NOW >
   < CO : Counter | n : N + 1 >
   < S : Stream | events :
       insert( < N + 1 : X | ..., ts : NOW >, L ) >
   < F : X-Factory | ..., wakeUpAt : NOW + T >
if T := ... /\ ... .
```

LISTING 17. Rule structure for batch window patterns generating event `X`.

section (`if`), which also permits computing the value of state variables. The event is inserted in the stream, and the new wake up time is set in the factory object. Such a wakeup time is set to the length of the window, hence acting as a timer. Once the wakeup time indicated by the event factory object coincides with the current time (`NOW`), the rule explores the events of the stream in the time interval, and generates the complex event (if appropriate).

Example: Listing 18 shows Maude rule `[AccidentsReport]`, which creates an `AccidentsReport` event with the number of `Accident` events that have happened in a location in the last 24 hours (i.e., 86,400 seconds).

```
rl [AccidentsReport] :
< C : Clock | time : NOW >
< CO : Counter | n : N >
< S : Stream | events : L >
< F : AccidentsReportFactory | location : S1,
      wakeUpAt : NOW >
=> < C : Clock | time : NOW >
   < CO : Counter | n : N + 1 >
   < S : Stream | events : insert(
       < N + 1 : AccidentsReport | location : S1,
         num : getAccidents(L, S1, NOW - 86400, NOW),
         ts : NOW >, L ) >
   < F : AccidentsReportFactory | location : S1,
         wakeUpAt : NOW + 86400 > .
```

LISTING 18. `AccidentsReport` rule.

It uses the `Clock` object to check that the system time (`NOW`) coincides with the time at which `AccidentsReportFactory` object—a subclass of `TimedFactory`—was supposed to wake up. If so, it looks for `Accident` events in the stream using the `getAccidents()` operation and inserts an `AccidentsReport` event in the stream with that number. Moreover, the `AccidentsReportFactory` object is reprogrammed to wake up 24 hours later. There will be one factory per location, hence the presence of attribute `location` in the factory. All factories will be created in the initial model of the system (see next subsection).

Let us show here the definition of operation `getAccidents()`, whose parameters are a stream, a location, and an interval of time defined by two time moments. It returns the number of `Accident` objects that exist in the stream for that location within that time interval. The Maude code is shown in Listing 19.

2) CEP PATTERNS WITH NO WINDOW

Some CEP patterns are aimed to detect circumstances that happen in the event stream, but with no time restriction.

```
op getAccidents : List{Object} String Time Time -> Int .
eq getAccidents ( nil, S1, T1, T2 ) = 0 .
eq getAccidents ( (< O : Accident |
     ts : T, location : S1 > ; L), S1, T1, T2 ) =
     if (T >= T1) and (T < T2) then
          1 + getAccidents ( L , S1, T1, T2 )
     else getAccidents ( L , S1, T1, T2 )
     fi .
eq getAccidents ((OB ; L), S1, T1, T2) =
     getAccidents ( L , S1, T1, T2 ) [owise] .
```

LISTING 19. Operation `getAccidents()`.

For example, the occurrence of a particular event, or a state change. For representing them, we make use of `HistoryFactory` objects, using the general structure shown in Listing 20.

```
crl [Event-Z] :
< C0 : Counter | n : N >
< S : Stream | events : L >
< F : Z-Factory | ..., startTime : T0, lastEvent : TLE >
=> < C0 : Counter | n : N + 1 >
   < S : Stream | events :
           insert( < N + 1 : Z | ..., ts : T1 >, L ) >
   < F : Z-Factory | startTime : T1 + 1, lastEvent : T1 >
if
   OB1 ; L1 := checkForZ( L, M, T0 ) /\
   T1 := ts(OB1)  .
```

LISTING 20. Structure of a rule representing a CEP pattern with no window.

Note that they do need the `Clock` object, because they traverse the stream independently of time. When the expected situation is found, the complex event is generated and inserted in the stream, and the pointer to the stream that indicates the position from where to start next time (`startTime`) is updated. In order to coordinate with the rest of the rules, which are also exploring the stream, attribute `lastEvent` serves as a pointer to the last event that the rule has analyzed (see Section IV-C.4).

Example: An example of a CEP pattern that involves a *followedBy* operator but with no time restriction corresponds to event `DriverLeftSeat`, that happens every time a driver leaves the seat (Listing 21).

```
crl [DriverLeftSeat] :
< C0 : Counter | n : N >
< S : Stream | events : L >
< F : DriverLeftSeatFactory | motorbikeId : M,
       startTime : T0, lastEvent : TLE >
=> < C0 : Counter | n : N + 1 >
   < S : Stream | events : insert(
        < N + 1 : DriverLeftSeat | motorbikeId : M,
                  location : S2, ts : T2 >, L ) >
   < F : DriverLeftSeatFactory | motorbikeId : M,
         startTime : T1 + 1, lastEvent : T1 >
if
   OB1 ; L1 := filterSeated( L, M, T0 ) /\
   T1 := ts(OB1) /\
   OB2 ; L2 := filterUnseated( L1, M, T1 ) /\
   T2 := ts(OB2) /\
   S2 := loc(OB2) .
```

LISTING 21. `DriverLeftSeat` rule.

This Maude rule checks for events where the driver was seated, *followed by* events where the driver was not seated. This is done in the guard of the action (`if`), using operations `filterSeated()` and `filterUnseated()`. The first one traverses the stream looking for the first event in which motorbike `M` has its driver seated, and whose timestamp (`T1`) is greater than `T0` (the starting time indicated in the factory event). Then, the second operation looks for the first event where the driver of motorbike `M` is unseated, and whose timestamp (`T2`) is greater than `T1`. If both events exist, a complex event `DriverLeftSeat` is added to the stream, and the rule is reprogrammed with the new starting time `T1 + 1`. As above, there should be a `DriverLeftSeatFactory` object per motorbike.

Operation `filterSeated()` is shown in Listing 22 for illustration purposes. The specification of `filterUnseated()` operation is similar.

```
op filterSeated : List{Object} Int Time -> List{Object} .
eq filterSeated ( nil, M, T ) = nil .
eq filterSeated ( (< O : Motorbike |
        ts : T1, seat : true, id : M > ; L), M, T ) =
   if (T1 >= T) then
      < O : Motorbike | ts : T1, seat : true, id : M > ; L
   else filterSeated (L, M, T)
   fi .
eq filterSeated ( (OB ; L) , M, T ) =
     filterSeated ( L, M, T ) [owise] .
```

LISTING 22. Operation `filterSeated`.

3) CEP PATTERNS WITH SLIDING WINDOWS

These kinds of patterns are also represented by Maude rules that use `HistoryFactory` objects. Listing 23 shows the common structure of these kinds of rules.

```
crl [Event-Y] :
< C0 : Counter | n : N >
< S : Stream | events : L >
< F : Y-Factory | ..., startTime : T0,
      windowTimeLength : T, lastEvent : TLE >
=> < C0 : Counter | n : N + 1 >
   < S : Stream | events : if (...) then
           insert( < N + 1 : Y | ..., ts : T2 >, L )
           else L fi >
   < F : Y-Factory | ..., startTime : T1 + 1,
         windowTimeLength : T, lastEvent : T1  >
if ... .
```

LISTING 23. Sliding window rule structure.

Whenever triggered, they explore the stream starting from the moment indicated by `startTime` attribute of the factory object, looking for the first event that satisfies the pattern. Once found, the rule checks for the rest of the conditions until the end of the window (whose length is indicated by the `windowLength` attribute). After exploring that window, the rule updates the values of the next window to explore, and inserts the event in the stream, if appropriate.

Note that these rules are not affected by time. We could have modeled them using timers, but we realized that this might require the clock to stop in almost every time unit. In this way, we always explore the stream starting from the first event that could satisfy the pattern. As in the previous rules, attribute `lastEvent` points to the last event that the rule has analyzed.

Example: One interesting example of a pattern that requires the use of a sliding window is the one that detects `Crash` events. Here, the Maude rule has to detect a motorbike driving at a speed above 50 km/h, *followed by* another event where the speed of that motorbike is 0, in a window of 3 seconds.

Unlike in previous cases, we use two Maude rules to encode that CEP rule. The first one (presented in Listing 24) is the obvious one that finds the event.

```
crl [Crash1] :
< C : Clock | time : NOW >
< CO : Counter | n : N >
< S : Stream | events : L >
< F : CrashFactory | startTime : T0,
      motorbikeId : M, windowTimeLength : T,
      lastEvent : TLE >
=> < C : Clock | time : NOW >
   < CO : Counter | n : N + 1 >
   < S : Stream | events : insert(
         < N + 1 : Crash | motorbikeId : M,
               initialSpeed : X1, ts : T2 >, L ) >
   < F : CrashFactory | startTime : T1 + 1,
         windowTimeLength : T, motorbikeId : M,
         lastEvent : T1  >
if
   OB1 ; L1 := filterSpeedGE(L, 50, M, T0) /\
   T1 := ts(OB1) /\
   OB2 ; L2 := filterSpeedLE(L1, 0, M, T1, T1 + T) /\
   T2 := ts(OB2) /\
   X1 := speed(OB1) .

*** Auxiliary operation filerSpeedGE
op filterSpeedGE : List{Object} Rat Int Time ->
                   List{Object} .
eq filterSpeedGE ( nil, X, M, T ) = nil .
eq filterSpeedGE ( (< O : Motorbike | ts : T1, speed : Y,
                   id : M > ; L) , X, M, T ) =
    if (T1 >= T) and (Y >= X) then
       < O : Motorbike | ts : T1, speed : Y, id : M > ; L
    else filterSpeedGE (L, X, M, T) fi .
eq filterSpeedGE ( (OB ; L) , X, M, T ) =
       filterSpeedGE ( L , X, M, T ) [owise] .
*** Auxiliary operation filerSpeedLE
op filterSpeedLE : List{Object} Rat Int Time Time
                   -> List{Object} .
eq filterSpeedLE ( nil, X, M, T1, T2) = nil .
eq filterSpeedLE ( (< O : Motorbike | ts : T, speed : Y,
                   id : M  > ; L) , X, M, T1, T2 ) =
    if (T >= T1) and (T < T2) and (Y <= X) then
       < O : Motorbike | ts : T, speed : Y, id : M >
    else filterSpeedLE (L, X, M, T1, T2) fi .
eq filterSpeedLE ( (OB ; L), X, M, T1, T2 ) =
       filterSpeedLE ( L , X, M, T1, T2 ) [owise] .
```

LISTING 24. `Crash1` **rule.**

Note how operation `filterSpeedGE` looks for the first appearance of the event that starts triggering the rule, being not upper bounded. In turn, operation `filterSpeedLE` searches for the second event within the bounds of the window.

However, this rule is not enough because we also need to cover the case whereby the first event is found, but the second one (`speed = 0`) is not found in the time window. In this case, we need to re-program the start time of the factory to the end of the sliding window as shown in Listing 25.

An interesting feature of our proposal is that it allows to easily implement *adaptive sliding windows*, whose size can be dynamically adapted depending on the incoming events, a particular kind of *frames* [26]. The static size of the window is a limitation of many applications, specially those that require *cleaning* the received events because they come

```
crl [Crash2] :
< CO : Counter | n : N >
< S : Stream | events : L >
< F : CrashFactory | startTime : T0,
      windowLength : T, motorbikeId : M >
=> < CO : Counter | n : N >
   < S : Stream | events : L >
   < F : CrashFactory | startTime : T1 + 1,
         windowLength : T, motorbikeId : M >
if
   OB1 ; L1 := filterSpeedGE( L, 50, M, T0 ) /\
   T1 := ts(OB1) /\
   L2 := filterSpeedLE(L1, 0, M, T1, T1 + T) /\
   L2 == nil .
```

LISTING 25. `Crash2` **rule.**

from unreliable sources and may have both duplicate readings and false positives [27]. In these contexts, a large window may induce false positive readings, and a small window cannot fill false negative readings. Cleaning methods such as SMURF [28] or any of its derivatives [29] can be used to dynamically adapt the window size for unreliable RFID data streams.

4) HIGHER-ORDER COMPLEX EVENTS

Let us describe how to deal with complex events created from other complex events (i.e., by CEP patterns), because they need special treatment when being represented in Maude. Whilst the occurrence of simple events is determined by external sources and we can assume that they are in the stream if they have indeed happened, this may not be true for higher-order complex events. For them, we need to wait until all the Maude rules that create the corresponding lower-order complex events upon which they depend have finished, before exploring the stream looking for their occurrence. For this, we need to include the factories of the corresponding events and make sure they have already explored the segment of the stream that we are about to explore. This is achieved by using attribute `lastEvent` of the factories of the lower-order objects upon which the higher-order event depends.

Example: This is illustrated in the case of the rule that creates `Accident` events. They require that the tire of a moving motorbike blows out (`BlowOutTire`), followed a `Crash` event, followed by a `DriverLeftSeat` event, all in less than 3 seconds. This is specified by the following Maude rules.

Note that, given that this CEP pattern depends on the occurrence of three events in a given interval, we need 3 Maude rules: one to cover the case when these three events happen (the `[Accident1]` rule in Listing 26); one for when the first two are found but the last one is not; and to specify the case when the first one happens but the other two do not. We do not need one for which the first one does not happen: this will simply not trigger any of them. The first of these rules is shown below. The other two are similar.

5) CEP PATTERNS WITH EVENT WINDOWS

So far we have considered the case of batch and sliding *time* windows. Representing CEP patterns with batch or sliding

```
crl [Accident1] :
< C0 : Counter | n : N >
< S : Stream | events : L >
< F : AccidentFactory | startTime : T0,
      windowTimeLength : T, motorbikeId : M >
< F1 : BlowOutTireFactory | lastEvent : TF1,
       motorbikeId : M >
< F2 : CrashFactory | lastEvent : TF2,
       motorbikeId : M >
< F3 : DriverLeftSeatFactory | lastEvent : TF3,
       motorbikeId : M >
=> < C0 : Counter | n : N + 1 >
   < S : Stream | events : insert(
         < N + 1 : Accident | motorbikeId : M,
                   location : S1, ts : T3 >, L ) >
   < F : AccidentFactory | motorbikeId : M,
         startTime : T3 + 1, windowTimeLength : T,
         lastEvent : T3 >
   < F1 : BlowOutTireFactory | lastEvent : TF1,
          motorbikeId : M >
   < F2 : CrashFactory | lastEvent : TF2,
          motorbikeId : M >
   < F3 : DriverLeftSeatFactory | lastEvent : TF3,
          motorbikeId : M >
if *** complex events have been created:
   (T0 + T) < min(min(TF1,TF2),TF3) /\
   OB1 ; L1 := filterBlowOutTire( L, M, T0 ) /\
   T1 := ts(OB1) /\
   OB2 ; L2 := filterCrash(L1, M, T1, T1 + T) /\
   T2 := ts(OB2) /\
   OB3 ; L3 := filterDriverLeftSeat(L2, M, T2) /\
   T3 := ts(OB3) /\
   S1 := loc(OB3) .
```

LISTING 26. `Accident1` **rule.**

```
op InitialModel : -> System .
eq InitialModel = {
 < 'c : Clock | time : 0 >
 < 'x : Counter | n : 1000 >
 < 's : Stream | events : nil >
 < 'f : MotorBikeFactory | >
 *** And now the Factory objects for the complex events
 < 101 : BlowOutTireFactory | motorbikeId : 1,
         tire : 1, startTime : 0, windowLength : 5 >
 < 102 : BlowOutTireFactory | motorbikeId : 1,
         tire : 2, startTime : 0, windowLength : 5 >
 < ... rest of Factory objects ... >
} .
```

LISTING 27. Initial model of the system.

```
rl [Motorbike] :
< C0 : Counter | n : N >
< C : Clock | time : NOW >
< O : MotorbikeFactory | >
< S : Stream | events : L >
=> < C0 : Counter | n : N >
   < C : Clock | time : NOW >
   < S : Stream | events :
     < 1 : Motorbike | ts : 1488326400, id : 1,
           location : "Cadiz", speed : 100, seat : true,
           tirePressure1 : 310, tirePressure2 : 310 > ;
     < 2 : Motorbike | ts : 1488326401, id : 1,
           location : "Cadiz", speed : 90, seat : true,
           tirePressure1 : 310, tirePressure2 : 310 > ;
     < ... rest of Motorbike events... >
   > .
```

LISTING 28. Rule for defining the set of initial Motorbike events.

event windows is similar to representing CEP patterns with sliding time windows. We just need to specify a rule that traverses the stream looking for events in that range (of number of events). Attribute `windowLength` of `SizeFactory` objects is precisely defined for that.

Example: The rule that generates a `DangerousLocation` event every time we reach 100 `Crash` events for a given location is an example of this kind of CEP patterns. The Maude rule is not shown here for brevity, it is similar to rule `AccidentsReport`, but using a `SizeFactory` object.

D. EXECUTING THE SYSTEM

Once the system specifications are written using this modeling approach, what we get is a rewriting logic specification of the system. Since the rewriting logic specifications produced are executable, this specification can be used as a prototype of the system, which allows us to simulate it.

For executing the specifications we need an initial model of the system, which is just a configuration of objects that contains the basic system objects (`Clock` and `Counter`), the stream with the initial simple events, and the factory objects for the complex events. In the motorbike example, it can be specified as in Listing 27.

Initially the stream of events can be empty, and a separate dedicated rule can be in charge of producing them. For instance, the rule presented in Listing 28. In the simplest case, we take the set of input simple events from the ones we have in a file, spreadsheet or database, and include them in the stream.

Note that the `MotorbikeFactory` object does not appear in the right-hand side of that Maude rule, which implies that the rule will be executed only once, and it will populate the stream that the rest of the rules will explore. Instead of providing all simple events in this way, we will later discuss in Section V-B.1 how it is also possible to continuously generate them using different strategies.

Starting from the initial model, the Maude toolkit allows the execution of our timed specifications using a set of commands for performing rewriting. In this way, an execution simulates one of the possible behaviors of the system from the initial configuration of objects (the initial state of the system).

For example, we may use the `rew` command (as shown below) to execute a fair rewrite of the system starting from an initial model configuration.

```
$ ./runCEP.sh accidents 256
```

Parameter `[10000000]` establishes the upper bound on the number of rewrite steps to perform, and `InitialModel` is the configuration with the initial model described above. Since our specification contains a large number of elements, it is recommended to limit the time bound—otherwise the execution may never finish because in general our specifications can be non-terminating. This limit can be achieved either by setting a bound on the number of rewrite steps, as done above, or by adding a constraint on the `[tick]` rule that limits the value of variable `NOW`, e.g., "`/\ (NOW < 5000000)`".

As a result of a rewrite command, what we get is the last state of the simulated behavior. In this last state of the system, the stream will contain all the events that have been produced.

V. ANALYSIS AND EXPERIMENTATION

Although the goal of EPLs is to provide mechanisms to make easier the CEP applications development, writing the patterns that specify the application logic is normally a complex and cumbersome task: it not only implies checking the cor-

rectness of the patterns regarding the desired behavior of the application, but also considering the dependencies and mutual interactions among them, and the possible assumptions on the environment in which the system works [16]. This is why some kind of formal support to the analysis and verification of CEP patterns is needed.

Some of the existing proposals to achieve formal analysis of CEP patterns use Linear Temporal Logic (LTL) model checking techniques to verify the properties of interest, e.g., [30], [31]. Although interesting from a theoretical point of view, the fact that CEP applications have to manage huge data sets and a significant number of patterns (usually one per complex event), makes this kind of (exhaustive) approach normally impractical: it suffers from memory problems due to rapid state-space explosions and hence prohibitive computational costs.

Similarly, approaches translating patterns and properties into logical formulas that can be analyzed by SAT/SMT solvers provide similar feasibility problems—although recent advances in this field are showing outstanding performance improvements (see, e.g., [16]).

Other approaches that make use of formalisms such as Petri Nets (e.g., [11]) also face the complexity issues inherent to CEP applications, leading to networks which are hard to produce and analyze.

In this section, we will report about the use of some analysis techniques available for Maude specifications in the realm of CEP patterns, beyond the simple execution described in the previous section. The fact that we have mapped CEP patterns directly into Maude rules and CEP events into Maude objects, opens the possibility of using the Maude mechanisms and tools for analyzing these kinds of applications.

More precisely, we discuss some static and dynamic analysis we can perform on the Maude specifications that represent the CEP application, in order to understand its behavior and to check that it is correct. Static analysis include checking for pattern acyclicity and race conditions. Both are needed because in general, the application of patterns can be neither deterministic nor confluent, as we shall later discuss, and therefore it is important to uncover as soon as possible potential errors in the CEP application. Dynamic analysis are useful to understand the behavior of the CEP applications, which is often probabilistic due to the stochastic nature of the environment in which they operate. We are interested in performing statistical analysis of CEP applications in order to estimate the probabilities of events satisfying a given logical formula, such as the occurrence of a certain circumstance of interest to the modeler. We also discuss some important aspects of these analyzes, such as their performance costs and the estimation of their confidence levels. Finally, we present other case studies we have used to validate our proposal.

A. STATIC ANALYSIS

There are some properties that can be statically checked in the Maude specification, which permit detecting interesting properties of the CEP pattern set, e.g., *Pattern Acyclicity* and *Pattern Race Conditions*. They both have to deal with the non-deterministic nature of rule-based systems (in case two or more patterns can be applied simultaneously), and confluence (the order of application of the patterns at a given moment in time may produce different results).

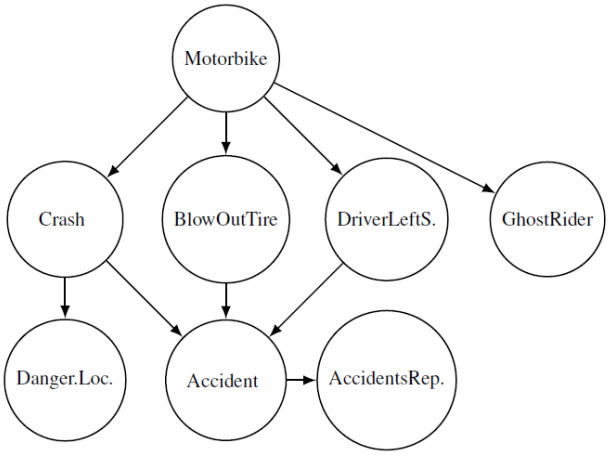

FIGURE 1. Directed graph of events for the Motorbike case example.

1) PATTERNS ACYCLICITY

When the number of CEP patterns is large and they have been independently defined by separate people at different times (as it happens, e.g., when the CEP application evolves over time and new patterns and events are added), there may be complex events that produce others, which in turn produce events of the former types—either directly or indirectly. These kinds of cycles can result in non-terminating and ill-defined pattern sets, with potential infinite loops. Of course, the presence of cycles does not necessarily imply errors in the patterns, but permits warning about potential errors.

To analyze this property we derive a directed graph (the *pattern dependency graph*) whose vertices represent the CEP patterns that produce the events and the edges represent dependencies among them: an edge between vertex P and Q means that pattern Q requires the events produced by pattern P to happen. We will say that pattern Q *is dependent on* P.

Example: Figure 1 shows the pattern dependency graph corresponding to the motorbike case study. Such a directed graph is built directly from the Maude rules, parsing the right hand side of the rules for the target events, and the left hand side and guards for their corresponding source events. In this case, no cycles occur in the graph.

2) PATTERN RACE CONDITIONS AND NON-CONFLUENT PATTERN SETS

Another potential source of errors in the application of the CEP patterns is due to the presence of *dependent* patterns that are evaluated in the wrong order. For example, event `AccidentsReport` depends on event `Accident`. Suppose that the system evaluates the stream of objects

at one moment in time, and it first evaluates pattern `AccidentsReport` before pattern `Accident`. The former pattern may miss the accidents that are produced by the latter.

In general, if pattern P_1 produces events of type a out of events of type b, and pattern P_2 produces events of type b from events of type c, the order of applications of patterns P_1 and P_2 is relevant: if an event c has happened, and pattern P_2 is applied before P_1, the CEP application will produce two events: b and a. However, if pattern P_1 is applied before P_2, only one event b will be produced since when P_1 is applied, P_2 has not produced event b that pattern P_1 requires. This leads to the need to identify *critical pairs* of patterns that can cause these kinds of problems.

To tackle this problem, some CEP engines, such as Esper, permit assigning *priorities* to patterns, which are in charge of deciding the precedence of application of patterns. However, a common mistake is forgetting the assignment of priorities, specially when the modeler is not conscious of the non-confluent behavior of the system he/she is specifying. This is precisely where our static analysis can play a relevant role, identifying such kinds of potential issues, and the need to address them in one way or another.

In Maude there are at least two ways of solving these kinds of problems. The first one is by using *strategies* at the metalevel [32], [33] that permit deciding the order in which the rules are triggered. Again, the pattern dependency graph permits determining the order of execution of the Maude rules, it is a matter of defining a topological order between the rules, based on the dependency graph.

The second approach to dealing with the pattern race problem is by explicitly tackling it at the rule level, as we did for instance in the specification of rule `Accident`. We showed how it is possible to check, using attribute `lastEvent` of the factory objects, that the initial rules have been executed on the stream, before triggering the dependent rule.

B. DYNAMIC ANALYSIS: STOCHASTIC SIMULATION

CEP systems are often probabilistic, due to the stochastic and uncertain nature of the environments with which they interact [34]. This is why probabilistic modeling and analysis is necessary to understand the behavior of CEP applications [34]–[36].

The typical methods used to analyze quantitative properties of stochastic systems rely on analytical methods that permit computing the probability of the events satisfying given logical formulae, with tools such as PRISM [37] or UPPAAL [38]. However, analytically constructing the corresponding closed-form probabilistic model is infeasible in many cases [39].

Alternative methods use Monte Carlo simulations, executing the specifications with different sample suites of input events. These events are generated according to the probability distributions the inputs of the system are expected to follow.

Example: Let us suppose that the sensors of the control system of a motorbike are expected to send events following an exponential distribution with mean $x = 60$ seconds. Starting from some initial values for the speed (s_0) and the pressure of the two tires (p_1 and p_2), suppose every measurement follows a normal distribution whose mean is the initial value and the standard deviation is half of it. Furthermore, suppose that in average seat sensors produce a wrong measurement every 10, 000 events.

Statistical methods permit estimating the probability of the satisfaction of a given logical formula with some confidence level and error bound—or precision. For example, we would like to estimate the probability of two or more accidents happening to the same motorbike in the same day, or the probability of a tire pressure to go above a given threshold before it blows out.

The next sections describe the tools we use to perform statistical analysis on CEP systems using Maude capabilities and tools.

1) GENERATING EVENTS RANDOMLY

Using the Maude random number generator [18], we developed in [40] a library of probability distributions that can be used to generate the random values that we need here. As an example, Listing 29 shows the specification of the Uniform distribution. It uses Maude `random` and `counter` operations [18].

```
op random01 : Int -> Rat .
eq random01(SEED) =  random(SEED) / 4294967295 .
vars LBI UBI : Int . vars LB UB : Float .
op UDistr : Int Int -> Int .
eq UDistr(LBI, UBI) = UDistr(LBI, UBI, counter) .
op UDistr : Float Float -> Float .
eq UDistr(LB, UB) = UDistr(LB, UB, counter) .
op UDistr : Int Int Int -> Int .
eq UDistr(LBI, UBI, SEED) =
       LBI + random(SEED) rem (UBI - LBI) .
op UDistr : Float Float Int -> Float .
eq UDistr(LB, UB, SEED) =
       LB + float(random01(SEED)) * (UB - LB) .
```

LISTING 29. Maude specification of a Uniform distribution.

Example: The Maude rule in Listing 30, `[Motorbike]`, illustrates the use of this library in the case of the motorbike case study. It generates `Motorbike` events every X seconds, where X is a random variable that follows an exponential distribution with mean 60. This is specified in the attribute `wakeUpAt` of the history factory object that creates the events, which determines the next time this rule is triggered. Notice that the attributes of the newly created `Motorbike` event make use of normal and uniform distributions to implement the requirements described above.

This factory object also incorporates a variable, `num`, which permits setting a bound on the number of `Motorbike` events to be generated. Initializing that variable to `-1` means no bound; this is useful in case we want instead to set a time bound on the execution.

Using this approach we allow CEP system analysts to specify two different forms of uncertainty for events:

```
crl [Motorbike] :
< CO : Counter | n : N >
< C : Clock | time : NOW >
< O : MotorbikeFactory | id : M, location : S1, speed : I1,
      tirePressure1 : Y1, tirePressure2 : Z1,
      seat : B1, num : I, wakeUpAt : NOW >
< S : Stream | events : L >
=> < C : Clock | time : NOW >
   < O : MotorbikeFactory | id : M, location : S1, speed :
         I1, tirePressure1 : Y1, tirePressure2 : Z1,
         seat : B1, num : I - 1,
         wakeUpAt : NOW + expDistr(float(1 / 60)) >
   < CO : Counter | n : N + 1 >
   < S : Stream | events : insert(
      < N + 1 : Motorbike | id : M, location : S1,
        speed : I1 + normDistr(0.0, I1 / 2),
        tirePressure1 : Y1 + normDistr(0.0, Y1 / 2),
        tirePressure2 : Z1 + normDistr(0.0, Y2 / 2),
        seat : if UDistr(0.0, 1.0) <= 0.0001
               then (not B1) else B1 fi,
        ts : NOW > , L ) >
if I =/= 0 .
```

LISTING 30. Creation of random `Motorbike` events.

the uncertainty about their contents (i.e., the values of their attributes); and the uncertainty about their occurrence [41].

2) MONTE CARLO SIMULATIONS

Using a similar approach to VeStA [36] and PVeStA [42], we have used a simple simulator that we developed in [40], [43], which permits performing a number of executions of the Maude specification of a CEP system, and to collect the outcome of these runs. All individual executions are executed in parallel, which permits making use of the parallel resources available in our machine.

Example: Using our application, the command shown below performs the parallel execution of 256 processes, each one in charge of executing the Maude specifications stored in the file `accidents.maude`, which simulates the behavior of the system for one day (86400 seconds). The input `Motorbike` events for the simulation are randomly generated as shown in Listing 30.

```
$ ./runCEP.sh accidents 256
```

An associated file `accidents.events` lists the names of the (simple and complex) events of interest. The results of the executions are stored in the file `accidents.csv`. Such a file contains the basic statistics about the executions including the number of generated simple events, the number of complex events produced during the simulation, etc. The duration of the runs (in seconds) are also included, in the `Duration` row.

An example of the contents on file `accidents.csv` after the concurrent executions is shown in Listing 31.

```
Event,Min,Max,Median,Mean,Std.Dev
Motorbike,2834,4570,4320,4207.64,301.152
DriverLeftSeat,0,2,0,0.269531,0.516911
GhostRider,0,3,0,0.371094,0.648755
BlowOutTire,0,2,0,0.246094,0.498028
Crash,0,3,0,0.292969,0.621953
Accident,0,3,2,1.62305,1.42127
AccidentsReport,3,3,3,3,0
Duration,178,225,217,213.492,8.85737
```

LISTING 31. Output file after the simulation.

The overall simulation time, for the 256 processes is 225 seconds, using a machine whose operating system is Ubuntu 14.04 LTS with 64 GB of RAM memory and a processor with 24 cores of 2.67 GHz each.

3) PERFORMANCE EVALUATION

As mentioned in the introduction of this section, the scalability and acceptable performance of the analysis of any formal tool are essential to ensure its applicability. The analysis of the Maude specifications that represent a CEP system should be efficient enough to be used in practice and provide simulation and analysis results in reasonable time.

In this respect, the Maude encoding of CEP event patterns described in Section IV works from a conceptual point of view, but some optimizations are required in order to improve the execution performance when dealing with large sets of events.

First, the use of only one list of objects in the stream of events is not efficient because the complete list needs to be traversed every time a rule is evaluated to match the CEP pattern. One way to address this issue is to periodically prune the list by removing those events (both simple and complex) that will no longer be needed. This can be specified in Maude by using a class `EventPruningFactory` (Listing 32) that contains the pruned events and a rule that periodically moves the "old" events to the backup list.

```
class EventPruningFactory |
    period : Time, savedEvents : List{Object} .
subclass EventPruningFactory < TimedFactory .
rl [PruneEvents] : *** pruning of old events
< C : Clock | time : NOW >
< S : Stream | events : L >
< F : EventPruningFactory | period : T,
      savedEvents : BL, wakeUpAt : NOW >
=> < C : Clock | time : NOW >
   < S : Stream | events : removeOldEvents(L, NOW - T)>
   < F : EventPruningFactory | period : T,
         savedEvents : (addOldEvents(L, NOW - T) ; BL),
         wakeUpAt : NOW + T > .
```

LISTING 32. `EventPruningFactory` class.

Then we only need to include the factory object (see Listing 33) in the initial model, whose pruning period is defined by means of a constant, `PRUNINGPERIOD`:

```
< 'p : EventPruningFactory | savedEvents : nil,
     period : PRUNINGPERIOD, wakeUpAt : PRUNINGPERIOD >
```

LISTING 33. Object for pruning events.

The period is set to one day because there is a rule that needs to take into consideration the events that have been produced in the last 24 hours: the one that generates `AccidentsReport` complex events.

This takes us to our second optimization. We are now forced to combine events whose window is a few seconds (e.g., `Crash` or `Accident`) with others with a window of one day (`AccidentsReport`). This implies that the list of events, even after the pruning process, has to store too many elements to be efficiently explored. To address this

TABLE 1. Simulation times (in seconds) for the `accidents.maude` specifications for different durations (from 1 to 16 days) and different number of parallel executions for each case.

Num. Executions	1	2	4	8	12	16	24	32	48	64	96	128	192	256	384
1 day	14	15	15	14	14	17	22	28	42	56	84	115	166	225	318
2 days	22	23	23	24	25	29	38	50	73	97	145	204	305	401	622
4 days	45	46	48	49	51	56	74	99	144	192	285	382	583	777	1164
8 days	95	95	95	100	100	116	150	195	289	387	567	756	1164	1541	2317
16 days	204	189	192	194	203	228	301	383	564	757	1139	1515	2303	3063	4616

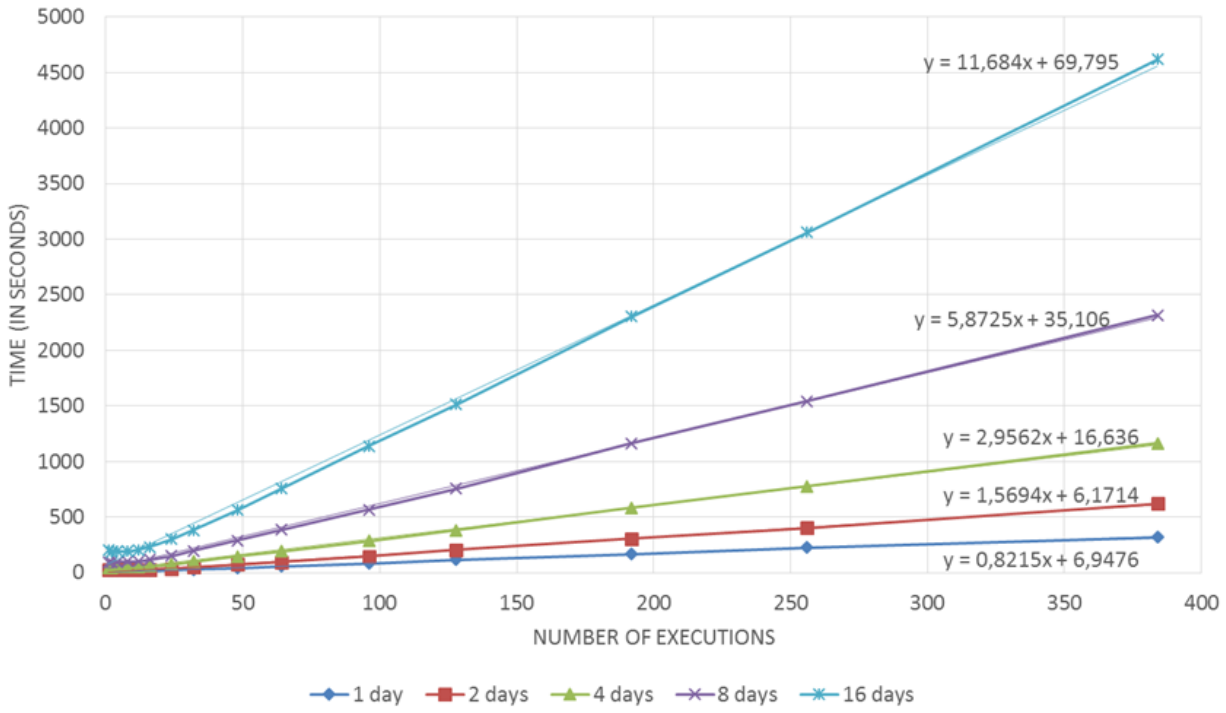

FIGURE 2. Graphical representation of the simulation times for the `accidents.maude` specification.

issue we have defined two streams of events: one for those events with a short lifetime (such as `Motorbike`) and that can be pruned shortly (every 5 seconds in this case, which is the duration of the longest window for most *fine-grained* events); and one stream for coarse-grained events such as `AccidentsReport` (see Listing 34). In this second stream we only store `Accident` events, which are the ones needed to generate `AccidentsReport` events. This list can be pruned every 24 hours. The advantage of this solution is that the `[AccidentsReport]` rule has to traverse only the list with the coarse-grained events, which contains just the `Accident` events of the last day. In this way we are able to maintain the sizes of the two lists under control, pruning them at different times.

Finally, a third improvement is achieved by not storing the list of pruned events but just the number of their occurrences. We do this by adding a new attribute (`npe`) to each `Factory` object that represents a counter with the number of produced events of this type.

Table 1 shows the execution times of different simulations of the `accidents.maude` specification, depending on the

```
rl [AccidentsReport] :
< C : Clock | time : NOW >
< CO : Counter | n : N >
< S : Stream | events : L >
< CGS : CGStream | eventscg : LO >
< F : AccidentsReportFactory |
      location : S1, wakeUpAt : NOW >
=> < C : Clock | time : NOW >
   < CO : Counter | n : N + 1 >
   < S : Stream | events : insert(
       < N + 1 : AccidentsReport | location : S1,
           num : getAccidents(LO, S1, NOW - ONEDAY, NOW),
           ts : NOW > , L ) >
   < CGS : CGStream | eventscg : LO >
   < F : AccidentsReportFactory |
         location : S1, wakeUpAt : NOW + ONEDAY > .
```

LISTING 34. Coarse-grained and fine-grained pruning of events.

number of days that every execution simulates (from 1 to 16) and the number of parallel executions of each specification. The one mentioned above corresponds to 1 day and 256 executions. Interestingly, we can see how times grow linearly with respect to the number of days and the number iterations, as Figure 2 shows (linear expressions correspond to the adjusted functions that explain such relationships with

a coefficient of determination $R^2 \geq 0.99$ in all cases). Moreover, we can also observe a linear relationship among the number of days being simulated.

We have also made tests using a machine with MacOS operating system, 2.3 GHz dual-core Intel Core i5 processors and 8 GB memory; and other with Ubuntu and twelve 2.67 GHz cores, obtaining comparable results. In particular, when the number of iterations is above the number of processors, there is a linear relation between the performance figures obtained for the three machines that exactly respects the ratio between the number of cores: the 12-cores machine is 6 times faster than the 2-cores machine, and the 24-cores machines is 2 times faster than the 12-cores one.

Maude executions are also memory-demanding, and when the RAM is used-up by the running processes and disk access is needed, the overall performance degrades. But the linear nature of the executions permits overcoming this issue rather easily. If memory limits are reached when running in parallel 256 processes, instead of launching 256 processes at once, the application can be launched 2 times with 128 parallel processes, or 4 times with 64, and then gather all the results. Since every execution uses a different seed for the random number generator, the results of the Montecarlo simulations are still valid. In this way we can overcome the problem of having a limited number of processors and/or memory. It is enough to identify the number E of maximum parallel executions that the target machine can support without requiring the use of memory swaps, and launch n/E times the application, with n the total number of executions we are interested in running (Section V-D describes how to calculate that number).

C. CHECKING PROPERTIES OF THE CEP APPLICATION

In our proposal we are interested in checking whether a given condition will ever happen (occurrence), and whether a certain condition will always hold (invariant).

The first kinds of checks correspond to safety or liveness properties, i.e., that a certain condition will never happen (e.g., no Accident is detected if the driver is still sitting on the motorbike seat) or another will end up happening (e.g., a specific pattern will indeed be triggered—which is equivalent to the fact that a particular event will happen). The second kind of property corresponds to invariants, which in turn can be on the CEP application itself (e.g., the values of the attributes of one or more events stay below a certain threshold) or on its environment, which is not necessarily modeled in the pattern set (e.g., that the values of the tires pressure never exceed 4.0 BAR). To check that an invariant holds we will simply use its negation, checking if the negation may happen in the execution of the CEP pattern set [16].

In this proposal, the properties to check are modeled in terms of CEP patterns, which specify the (complex) events that would happen if the property held. Then, we encode each property as a complex event and add it to the CEP pattern set. This also permits the use of one single language to specify both the system and the properties to analyze.

For example, to check the aforementioned property of a motorbike having two or more accidents within the same day we just define a CEP pattern that generates a complex event (`TwoAccidentsTheSameDay`) which searches for two accidents of the same motorbike in a time window of one day. Similarly, for an invariant that checks the tire pressure sensors never reach 4.0 BAR, we simple create a CEP pattern that creates a complex event `TooHighTirePressure` every time such a situation is detected. Using the statistical model checking techniques we have described above, what we will get is the probability that these events happen.

For illustration purposes, Table 2 shows for the Motorbike case study the ratios of occurrence of complex events with respect to the number of simple events, for different simulations: 1 day and 16 days; 128, 256 and 384 executions. We can see how the invariant about the valid input values produced by the motorbike tire sensors is not violated in any of the executions. However, there is a small probability that a motorbike suffers two accidents the same day, which is detected when the number of executions is large enough, as shown in the table. This bring us to the next question: which is the minimum number of executions required to provide meaningful results, with certain error bounds?

D. DETERMINING THE REQUIRED NUMBER OF EXECUTIONS

Once the properties are defined in terms of CEP patterns, we can proceed to check whether they happen or not. As mentioned above, instead of conducting exhaustive static analyses, we will perform Monte Carlo simulations as explained in Section V-B.2. The goal is to calculate the probability of the property under analysis to happen, based on the (stochastic) values assigned to the event attributes and the probabilities defined for the simple events to happen.

There are two main approaches to statistical model checking: sequential [44] and black-box [35] testing. In the former, executions are performed until the results can be considered correct within the required error bounds (as, e.g., VeStA [36] and PVeStA [42] do). In black-box testing, error bounds and a confidence level are calculated based on the results of the set of performed executions.

In our approach we use sampling methods [45] to determine the required number of executions to provide meaningful results, based on several parameters, which include the required confidence level $(1 - \alpha)$ and the precision (d) of the results. For example, one may wish to have the 95% confidence interval be less than 0.06 units wide (i.e., a required precision of $d = 0.03$).

Confidence levels are given by their corresponding critical values (Z_α-values) [46]. Besides, in this context we are concerned with the estimation of *proportions*, since we can formulate the problem as the estimation of the ratio of executions that fail the test, e.g., that find a counterexample about the property we want to check. If the ratio of positive observations is unknown, the recommended practice is to assume it to be $\hat{p} = 0.5$. This implies the maximum variance

TABLE 2. Ratios of occurrence of the events defined for the Motorbike case study (vs. number of simple Motorbike events).

	1 day - 128	1 day - 256	1 day - 384	16 days - 128	16 days - 256	16 day - 384
Accident	5.360854×10^{-5}	5.848742×10^{-5}	4.92492×10^{-5}	4.939658×10^{-5}	4.896661×10^{-5}	5.159531×10^{-5}
BlowOutTire	5.360854×10^{-5}	5.848742×10^{-5}	4.986502×10^{-5}	4.951304×10^{-5}	4.931584×10^{-5}	5.213967×10^{-5}
Crash	5.545723×10^{-5}	6.962787×10^{-5}	5.29429×10^{-5}	5.41731×10^{-5}	5.327516×10^{-5}	5.645544×10^{-5}
DriverLeftSeat	5.730592×10^{-5}	6.405752×10^{-5}	5.540544×10^{-5}	5.405663×10^{-5}	5.368268×10^{-5}	5.664984×10^{-5}
GhostRider	1.146116×10^{-4}	8.819528×10^{-5}	1.0342349×10^{-4}	9.96087×10^{-5}	9.723445×10^{-5}	1.0330723×10^{-4}
TooHighTirePressure	0	0	0	0	0	0
TwoAccidentsTheSameDay	0	0	0	0	5.822×10^{-8}	7.776×10^{-8}

TABLE 3. Confidence levels, corresponding Z_α values and required number of executions.

Confidence level	80%	85%	90%	95%	98%	99%
Z_α values	1.28	1.44	1.64	1.96	2.33	2.58
No. executions ($p = 0.5, d = 0.03$)	455	576	747	1067	1508	1849
No. executions ($p = 0.05, d = 0.03$)	86	109	142	203	287	351

of this distribution ($0.25/n$) and hence the more conservative case. But sometimes we may have some a-priori estimation of the upper bound of the probability of the error to occur.

With all this, the formula to determine the required number of executions n is: $n = Z_\alpha^2 \cdot p \cdot q/d^2$, where Z_α is the critical value for the corresponding confidence level; d is the required precision; p is the proportion of positive observations, and q is the proportion of negative observations ($q = 1 - p$). The result of the formula is always rounded to the next integer value. Table 3 shows some common confidence levels, their corresponding Z_α values [46], and the number executions calculated using the formula when the required precision is 3% ($d = 0.03$) for the cases in which we do not have information about the proportions ($p = 0.5$) and when we expect to fail only in at most 5% of the cases ($p = 0.95$).

E. TIME-BOUNDED REACHABILITY ANALYSIS

We can also take advantage of Maude's search capabilities to provide timed search commands for analyzing all possible behaviors starting from an initial configuration. In particular, the timed search obtains states which are reachable in a certain time interval from the initial state. An upper bound to the time interval to explore must be given in order to restrict the number of states of the search space. Thus, we can search for executions leading to undesirable states which we want to avoid in the system, or for situations violating any of the properties which we want to prove on the system. It is important to note that to conduct the search process random numbers need to be abstracted away, replacing them by actual values in the Maude specifications. Both kinds of analyses, search and stochastic simulations, need to be separated.

Note as well that if the system is confluent, given an initial state there will be only one possible execution path. In this case, the `search` command will explore just this behavior, looking for the occurrence of the specified situation.

For example, suppose we want to search for those executions in which certain event is detected (for instance, `TooHighTirePressure`), starting from the initial configuration `InitialModel`, and having an execution bound of 15,000 time units. The `search` command that can be used for such a search is presented in Listing 35.

```
(omod RUN is
  inc MOTORBIKE .
  eq TIMELIMIT = 15000 .
 endom)
(search InitialModel =>*
  { < S:Oid : Stream | events :
     ( L1:List`{Object`} ;
       < O:Oid : TooHighTirePressure:Cid |
           ATTS1:AttributeSet > ; L2:List`{Object`} ) >
      CONF:Configuration }
.)
```

LISTING 35. Example of Maude `search` command.

The result will be a collection of states (solutions) fulfilling the given search pattern within the specified time frame. In this example the result is the empty set, because no solution is found.

F. VALIDATION OF OUR PROPOSAL USING FURTHER CASE STUDIES

In addition to the case study presented in the paper to illustrate our approach, we have validated our proposal with different examples to check its expressiveness, fitness for purpose and applicability. Here we will mention two of them, a simple one commonly used in some CEP tutorials, and another of an existing real application of environmental monitoring currently used in Andalusia, Spain. Their complete implementations in Esper and in Maude are available from [47].

1) TEMPERATURE MONITORING CASE STUDY

The first application we used to validate our approach simulates a simplistic temperature monitoring system at a nuclear power plant [48]. Using the temperature measurements produced every second by the Plant sensors as the *simple* events, the CEP application defines and processes three types of events: `Monitor` just tells us the average temperature every 10 seconds; `Warning` warns us if we have 2 consecutive temperatures above a certain threshold (400°C); finally, `Critical` alerts us of any sudden, rising escalating temperature spike whereby we have 4 consecutive events, with the first one above 100°C, each subsequent one greater than the previous, and the last one being 1.5 times greater than the first.

The CEP application is defined by three patterns written in Esper, one per complex event. The corresponding Maude specifications have also three rules, in addition to the one that generates the simple `Temperature` events. The static analyses show in this case that the complex events are independent among themselves, being only dependent on the simple events—i.e., the pattern graph is a tree. No cycles and no race conditions are therefore detected. The simulations permit calculating the probability of the complex events. Actually, in this example the complex events correspond to *properties* that we want to check about the input stream of `Temperature` events. What we get for free with our approach is a tool that, given the probability distributions of the input temperatures, computes the probabilities of the occurrences of such properties.

2) AIR QUALITY MONITORING CASE STUDY

This is an example of real CEP application that automatically controls air quality in real time across the Andalusian region (33,694 sq. miles, 8.4 million people). It uses the existing Andalusian Regional Government's sensor network composed of 61 sensor stations spread all over the region. Each station measures every 10 minutes six air pollutants including, e.g., carbon monoxide, ozone, nitrogen dioxide and sulfur dioxide. Air quality is measured using the index proposed by the U.S. Environmental Protection Agency (EPA), which defines windows of 1, 8 or 24 hours to analyze the simple events, depending on the specific air pollutant, and expresses the results in a 6-grade scale, from *Good* to *Hazardous* [49].

The CEP application is defined in terms of 43 patterns. The information about each pollutant measurement is aggregated by one pattern, and other six patterns determine the air quality grade for this pollutant. This makes seven patterns for every pollutant, and therefore 42 patterns are used to analyze the six pollutants. A final pattern, `AirQualityLevel`, calculates the global air quality level by computing the maximum of the grades obtained for each pollutant.

The static analyses for the application did not detect cycles in the pattern dependency graph but, interestingly, detected possible race conditions and non-confluence conditions in the original patterns. This is due to the fact that the pattern dependency graph has 4 levels: the final derived pattern `AirQualityLevel` depends on derived six patterns (the ones that calculate the level of each pollutant) which in turn rely on the derived patterns that aggregate the simple sensor air measurements for each pollutant. These dependencies may cause that the `AirQualityLevel` pattern is triggered before all the dependent patterns are executed, and therefore its results may not have into account all correct values.

There are two ways to deal with these issues in the application. The first one schedules the patterns of the complex events so that they always occur *after* the patterns they depend upon. The second approach assigns priorities to the patterns. This is the safer method in case the EPL used to develop the application supports this feature. Then a different priority should be assigned to all patterns at the same level in the graph, ensuring no pattern races occur. The real application was fixed accordingly, using this approach.

Our Maude specifications were also validated by comparing their behavior and results with the ones obtained by the Esper-based application, using the real sensing data collected from December 2016 to April 2017 in Andalusia. The results confirmed that the behavior of both systems was equivalent, i.e., they produced the same complex events, in the same order.

In addition, we were also able to simulate the system with randomly generated sets of 1000, 10000 and 100000 events, to understand the effects of changes in the probability distributions of the input simple events (the air measurements for the six pollutants) into the overall air quality of the area controlled by each sensor station. This showed to be very useful too—especially because we did not need any other tool or language to run these simulations.

In order to compare the correctness, expressiveness and efficiency of our proposal with other formal approaches, we conducted some experiments using a subset of this case study (focusing on just 2 quality factors) which was also used in [50] to formalize CEP systems using Petri Nets (PN). First, we executed the system with a set of controlled input events, observing that both systems produced the same complex events. Then, we executed both systems with a randomly generated set of 1000 input events. Maude was able to generate the events and simulate the system in 8 seconds, while the PN solution took 32 minutes (using a 2.3 GHz dual-core Intel Core i5 processors machine). Furthermore, the PN encoding of the same system was far more complex than the Maude one, as we discuss in the next section.

G. AUTOMATIC MAPPING FROM ESPER EPL TO MAUDE

Writing the Maude specifications that correspond to a CEP patterns set is not a simple task, and requires specialized knowledge. To facilitate the production of the Maude patterns we have developed a Esper parser and Maude code generator with Xtext [51] that assists the system analyst to generate the Maude specifications of CEP applications written in Esper. This has also served to validate the feasibility and applicability of our proposal.

Although the current version of the tool does not cover all features of the Esper language, it now implements the mappings for the most commonly used CEP patterns, including those that we have needed to analyze the systems studied so far. For those mappings, the application is able to generate all the basic Maude infrastructure and the skeleton of the corresponding Maude rules, with the user only expected to specify how the values of the attributes of the corresponding complex event need to be computed. The rest of the Maude specifications and apparatus is automatically generated.

VI. RELATED WORK

Rabinovich *et al.* [52] analyze the behavior of CEP applications using static and dynamic techniques for finding possible termination problems, event consequences and provenance,

tracing event impact, application artifacts evaluation, and coverage. As an example, they can detect the set of patterns that a specific sequence of events triggers. Nevertheless, they do not automatically generate input events to stress a particular property, as we do in our proposal.

CAVE [16] transforms CEP patterns into basic constraints and uses constraint solvers to analyze the satisfiability of CEP programs in a very efficient manner. This work is complementary to ours. In fact, CAVE allows CEP system developers to verify application-specific properties, but it does not support simulation. However, CAVE can generate sequences of events satisfying or violating a given property, which constitute significant inputs to validate a concrete implementation.

REX [30] uses model checking for the formal analysis of CEP patterns. REX enables developers to write and verify application-specific properties. Patterns are encoded in temporal automata while properties are encoded as computational tree logic (CTL) formulas. In order to verify properties, the UPPAAL model checker [53] is used. The problem of model checking techniques for analyzing CEP applications is the rapid explosion of states. Another formalization of CEP systems that makes use of timed automata can be found in [54] and [55], which is similar to the TESLA language. TESLA [56] is a highly expressive language that provides timers, temporal and content filters, negations, aggregations, as well as customizable policies for event selection and consumption.

Hinze and Voisard [57] propose a formal approach that defines a parametrized event algebra (EVA) in order to support adaptable event composition, including temporal restriction. In the context of Active DBMSs [58] there is a group of works related to the analysis of rule-based reactive systems, including confluence [59], [60], termination [61], and correctness [62], [63]. Our work is similar to those, applied to the context of CEP systems.

There are other works modeling CEP systems using Petri Nets. An an example, Ahmad *et al.* [64] model CEP applications using Timed Net Condition Event Systems (TNCES), a formalism based on Timed Petri Nets that provides a modular modeling formalism for discrete event dynamic systems. According to the authors, TNCES allows us to prevent machine breakdown and deadlocks involved in event-driven systems. Moreover, this approach makes the validation of events possible, checking if the system satisfy certain properties. 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. Our Maude notation for EPL patterns is more concise and manageable.

Weidlich *et al.* [65] present an analysis and simulation tool that maps Event Processing Networks (EPNs) into Colored Petri Nets with Priorities and Time (PTCPNs). They enable the verification of some internal properties, such as the presence of unused transitions in the EPN graph. In turn, Macià *et al.* [11] have proposed the use of Prioritized Colored Petri Nets (PCPNs) to formalize event patterns in a compositional way and to perform stepwise simulation and debugging of the CEP programs. Unlike [65], Macià *et al.* annotate the event timestamp as a field in the place color set, instead of using the timed capabilities of PTCPNs (timed color sets). This is because the use of timed tokens entangles the translations unnecessarily—timed tokens only can be used when they are available—and the composition of patterns requires more than one processing of the input event sequence. The problem, again, is the different abstraction level of PNs and CEP patterns, which produces very large encodings in PNs, rather cumbersome to understand and analyze.

To detect hazards in the aerospace application domain, Carle *et al.* [66] also use untimed colored Petri nets. They define *chronicles*, a situation description language that enables the detection of simple events, disjunction conjunction, sequence and absence of chronicle operators. However, timed events and data windows are not considered in this particular language, being a limitation for modeling CEP-based systems.

As mentioned in Section V-F, the analysis of event patterns formalized in Maude is more efficient than in Petri Nets. This is consistent with previous comparison studies, such as [67], which also support our conclusion, when they use Maude's rewriting semantics as a logical and operational representation of PN models for both formal verification and execution.

Our work is also similar to those proposals that enable static analysis of probabilistic CEP applications, which use analysis tools, such as PRISM, to predict their behavior. As part of these works, Debbi [34] uses probabilistic model checking and PRISM to verify probabilistic CEP systems and for estimating the probability of the occurrence of relevant events. One advantage of a simulation approach like ours is that it does not impose any constraint on the distribution models of the input events and on the values of their attributes.

There is also a plethora of works dealing with the probabilistic nature of CEP applications. As an example, PROXIMA project [68] provides cost-effective software timing analysis using probabilistic analysis for multi/many-core critical real-time embedded systems. Although we do not currently perform analyses on these types of systems, Bijo *et al.* [69] have demonstrated that Maude can be effectively used for formalizing programs executing on cache coherent multicore architectures.

There is recent work that also proposes Maude to formalize CEP events [31] although their mapping is different to ours (they do not use Full Maude) and they just propose LTL model checking to analyze the CEP patterns. Being an initial proposal the authors do not show any performance or scalability figures.

There are several tools to perform simulation of CEP applications, e.g., AMIT [70], BiCEP [71], FINCoS [72], CEPBen [73] and CEPSim [74], as we do with our proposal. However, we also provide further analyses within the same framework. In general, comprehensive analysis of CEP

application for correctness, integrity and predictable behavior is still an active area of research.

VII. CONCLUSIONS AND FUTURE WORK

This paper has presented a translation from CEP patterns to Maude specifications for the static and dynamic analyses of CEP applications. A set of initial validation experiments have shown appropriate results about the usability, effectiveness and performance of our approach. The analyses also permit detecting potential errors in the CEP pattern set, as it happened in one real application we used to validate our approach. Additionally, the analyses allow us to compute the probabilities of certain properties of the CEP application to occur: safety and liveness properties, as well as invariants.

Our work can be continued in various directions. First, we support the basic features of CEP languages, but there are some concepts and mechanisms not currently implemented. For example, we now support batch and sliding windows, but not other more complex window types such as *frames* [26]. We plan to start adding these kinds of features to our mapping. Likewise, the tool that automatically generates the Maude specifications from the CEP patterns expressed in Esper EPL now covers the set of features we have currently required. However, it may need to be extended to cover more CEP constructs and mechanisms as we conduct more validation experiments, widening the number and the scope of applications that we can cover. Currently, we do not deal with *distributed* stream processing (DSP), either. Providing semantics for DSP is a difficult issue because of all the complexity introduced by the distribution aspects. This is something we would like to explore next, using our current proposal as a basis. Finally, in this paper we have focused on some properties of CEP applications. We do not deal, however, with other important aspects such as performance estimation (response time, throughput, or memory requirements, for example); robustness, reliance, and so forth. These can also be interesting extensions to our work.

ACKNOWLEDGMENT

The authors would also like to thank Francisco Durán for his help with the Maude language and his insightful comments and suggestions.

REFERENCES

[1] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, 2012.

[2] O. Etzion and P. Niblett, *Event Processing in Action*. Shelter Island, NY, USA: Manning Publications, 2010.

[3] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Reading, MA, USA: Addison-Wesley, 2002.

[4] D. C. Luckham, *Event Processing for Business: Organizing the Real-Time Enterprise*. Hoboken, NJ, USA: Wiley, 2012.

[5] S. Greengard, *The Internet of Things*. Cambridge, MA, USA: MIT Press, 2015.

[6] A. Garcia-de-Prado, G. Ortiz, and J. Boubeta-Puig, "COLLECT: COLLaborativE ConText-aware service oriented architecture for intelligent decision-making in the Internet of Things," *Expert Syst. Appl.*, vol. 85, pp. 231–248, Nov. 2017.

[7] C. Zang and Y. Fan, "Complex event processing in enterprise information systems based on RFID," *J. Enterprise Inf. Syst.*, vol. 1, no. 1, pp. 3–23, 2007.

[8] W. Yao, C. Chu, and Z. Li, "Leveraging complex event processing for smart hospitals using RFID," *J. Netw. Comput. Appl.*, vol. 34, no. 3, pp. 799–810, 2011.

[9] F. Wang and P. Liu, "Temporal management of RFID data," in *Proc. VLDB Endowment*, 2005, pp. 1128–1139.

[10] K. Broda, K. Clark, R. Miller, and A. Russo, "SAGE: A logical agent-based environment monitoring and control system," in *Ambient Intelligence* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2009, pp. 112–117.

[11] H. Macià, V. Valero, G. Díaz, J. Boubeta-Puig, and G. Ortiz, "Complex event processing modeling by prioritized colored Petri nets," *IEEE Access*, vol. 4, pp. 7425–7439, 2016.

[12] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White, "Towards expressive publish/subscribe systems," in *Advances in Database Technology—EDBT* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2006, pp. 627–644.

[13] J. Boubeta-Puig, G. Ortiz, and I. Medina-Bulo, "MEdit4CEP: A model-driven solution for real-time decision making in SOA 2.0," *Knowl.-Based Syst.*, vol. 89, pp. 97–112, Nov. 2015.

[14] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Shelter Island, NY, USA: Manning Publications, 2015.

[15] S. Shi, D. Jin, and T. Goh, "Real-time public mood tracking of Chinese microblog streams with complex event processing," *IEEE Access*, vol. 5, pp. 421–431, 2017.

[16] G. Cugola, A. Margara, M. Pezzè, and M. Pradella, "Efficient analysis of event processing applications," in *Proc. DEBS*, 2015, pp. 10–21.

[17] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theor. Comput. Sci.*, vol. 96, no. 1, pp. 73–155, 1992.

[18] M. Clavel *et al.*, *All About Maude—A High-Performance Logical Framework* (Lecture Notes in Computer Science), vol. 4350. Heidelberg, Germany: Springer, 2007.

[19] Event Processing Technical Society. (2011). *Event Processing Glossary, Version 2.0*. [Online]. Available: http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Proc%essing_Glossary_v2.pdf

[20] EsperTech. *Esper—Complex Event Processing*. Accessed: Mar. 24, 2018. [Online]. Available: http://www.espertech.com/esper/

[21] *Portable Operating System Interface (POSIX(R))*, IEEE Standard 1003.1-2008, 2016, no. 7, sec. 4.16.

[22] J. Meseguer and G. Rosu, "The rewriting logic semantics project: A progress report," *Inf. Comput.*, vol. 231, pp. 38–69, Oct. 2013.

[23] J. Meseguer, "Membership algebra as a logical framework for equational specification," in *Recent Trends in Algebraic Development Techniques* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 1998, pp. 18–61.

[24] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer, "Specification and proof in membership equational logic," *Theor. Comput. Sci.*, vol. 236, no. 1, pp. 35–132, 2000.

[25] P. C. Ölveczky and J. Meseguer, "Semantics and pragmatics of real-time Maude," *Higher-Order Symbolic Comput.*, vol. 20, nos. 1–2, pp. 161–196, 2007.

[26] M. Grossniklaus, D. Maier, J. Miller, S. Moorthy, and K. Tufte, "Frames: Data-driven windows," in *Proc. DEBS*, 2016, pp. 13–24.

[27] C. Zhang, X. Yao, and J. Zhang, "Abnormal condition monitoring of workpieces based on RFID for wisdom manufacturing workshops," *Sensors*, vol. 15, no. 12, pp. 30165–30186, 2015.

[28] S. R. Jeffery, M. Garofalakis, and M. J. Franklin, "Adaptive cleaning for RFID data streams," in *Proc. VLDB Endowment*, 2006, pp. 163–174.

[29] L. Li, T. Liu, X. Rong, J. Chen, and X. Xu, "An improved RFID data cleaning algorithm based on sliding window," in *Internet of Things* (Communications in Computer and Information Science), vol. 312. Berlin, Germany: Springer, 2012, pp. 262–268.

[30] A. Ericsson, P. Pettersson, M. Berndtsson, and M. Seiriö, "Seamless formal verification of complex event processing applications," in *Proc. DEBS*, 2007, pp. 50–61.

[31] A. Riesco, M. Palomino, and N. Marti-Oliet, "Towards a formal framework for analyzing stream processing systems in Maude," in *Proc. PROLE*, 2017, p. 1.

[32] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo, "Deduction, strategies, and rewriting," *Electron. Notes Theor. Comput. Sci.*, vol. 174, no. 11, pp. 3–25, 2007.

[33] M. Roldán, F. Durán, and A. Vallecillo, "Invariant-driven specifications in Maude," *Sci. Comput. Program.*, vol. 74, no. 10, pp. 812–835, 2009.
[34] H. Debbi, "Modeling and formal analysis of probabilistic complex event processing (CEP) applications," in *Proc. ECMFA*, 2017, pp. 248–263.
[35] G. Agha, J. Meseguer, and K. Sen, "PMaude: Rewrite-based specification language for probabilistic object systems," *Electron. Notes Theor. Comput. Sci.*, vol. 153, no. 2, pp. 213–239, 2006.
[36] K. Sen, M. Viswanathan, and G. Agha, "On statistical model checking of stochastic systems," in *Computer Aided Verification*. Berlin, Germany: Springer, 2005, pp. 266–280.
[37] M. Z. Kwiatkowska, G. Norman, and D. Parker, "PRISM: Probabilistic model checking for performance and reliability analysis," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 36, no. 4, pp. 40–45, 2009.
[38] P. E. Bulychev *et al.*, "UPPAAL-SMC: Statistical model checking for priced timed automata," in *Proc. QAPL*, vol. 85. 2012, pp. 1–16.
[39] F. Durán, A. Moreno-Delgado, and J. M. Álvarez-Palomo, "Statistical model checking of e-motions domain-specific modeling languages," in *Fundamental Approaches to Software Engineering* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2016, pp. 305–322.
[40] J. Troya and A. Vallecillo, "Specification and simulation of queuing network models using domain-specific languages," *Comput. Standards, Interfaces*, vol. 36, no. 5, pp. 863–879, 2014.
[41] G. Cugola, A. Margara, M. Matteucci, and G. Tamburrelli, "Introducing uncertainty in complex event processing: Model, implementation, and validation," *Computing*, vol. 97, no. 2, pp. 103–144, 2015.
[42] M. AlTurki and J. Meseguer, "PVeStA: A parallel statistical model checking and quantitative analysis tool," in *Algebra and Coalgebra in Computer Science* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2011, pp. 386–392.
[43] J. Troya, A. Vallecillo, and F. Durán, and S. Zschaler, "Model-driven performance analysis of rule-based domain specific visual models," *Inf. Softw. Technol.*, vol. 55, no. 1, pp. 88–110, 2013.
[44] H. L. S. Younes and R. G. Simmons, "Probabilistic verification of discrete event systems using acceptance sampling," in *Computer Aided Verification* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2002, pp. 223–235.
[45] S. K. Thompson, *Sampling*. Hoboken, NJ, USA: Wiley, 2012.
[46] R. M. Groves, F. J. Fowler, Jr., M. P. Couper, J. M. Lepkowski, E. Singer, and R. Tourangeau, *Survey Methodology*, 2nd ed. Hoboken, NJ, USA: Wiley, 2009.
[47] J. Boubeta-Puig, L. Burgueño, and A. Vallecillo. *Formalization of Complex Event Processing Systems*. Accessed: Mar. 24, 2018. [Online]. Available: http://atenea.lcc.uma.es/projects/FormalizingCEP.html
[48] A. Milne. *Complex Event Processing Made Easy (Using Esper)*. Accessed: Mar. 24, 2018. [Online]. Available: http://www.adrianmilne.com/complex-event-processing-made-easy/
[49] D. Mintz, "Technical assistance document for the reporting of daily air quality—The air quality index (AQI)," U.S. Environ. Protection Agency, Washington, DC, USA, Tech. Rep. EPA-454/B-16-002, 2016. [Online]. Available: http://www.epa.gov/airnow/aqi-technical-assistance-document-may2016.pdf
[50] J. Boubeta-Puig, G. Díaz, H. Macià, V. Valero, and G. Ortiz, "MEdit4CEP-CPN: An approach for complex event processing modeling by prioritized colored Petri nets," *Inf. Syst.*, 2017, doi: 10.1016/j.is.2017.11.005.
[51] *Xtext*. Accessed: Apr. 2018. [Online]. Available: http://www.eclipse.org/Xtext/
[52] E. Rabinovich, O. Etzion, S. Ruah, and S. Archushin, "Analyzing the behavior of event processing applications," in *Proc. DEBS*, 2010, pp. 223–234.
[53] G. Behrmann, A. David, and K. G. Larsen, *A Tutorial on UPPAAL* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2004, pp. 200–236.
[54] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *Proc. SIGMOD*, New York, NY, USA, 2008, pp. 147–160.
[55] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman, "On supporting Kleene closure over event streams," in *Proc. ICDE*, Apr. 2008, pp. 1391–1393.
[56] G. Cugola and A. Margara, "TESLA: A formally defined event specification language," in *Proc. DEBS*, New York, NY, USA, 2010, pp. 50–61.
[57] A. Hinze and A. Voisard, "EVA: An event algebra supporting complex event specification," *Inf. Syst.*, vol. 48, pp. 1–25, Mar. 2015.
[58] J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules for Advanced Database Processing*. San Mateo, CA, USA: Morgan Kaufmann, 1996.
[59] S. Comai and L. Tanca, "Termination and confluence by rule prioritization," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 2, pp. 257–270, Mar. 2003.
[60] A. Aiken, J. M. Hellerstein, and J. Widom, "Static analysis techniques for predicting the behavior of active database rules," *ACM Trans. Database Syst.*, vol. 20, no. 1, pp. 3–41, 1995.
[61] E. Baralis, S. Ceri, and S. Paraboschi, "Compile-time and runtime analysis of active behaviors," *IEEE Trans. Knowl. Data Eng.*, vol. 10, no. 3, pp. 353–370, May 1998.
[62] E. Falkenroth and A. Törne, "How to construct predictable rule sets," *IFAC Proc. Vol.*, vol. 32, no. 1, pp. 81–88, 1999.
[63] S.-Y. Lee and T.-W. Ling, *Verify Updating Trigger Correctness*. Berlin, Germany: Springer, 1999, pp. 382–391.
[64] W. Ahmad, A. Lobov, and J. L. M. Lastra, "Formal modelling of complex event processing: A generic algorithm and its application to a manufacturing line," in *Proc. INDIN*, 2012, pp. 380–385.
[65] 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* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2013, pp. 270–290.
[66] P. Carle, C. Choppy, R. Kervarc, and A. Piel, "A formal coloured Petri net model for hazard detection in large event flows," in *Proc. APSEC*, 2013, pp. 323–330.
[67] M.-O. Stehr, J. Meseguer, P. C. Ölveczky, "Rewriting logic as a unifying framework for Petri nets," in *Unifying Petri Nets* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2001, pp. 250–303.
[68] European Project 611085. (2017). *Probabilistic Real-Time Control of Mixed-Criticality Multicore and Manycore Systems (PROXIMA)*. [Online]. Available: http://www.proxima-project.eu/
[69] S. Bijo, E. B. Johnsen, K. I. Pun, and S. L. T. Tarifa, "A Maude framework for cache coherent multicore architectures," in *Rewriting Logic and Its Applications* (Lecture Notes in Computer Science). Cham, Switzerland: Springer, Apr. 2016, pp. 47–63.
[70] A. Adi and O. Etzion, "Amit—The situation manager," *VLDB J.*, vol. 13, no. 2, pp. 177–203, 2004.
[71] P. Bizarro, "BiCEP—Benchmarking complex event processing systems," in *Proc. Daghstuhl Seminar Event Process.*, vol. 07191. Dagstuhl, Germany, 2007. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2007/1143/
[72] M. R. Mendes, P. Bizarro, and P. Marques, "FINCoS: Benchmark tools for event processing systems," in *Proc. ICPE*, 2013, pp. 431–432.
[73] C. Li and R. Berry, "CEPBen: A benchmark for complex event processing systems," in *Proc. TPCTC*, 2013, pp. 125–142.
[74] W. A. Higashino, M. A. M. Capretz, and L. F. Bittencourt, "CEPSim: Modelling and simulation of complex event processing systems in cloud environments," *Future Generat. Comput. Syst.*, vol. 65, pp. 122–139, 2016.

LOLI BURGUEÑO received the B.Sc. degree in computer science and engineering from the Universidad de Málaga, Spain, in 2011, and received the master's degree in software engineering and artificial intelligence in 2012 and the Ph.D. degree (Hons.) in 2016. She is currently a Post-Doctoral Researcher and a Lecturer with the Universidad de Málaga. Her research interests include model-driven engineering; in concrete, she is involved in testing model transformations, the parallelization of the execution of model transformations, and the modeling of uncertainty in software models for its use in the Industry 4.0. Further information can be found at http://www.lcc.uma.es/~loli.

JUAN BOUBETA-PUIG received the degree in computer systems management and the B.Sc. and Ph.D. degrees in computer science from the University of Cádiz (UCA), Spain, in 2007, 2010, and 2014, respectively. Since 2009, he has been an Assistant Professor with the Department of Computer Science and Engineering, UCA. His research interests include the integration of complex event processing in event-driven service-oriented architectures, Internet of Things, and model-driven development of advanced user interfaces. He received the Extraordinary Ph.D. Award from UCA and the Best Ph.D. Thesis Award from the Spanish Society of Software Engineering and Software Development Technologies.

ANTONIO VALLECILLO is currently a Full Professor of computer science with the Universidad de Málaga. From 1986 to 1995, he was with the computer industry, working for Fujitsu and for ICL. In 1996, he joined the Universidad de Málaga, where he currently conducts research on software systems modeling and analysis. His research interests include model-based software engineering, open distributed processing, and software quality. He is involved in several standardization activities within AENOR, ISO, ITU-T, and OMG, and he is the Spanish Representative with IFIP TC2 and ISO SC7. He is on the Editorial Board of *Software and System Modeling* and the *Journal of Orthopaedic Trauma*journals. Further information about his publications, research projects, and activities can be found at http://www.lcc.uma.es/ av