IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 20, 2016, accepted October 22, 2016, date of publication October 26, 2016, date of current version November 18, 2016.

Digital Object Identifier 10.1109/ACCESS.2016.2621718

Complex Event Processing Modeling by Prioritized Colored Petri Nets

HERMENEGILDA MACIÀ[1], VALENTÍN VALERO[1], GREGORIO DÍAZ[1], JUAN BOUBETA-PUIG[2], AND GUADALUPE ORTIZ[2]

[1]School of Computer Science, University of Castilla-La Mancha 02071, Albacete, Spain
[2]Department of Computer Science and Engineering, University of Cádiz, 11519 Puerto Real, Cádiz, Spain

Corresponding author: G. Díaz (gregorio.diazg@uclm.es)

This work was supported in part by the Spanish Ministry of Science and Innovation and the European Union FEDER Funds with the Project DArDOS entitled Formal development and analysis of complex systems in distributed contexts: foundations, tools and applications under Grant TIN2015-65845-C3, subprojects 2-R and 3-R, and the Research Network on Services Science and Engineering under Grant TIN2014-53986-REDT, and in part by the University of Cádiz under Project PR2016-032.

ABSTRACT Complex event processing (CEP) is a technology that allows us to process and correlate large volumes of data by using event patterns, aiming at promptly detecting specific situations that could require special treatment. The event types and event patterns for a particular application domain are implemented by using an event processing language (EPL). Although some current model-driven tools allow end users to easily define these patterns, which are then transformed automatically into a particular EPL, the generated code is syntactically but not semantically validated. To deal with this problem, a prioritized colored Petri net (PCPN) model for CEP is proposed and conducted in this paper. This well-known graphical formalism together with CPNTools makes possible the modeling, simulation, analysis, and semantic validation of complex event-based systems. To illustrate this approach, a case study is presented, as well as a discussion on the benefits from using PCPN for modeling CEP-based systems.

INDEX TERMS Formal modeling, Petri nets, CEP, EPL, services for big data, data mining.

I. INTRODUCTION

Complex Event Processing (CEP) [1], [2] is a cutting-edge technology that allows us to process and analyze large volumes of data in the form of events with the aim of detecting relevant or critical situations for a particular domain in real time. For that purpose, the conditions describing situations to be detected must be specified by using the so-called event patterns. These patterns are implemented using specific languages developed for this purpose, known as event processing languages (EPLs) [3], and then are added into a CEP engine.

Even though a diversity of domains can currently benefit from CEP technology, the main handicap for subject matter experts is the need to define the event patterns for a particular domain in the EPL syntax provided by the CEP engine to be used in the system in question.

To solve this problem, we already proposed MEdit4CEP [4], a model-driven solution for real-time decision making in Event-Driven Service-Oriented Architecture (SOA 2.0) [5]. This solution allows domain experts to easily define the event patterns by using a graphical modeling editor [6], hiding all implementation details from them. These modeled patterns are then transformed automatically into a particular EPL, making use of Model-Driven Development (MDD) techniques [7]. Thus, we have the following consequent advantages: EPL technical aspects are hidden from end users and productivity is improved since models are easier to maintain. Furthermore, the automatic generated code will be *syntactic error*-free.

Nevertheless, this approach does not check the occurrence of *semantic errors* in the event pattern definition done by end users. It would be quite convenient that the designed event pattern models were formally verified before being implemented and deployed into the CEP engine. To address this issue, in this paper we propose to extend our event pattern models validation with a Prioritized Colored Petri Net model (PCPN) [8] and we use CPNTools [9] to support the semantic analysis of such event patterns, as illustrated in Figure 1. More specifically, the phases to be followed for defining event patterns with our approach including PCPN are detailed below:

1) *Event Pattern Model Definition:* the domain expert is responsible for graphically defining the event patterns

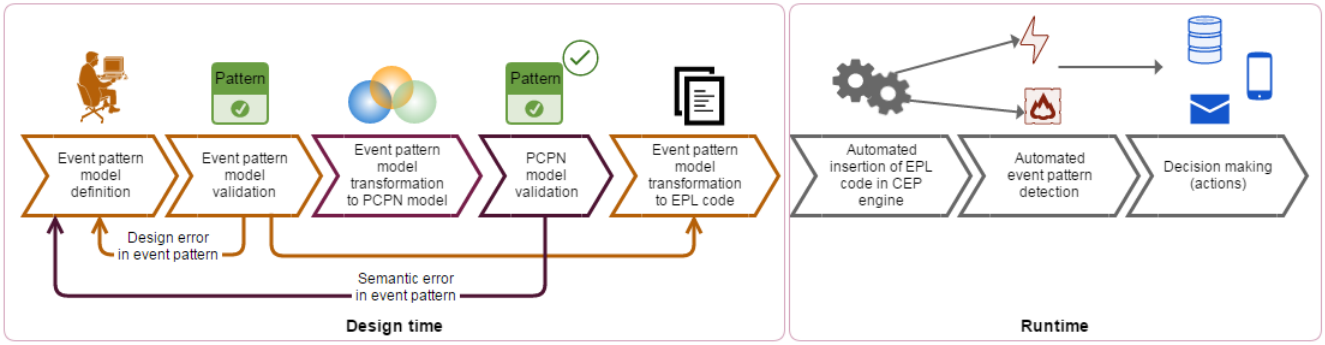


FIGURE 1. A model-driven approach for CEP modeling by PCPN.

to be detected in a specific application domain, such as health care, home automation and network security.

2) *Event Pattern Model Validation:* once an event pattern is modeled, the editor will syntactically validate it, showing the errors that should be fixed before going on. From this phase we can accomplish a semantic validation by PCPNs (phases 3 and 4) or we can proceed with phase 5 in order to transform the model into EPL code.
3) *Event Pattern Model Transformation to PCPN Model:* the event pattern model will be transformed into a PCPN model. This is currently done manually, but we expect to integrate the automatic transformation with MEdit4CEP in the near future.
4) *PCPN Model Validation:* the PCPN obtained from the previous phase can be semantically validated. We can feed the model with an arbitrary number of initial markings (stream of events), so as to check if the model is semantically correct. In the case that an error is discovered we will return to phase 1.
5) *Event Pattern Model Transformation to EPL Code:* the event pattern model will be automatically transformed into EPL code. This code will depend on the specific EPL provided by the chosen CEP engine.
6) *Automated Insertion of EPL Code in CEP Engine:* the EPL code of the modeled event pattern will be automatically inserted into the CEP engine at runtime should it be necessary.
7) *Automated Event Pattern Detection:* the engine will be able to detect the critical or relevant situations described by the deployed EPL event pattern in real time.
8) *Decision Making (Actions):* upon detecting the new situations of interest, their associated actions will be carried out at runtime.

The main contribution of this paper is the one described in phase 3 of this methodology: the procedure to be followed to convert the event pattern model to PCPNs with accuracy, starting from a particular set of patterns modeled with the MEdit4CEP tool. Once we have the PCPN capturing the EPL behavior we can accomplish the model validation phase (phase 4), in which the CPNTools framework is used in order to check if the event patterns are correct. Model validation looks for design errors in the event patterns, so the output produced by the execution of the obtained PCPN is analyzed by using some tests, which are included after the PCPN model has been obtained. How these tests are included depends on the specific application domain area, but the final intention is that they will be a PCPN extension of the PCPN obtained by the transformation. In this paper we focus our attention in the procedure required to transform the event patterns to PCPNs in order to facilitate the future automatization of the transformation and its integration with MEdit4CEP, so validation is only done at the level of the CEP engine, checking that the PCPN is correct, i.e. it generates the same output (complex events) as the CEP engine. With the PCPNs thus obtained we will have the ability to make further analysis of the defined event patterns, either quantitative or qualitative, once this transformation has been implemented in the MEdit4CEP tool.

The rest of the paper is organized as follows. In Section II, we describe the required background to facilitate the understanding of this paper. The PCPN models for a set of relevant event patterns elements are presented in Section III, and a case study illustrating the applicability of this approach is presented in Section IV. Some relevant related works are presented in Section V and finally, Section VI presents some conclusions and the lines of future work.

II. BACKGROUND

In this section we explain the background for the CEP technology together with the MEdit4CEP tool, and the PCPN formalism that we use for modeling complex event based systems.

A. COMPLEX EVENT PROCESSING

CEP is an emerging technology that allows us to analyze and correlate enormous amounts of data in the form of events with the purpose of detecting relevant or critical situations (complex events) in real time. A *situation* is an event occurrence or an event sequence that requires an immediate

reaction [10]. Events can be classified into two main categories: a *simple event* is indivisible and happens at a point in time, a *complex event* contains more semantic meaning, which summarizes a set of other events [11]. Events can be derived from other events by applying or matching *event patterns*, i.e. templates where the conditions describing situations to be detected are specified. A *CEP engine* is the software used to match these patterns over continuous and heterogeneous event streams, and to raise real-time alerts after detecting them.

These event patterns are implemented by using EPLs. It is noteworthy that we have chosen Esper EPL [12] as EPL in this work, since this rich high level processing language is more complete than others are, providing more temporal and pattern operators for defining the situations of interest. In addition, its open-source CEP engine is very efficient: it can process over 500,000 events/s [12]. For the sake of brevity, we refer to the particular language Esper EPL simply as EPL through the rest of the paper.

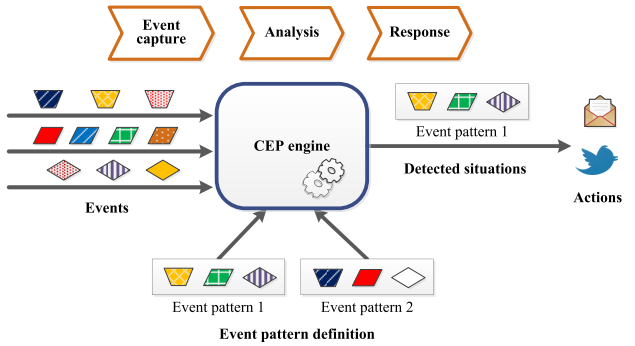


FIGURE 2. Complex event processing stages.

CEP is performed in 3 stages, as depicted in Figure 2:

1) Event capture–it receives events to be analyzed by CEP technology,
2) Analysis–based on the event patterns previously defined in the CEP engine, the latest will process and correlate the information in the form of events in order to detect critical or relevant situations in real time,
3) Response–after detecting a particular situation, it will be notified to the system, software or device in question.

The main advantage of using this technology is that such relevant or critical situations can be identified and reported in real time; thus reducing latency in decision making, unlike the methods used in traditional software for event analysis. Moreover, CEP presents other benefits [13]: decision quality improvement, faster and (semi-)automatic reply, information overload prevention and human workload reduction.

B. MEdit4CEP

MEdit4CEP [4] is a model-driven solution for real-time decision making in SOA 2.0. Its main aim is to make easier domain experts' tasks of defining both event patterns –situations of interest to be detected– and alerts for real time notification, hiding all implementation details from them.

More specifically, this solution is composed of a model-driven approach for CEP in SOA 2.0 together with both a graphical modeling editor for CEP domain definition and a graphical modeling editor for event pattern and action definition as well as code generation.

The main purpose of this model-driven approach is the definition of high-level models, which are approachable and understandable to any user. These models are created by using ModeL4CEP [14] —graphical Domain-Specific Modeling Languages (DSMLs) for CEP domains and event patterns— whose definition consists of three distinct parts: (1) the abstract syntax that consists of both a meta-model –model describing language concepts and relationships between them– and validation rules to check whether a model is well formed –the model conforms to its meta-model—, (2) the concrete syntax or DSML notation –the set of useful graphical symbols for drawing model diagrams—, and (3) model-to-code transformations for generating automatically the code that can be executed in CEP engines and Enterprise Service Buses (ESBs). A detailed explanation about these parts can be found at [14].

Table 1 summarizes all the language concepts supported by MEdit4CEP to define event patterns in a user-friendly and graphical way.

The key feature of the event pattern editor is its ability to reconfigure itself for different CEP domains, modeled by domain experts. The fact that the editor can reconfigure the tool palette —Simple Events and Complex Events categories, see Table 1— dynamically from different CEP domain models allows users to enjoy a graphical interface adapted to the specific context required.

This is a novel solution for bringing CEP technology closer to any user, positively impacting on the decision making process.

C. PRIORITIZED COLORED PETRI NETS

We use prioritized colored Petri nets, which are a prioritized extension of colored Petri nets [8], [15], [16], the well-known model supported by CPN Tools [9], developed by the CPN group at the University of Aarhus, Denmark.

A Petri Net (PN) is a directed graph, which consists of places (circles), transitions (rectangles) and arcs connecting places and transitions and viceversa. In colored PN (CPN) places have an associated *color set* (a data type), which specifies the set of allowed token colors at this place. Each token then has an attached data value, a *color*, which belongs to the corresponding place *color set*.

Arcs can have inscriptions (*arc expressions*), constructed using variables, constants, operators and functions. To evaluate an arc expression we need to bind the variables, which consists of assigning a value to the variables that appear in the arc inscription. These values are then used to select the token colors that must be removed or added when firing the corresponding transition.

TABLE 1. MEdit4CEP palette tools.

Category	Name	Description
	Link	It defines the graphical representation of one or more relationships between operands and operators.
	Value	It defines a *Boolean*, *Integer*, *Long*, *Double*, *Float* or *String* value.
Simple Events	Event	It describes an event type for a concrete CEP domain.
	EventProperty	It describes a feature of an event. Nested properties are supported.
Complex Events	ComplexEvent	It describes the complex event type to be created upon pattern detection.
	ComplexEventProperty	It describes a feature of a complex event. Nested properties are supported.
Pattern Timers	TimerInterval	It waits for the specified time period (*years*, *months*, *weeks*, *days*, *hours*, *minutes*, *seconds* and *milliseconds*) before turning to true.
	TimerSchedule	It turns into true at a defined time (*dayOfWeek*, *dayOfMonth*, *month*, *hour*, *minute*, *second*).
	WithinTimer	It is permanently evaluated to false if the contained pattern expression does not turn to true during the specified time period (*years*, *months*, *weeks*, *days*, *hours*, *minutes*, *seconds* and *milliseconds*).
Pattern Operators	Every	It selects every event belonging to the specified type.
	EveryDistinct	It is similar to Every, but eliminates duplicated results according to a given *distinct-value* expression.
	FollowedBy	It determines a pattern expression that must be followed by another.
	Range	It specifies the minimum (*lowEndpoint*) and maximum (*highEndpoint*) number of times a pattern expression must occur.
	Repeat	It defines how many times (*count*) a pattern expression must occur.
	Until	It checks a pattern expression until the condition (another pattern expression) is evaluated to true.
	While	It checks a pattern expression while the condition (another pattern expression) is evaluated to true.
Logical Operators	And	It returns a true value only if all operands are true.
	Or	It returns a true value if at least one operand is true.
	Not	It returns a true value if the operand is false, and a false value if the operand is true.
Comparison Operators	Equal	It returns a true value if $operand1 = operand2$.
	GreaterEqual	It returns a true value if $operand1 \geq operand2$.
	GreaterThan	It returns a true value if $operand1 > operand2$.
	LessEqual	It returns a true value if $operand1 \leq operand2$.
	LessThan	It returns a true value if $operand1 < operand2$.
	NotEqual	It returns a true value if $operand1 \neq operand2$.
Arithmetic Operators	Addition	It adds two numeric values.
	Division	It divides one numeric value by another.
	Modulus	It returns the remainder of dividing one numeric value by another.
	Multiplication	It multiplies two numeric values.
	Subtraction	It subtracts one numeric value from another.
Aggregation Operators	Avg	It returns the average of the values in an expression.
	Count	It returns the number of values in an expression.
	Max	It returns the highest value in an expression.
	Min	It returns the lowest value in an expression.
	Sum	It adds the values in an expression.
Data Windows	BatchingEventInterval	Tumbling window up to the specified number of events (*size*).
	BatchingTimeInterval	Tumbling window that batches events and releases them every specified time period (*years*, *months*, *weeks*, *days*, *hours*, *minutes*, *seconds*, *milliseconds*).
	SlidingEventInterval	Sliding window by the specified number of events (*size*).
	SlidingTimeInterval	Sliding window by the specified time period (*years*, *months*, *weeks*, *days*, *hours*, *minutes*, *seconds*, *milliseconds*).
Actions	Email	It indicates an email must be sent with the complex event created when detecting its corresponding event pattern.
	Twitter	It indicates a message must be sent to a Twitter account with the complex event created when detecting its corresponding event pattern.

Transitions can also have guards and priorities. Guards are predicates constructed by using the variables, constants, operators and functions of the model, and they must evaluate to true with the selected binding for the transition to be *fireable*. Furthermore, priorities can be used to establish an order of firing. Specifically, we use the following priorities:

P_HIGH, P_NORMAL, P_LOW and P_LOW2, following this decreasing ordering of priority.

Definition 1 (Multisets):

- Multisets are defined as functions $C : X \to \mathbb{N}$, providing us with the number of instances of each element $x \in X$. As usual, we will enumerate the elements of a multiset C as follows: $C = \{r_1.x_1, \ldots, r_n.x_n\}$, meaning that $C(x_i) = r_i$, for all $i = 1, \ldots, n$, and $C(x) = 0$, for all $x \neq x_i$, $i = 1, \ldots, n$.
  The set of multisets over a set X will be denoted by $\mathcal{B}(X)$. For any $x \in X$ and $C \in \mathcal{B}(X)$ we say that $x \in C$ if and only if $C(x) > 0$.
- For any $C_1, C_2 \in \mathcal{B}(X)$, we define:
  - $C_1 + C_2 \in \mathcal{B}(X)$, where $\forall x \in X,\ (C_1 + C_2)(x) = C_1(x) + C_2(x)$.
  - $C_1 \subseteq C_2$ if and only if $\forall x \in X,\ C_1(x) \leq C_2(x)$.
  - If $C_1 \subseteq C_2$ we can define the subtraction $C_2 - C_1 \in \mathcal{B}(X)$, where $\forall x \in X,\ (C_2 - C_1)(x) = C_2(x) - C_1(x)$.

Definition 2 (Prioritized Colored Petri Nets): We define a prioritized colored Petri net (PCPN) as a tuple (P, T, A, V, G, E, π), where[1]:

- P is a finite set of *places*, with colors in a set Σ.
- T is a finite set of *transitions* ($P \cap T = \emptyset$).
- $A \subseteq (P \times T) \cup (T \times P)$ is a set of directed *arcs*.
- V is a finite set of *typed variables* in Σ, i.e. $Type(v) \in \Sigma$, for all $v \in V$.
- $G : T \longrightarrow EXPR_V$ is the *guard function*, which assigns a Boolean expression to each transition, i.e. $Type(G(t)) = Bool$.
- $E : A \longrightarrow EXPR_V$ is the *arc expression function*, which assigns an expression to each arc, constructed using variables, constants, operators and functions.
- $\pi : T \longrightarrow IN$ is the *priority function*, which assigns a priority level to each transition, where low values correspond to high priorities. We take 1 as the priority level of a transition with priority P_HIGH, 2 for P_NORMAL, 3 for P_LOW and 4 for P_LOW2

A marking on a place is defined as a multiset of colored tokens on it, and the marking of the Colored Petri Net is defined by the marking of all its places.

Definition 3 (Markings): Given a PCPN $N = (P, T, A, V, G, E, \pi)$, a marking M is defined as a function $M : P \longrightarrow \mathcal{B}(\Sigma)$, which assigns a multiset of colors to each place (which can be empty). The corresponding marked PCPN is denoted by (N, M).

We define the semantics for MPCPNs (Marked PCPNs) in a similar way as in [8], now taking into account that transitions have associated priorities. We first introduce the notion of binding, then the enabling condition and finally the firing rule for MPCPNs.

[1]We use the classical notation on Petri nets to denote the precondition and postcondition of both places and transitions:

$$\forall x \in P \cup T : {}^\bullet x = \{y \mid (y, x) \in A\} \quad x^\bullet = \{y \mid (x, y) \in A\}$$

Informally, a *transition binding* is just a function that assigns values to the variables that appear in a transition or in the arcs connected with it.

Definition 4 (Bindings): Let $N = (P, T, A, V, G, E, \pi)$ be a PCPN. A binding of a transition $t \in T$ is a function b that maps each variable $v \in Var(t)$ into a value $b(v) \in \Sigma$, where $Var(t)$ is defined as the set of variables that appear both in the guard of t and in the arc expressions of the arcs connected to t. We will denote by $B(t)$ the set of all possible bindings for $t \in T$.

Given an expression $e \in EXPR_V$, we will denote by $e\langle b \rangle$ the evaluation of e for the binding b. A binding element is then defined as a pair (t, b), where $t \in T$ and $b \in B(t)$. The set of all binding elements is denoted by BE. □

Given a marking M and a transition t, we say that a transition binding of t is *enabled* at (N, M) when $G(t)$ is *true* under this binding, for every $p \in {}^\bullet$ we have enough tokens in $M(p)$ with colors matching with the indicated in $E(p, t)$ and no other transition t' has a binding fulfilling these conditions with $\pi(t') < \pi(t)$.

Definition 5 (Enabling Condition): Let (N, M) where $N = (P, T, A, V, G, E, \pi)$ is a PCPN and M a marking of it. We say that a binding element $(t, b) \in BE$ is enabled if and only if the following conditions are fulfilled:

1) $G(t)\langle b \rangle = \text{true}$.
2) For all $p \in {}^\bullet$, $E(p, t)\langle b \rangle \subseteq M(p)$.
3) There is no other binding element $(t', b') \in BE$ fulfilling the previous conditions such that $\pi(t') < \pi(t)$. □

Hence, the following conditions must be satisfied for a transition to be fireable: it must be binding enabled (for some specific binding), its guard must evaluate to true with the selected binding and there cannot be another transition with a greater priority fulfilling these conditions (with this or another binding). The firing of an enabled transition binding is non-deterministic (according to the possible bindings), and a new marking is obtained from the associated binding: for every place $p \in {}^\bullet$ we remove the selected tokens matching with $E(p, t)$ and we add new colored tokens on the places $p' \in t^\bullet$, according to the expression $E(t, p')$.

Definition 6 (Firing Rule): Let (N, M) where $N = (P, T, A, V, G, E, \pi)$ is a PCPN, and M a marking of N, and an enabled binding element $(t, b) \in BE$. The firing of (t, b) is possible obtaining a new marking M' as follows: $\forall p \in P : M'(p) = M(p) - E(p, t)\langle b \rangle + E(t, p)\langle b \rangle$

The following example illustrates the above definitions.

Example 1: Let us consider the marked PCPN depicted in Figure 3, obtained from CPN Tools. Tokens in CPN Tools are drawn using the notation n‘v, meaning that we have n instances of a token with color value v. Besides, the symbol ‘++’ is used to represent the union of multisets in CPN Tools.

All places in the example have as color set INT (int), and the variables x, y, z, w are integers. Transitions are labeled with their associated guard and priority information, and arcs are labeled with the corresponding expressions.

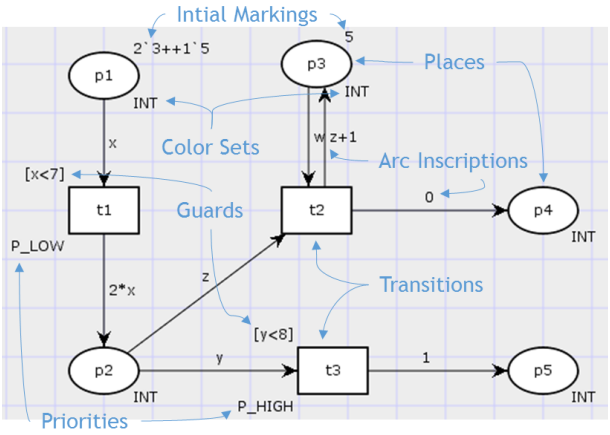


FIGURE 3. Graphical view of a PCPN.

Empty guards are always evaluated to true and empty priorities are considered as P_NORMAL.

From the initial marking shown in Figure 3 we can see that only transition $t1$ can be fired, and any token of those in $p1$ can be used for its firing (the binding can be either $x = 3$ or $x = 5$). Taking the binding $x = 5$, which fulfills the transition guard ($x < 7$), we can fire $t1$ which removes the token 5 from $p1$ and produces a new token on $p2$ with value 10. The only transition that can now be fired is $t2$ because the guard of $t3$ is not satisfied and the priority of $t1$ is lower than the priority of $t2$.

As a result, the token on $p3$ changes its value to 11, a new token with value 0 appears on $p4$ and two tokens with value 3 remain in $p1$. From this marking the sequence $t1; t3; t1; t3$ can be fired, thus reaching the final marking ($p1$ and $p2$ empty, 11 in $p3$, 0 in $p4$ and two tokens with value 1 in $p5$).

III. PCPN MODELING OF COMPLEX EVENT PROCESSING

In this section, we present the PCPN models for some of the most relevant elements supported by MEdit4CEP, such as events (simple and complex), pattern operators and actions. Let us see first an overview of how these elements are represented by using PCPNs:

- *Event capture*: For each type of event we will have a place whose tokens represent the specific events of this type that are used to feed the model. The color set of this place will be defined according to the properties of the specific event type.
- *Complex events*: These are generated by the application of the event patterns, and they are represented by specific places, whose color sets are defined according to the pattern schema that specifies the specific complex event.
- *Event patterns*: Each pattern will be modeled by a separate PCPN, in which we basically have a transition representing the pattern application and possibly other *control transitions and control places* that allow us to apply the specific filters the pattern requires. Thus, these transitions will have as input places the event capture places and/or the complex event places that they require to select the appropriate events, and they have as output places the complex event places corresponding to the events they produce. Guards and priorities will then be used in these transitions in order to enforce the correct pattern behavior.
- *Actions*: These are modeled by specific transitions that are associated to the different response actions. Their precondition places will be the complex event places related to their execution, and they have guards that specify the conditions under which their specific action must be performed.

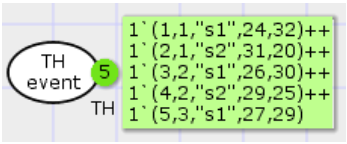


FIGURE 4. Event Place for the THevent Event Type.

A. CEP DOMAIN: EVENTS AND EVENT PROPERTIES

As mentioned before, according to MEdit4CEP, each type of event in the CEP domain is represented by a corresponding place, whose name will be the type of event. These places will be called *event places* and can be replicated as needed in the different patterns. Each event property has either a basic data type (integer, string, boolean, etc) which is represented in CPNTools by a corresponding basic color set (int, string, bool, etc) or a structured data type (a product color set in CPNTools). Furthermore, we have an ordered sequence of timed events, so tokens always have both a sequence number and a timestamp associated. Thus, the first two fields of all color sets for event places will be used to represent, respectively, the sequence number *(seqnumber)* and the instant at which the event was produced *(eventtime)*. It is important to notice that we are annotating the event timestamp as a field in the place color set, instead of using the timed capabilities of PCPNs (timed color sets), since the use of timed tokens entangles the translations unnecessarily, because we can only use timed tokens when they are available, and some patterns require a double processing of the input event sequence. Following these two fields we will have the data fields representing the event properties. Furthermore, all event properties are declared as variables in CPNTools, according to their respective data types.

Example 2: A simple type of event *THevent* is declared in EPL as follows:

```
create schema
    THevent(sensor string,temperature integer,
            humidity integer);
```

The corresponding declarations in CPNTools are:

```
colset TH = product INT*INT*STRING*INT*INT;
var sensor:STRING;
var seqnumber,eventtime,temperature,
                                    humidity:INT;
```

Figure 4 illustrates the previous declaration in CPNTools, where the place *THevent* is shown with a marking of 5 tokens (event instances) and the initial timestamp is 1.

The EPL event sequence declaration for the initial marking of *THevent* follows:

```
t=t.plus(1 seconds)
THevent={sensor='s1', temperature=24, humidity=32}
THevent={sensor='s2', temperature=31, humidity=20}
t=t.plus(1 seconds)
THevent={sensor='s1', temperature=26, humidity=30}
THevent={sensor='s2', temperature=29, humidity=25}
t=t.plus(1 seconds)
THevent={sensor='s1', temperature=27, humidity=29}
```

□

B. EVENT PATTERNS

According to MEdit4CEP, event patterns are defined by means of pattern operators and expressions constructed using condition operators (arithmetic, comparison and logical) on values, properties or events. Besides, an event pattern can be applied on a bounded set of events from an event stream (Data Window) and time conditions can also be considered in a pattern. In this paper, in order to gain readability and not to be repetitive, we only present a subset of the most relevant operators supported by MEdit4CEP for detecting situations of interest, but illustrating the guidelines that we should follow for the translation of the other patterns.

In this paper, therefore, we only deal with the pattern operators *every*, *followed by* and two combinations of them. We also include the *Sliding Time Interval Data Windows*, as illustration of the four types of Data Windows supported by MEdit4CEP. Thus, MEdit4CEP's operators *EveryDistinct, Range, Repeat, Until* and *While* are not considered in this paper, but they can be obtained following the same principles that we use for the patterns here presented. Pattern times are not considered either. Their translations to PCPNs are also obtained by applying the same techniques.

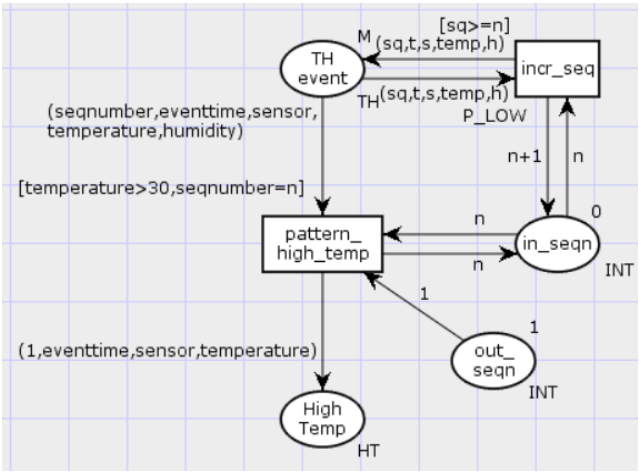
Conditional expressions will be translated to CPN-ML expressions in CPNTools. CPN-ML is a functional programming language based on Standard ML [17], and the conditional expressions are then constructed using the variables declared for events and properties.

1) PATTERN OPERATORS

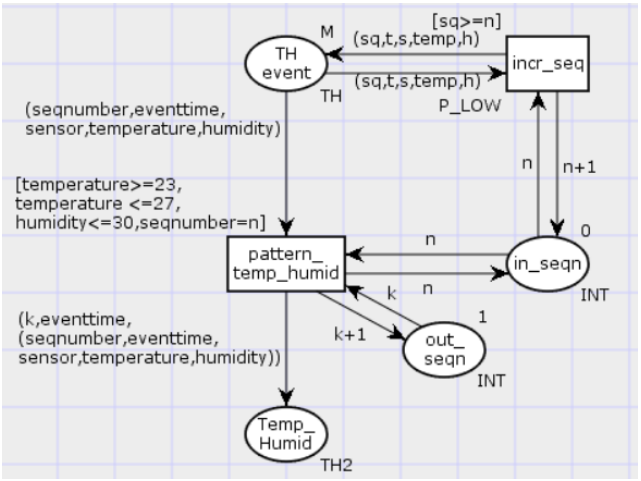
Let us start with a pattern that only selects the first event that fulfills a certain condition:

```
@Name('HighTemp')
insert into HighTemp
select a1.sensor as sensor,
       a1.temperature as temperature
from  THevent a1 where a1.temperature > 30
```

Which takes the first event of type THevent with a temperature greater than 30 and produces a complex event of type *HT*, with two properties (sensor identification and temperature) and the corresponding sequence number and event time. The PCPN for this pattern is shown in Figure 5.a. Transition *pattern_high_temp* has a CPN-ML guard selecting the tokens on the place *THevent* that fulfill the indicated condition, but this transition can only be fired once, since there is only one token on the place *out_seqn*. The place *in_seqn* is initially marked with one token of value 0, which is increased



(a)



(b)

FIGURE 5. PCPNs for Patterns *high_temp* and *temp_humid*. (a) PCPN for *high_temp*. (b) PCPN for *temp-humid*.

progressively by transition *incr_seq*. Due to its low priority, transition *incr_seq* can only be fired when the pattern transition is not fireable, so the latter is only fired for the first event in the sequence fulfilling the conditions. Taking the initial marking of Example 2 we have obtained the token (1, 1, "*s*2", 31) on the place *HighTemp*.

The pattern operator **every** selects every event belonging to the specified type that fulfills the indicated condition (if a condition has been defined). The following pattern *temp_humid* selects all events for which the temperature is between 23 and 27 and the humidity is less than or equal to 30 which are acceptable ranges of temperature and relative humidity for comfort in summer.

```
@Name('temp_humid')
insert into temp_humid
select a1.* from pattern
 [(every a1 =  THevent(a1.temperature >= 23
 and a1.temperature <=27 and a1.humidity <= 30))]
```

The PCPN for this pattern is shown in Figure 5.b. Transition *pattern_temp_humid* has again a CPN-ML guard that selects the tokens on the place *THevent* that fulfill the indicated conditions. These tokens are removed from the place *THevent* and they are inserted into the event place *Temp_Humid*, extended with their new output sequence number and time (color set *TH2*). Taking again the initial marking of Example 2 we have obtained the following tokens on *Temp_Humid*:

{(1, 2, (3, 2, "*s*1", 26, 30)), (2, 3, (5, 3, "*s*1", 27, 29))}.

The pattern operator **followed by** (−>) determines a pattern expression that must be followed by another. Let us see first the use of this pattern only using conditional expressions in both sides:

```
@Name('greater_temp')
insert into greater_temp
select a2.sensor as sensor,
       a1.temperature as temp1,
       a2.temperature as temp2
from pattern [(a1 =  THevent ->
       a2 =  THevent((a2.sensor = a1.sensor
       and a2.temperature > a1.temperature)))]
```

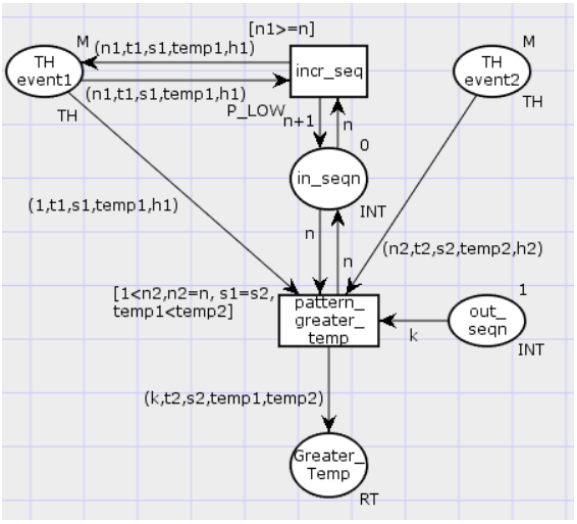


FIGURE 6. PCPN for the "greater_temp" Pattern.

This pattern takes the first event of the input sequence *a*1, and then it looks for the first event *a*2 in the sequence such that *a*1 precedes *a*2, they come from the same sensor and the temperature of *a*2 is greater than that of *a*1. The PCPN is shown in Figure 6, where the initial place *THevent* has been cloned (*THevent1* for *a*1 events and *THevent2* for *a*2 events). The place *in_seqn* is again used to represent an increasing sequence number, so as to fire the pattern transition for the first event *a*1 and the first event *a*2 of the sequence fulfilling the conditions.

The firing of transition *pattern_greater_temp* produces a new token on the place *Greater_Temp*, with sequence number 1, time of *a*2, sensor identification and both temperature values of *a*1 and *a*2. Taking the initial marking of Example 2 for both places *THevent1* and *THevent2* we have obtained the token (1, 2, "*s*1", 24, 26) on *Greater_Temp*.

Let us consider now two relevant combinations of the patterns *every* and *followed by*, namely, *every (A −> B)* and *every A −> B*, whose translations are depicted in Figure 7. Without loss of generality we have considered in these figures, respectively, the following EPL declarations:

```
create schema ME(id string,k integer);
@Name('every1')
insert into every1
from pattern [ every (a1 =  ME(a1.id="A") ->
                      a2 =  ME(a2.id = "B"))];
@Name('every2')
insert into every2
select a1.id as a1id,  a1.k as k1,
       a2.id as a2id,  a2.k as k2
from pattern [every  a1 =  ME(a1.id="A") ->
                     a2 =  ME(a2.id = "B")];
```

The PCPN structure for a different event type and/or different conditions A, B would be exactly the same; only the color sets on the event places and the guards on the *next_A* and pattern transitions would need an adjustment.

In both PCPNs the initial place has been replicated *(Ev1, Ev2)*, so they contain the same initial marking (event sequence). For the pattern *every1* (every (A −> B)) let us observe that transition *next_A* takes the next *A* event from the input sequence, always starting with the sequence number stored on the token on *in_seqn*, which is progressively increased by transition *incr_seq*. The *A* event token taken from *Ev1* is then stored on the place *Ev1A*. Transition *incr_seq* has a low priority, so it will increase the sequence number on the token of *in_seqn* until the pattern transition can be fired. Once the pattern transition is fired the token on the place *control* is recovered, which allows the transition *next_A* to find the next *A* after the found *B*.

For illustration, let us consider the following input sequence:

```
t=t.plus(1 seconds)
ME={id="A", k=1} ME={id="B", k=1} ME={id="C", k=1}
ME={id="B", k=2} ME={id="A", k=2} ME={id="D", k=1}
ME={id="A", k=3} ME={id="B", k=3} ME={id="E", k=1}
ME={id="A", k=4} ME={id="F", k=1} ME={id="B", k=4}
```

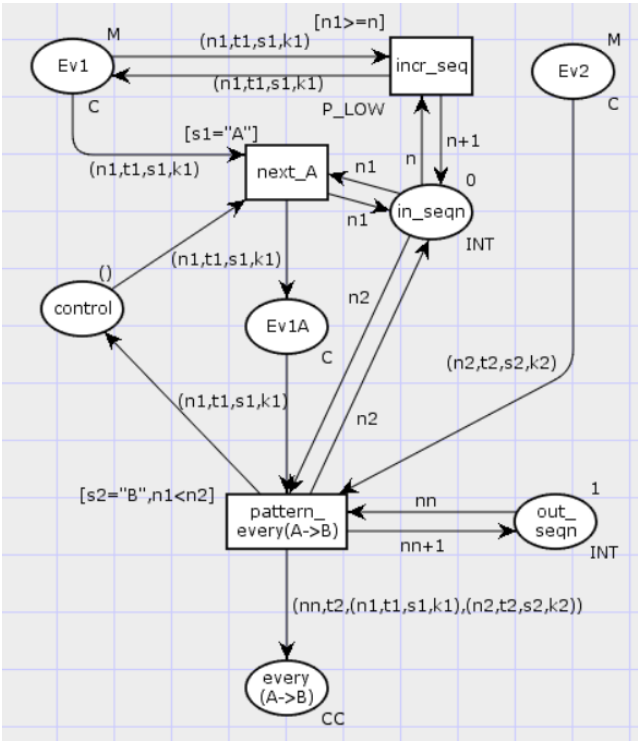
For which the corresponding initial marking on *Ev1, Ev2* is

$$M = \{(1, 1, \text{"}A\text{"}, 1), (2, 1, \text{"}B\text{"}, 1), (3, 1, \text{"}C\text{"}, 1), (4, 1, \text{"}B\text{"}, 2), (5, 1, \text{"}A\text{"}, 2), (6, 1, \text{"}D\text{"}, 1), (7, 1, \text{"}A\text{"}, 3), (8, 1, \text{"}B\text{"}, 3), (9, 1, \text{"}E\text{"}, 1), (10, 1, \text{"}A\text{"}, 4), (11, 1, \text{"}F\text{"}, 1), (12, 1, \text{"}B\text{"}, 4)\}$$

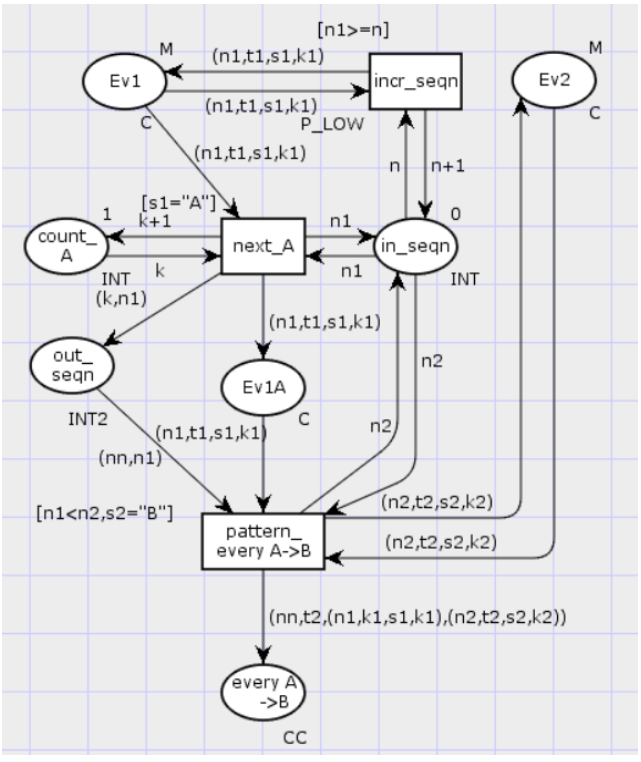
The resulting final marking on the place *every (A −> B)* is

$$M' = \{(1, 1, (1, 1, \text{"}A\text{"}, 1), (2, 1, \text{"}B\text{"}, 1)), (2, 1, (5, 1, \text{"}A\text{"}, 2), (8, 1, \text{"}B\text{"}, 3)), (3, 1, (10, 1, \text{"}A\text{"}, 4), (12, 1, \text{"}B\text{"}, 4))\}$$

Regarding the pattern *every A −> B* (Figure 7.b), the place *count_A* is now used to count the number of *A* events



(a)



(b)

FIGURE 7. PCPNs for "every-followed by" Patterns. (a) CPN for every(A -> B). (b) CPN for every A -> B.

processed from the input sequence (*Ev1*). These *A* events are written again in *Ev1A*, and we annotate its number of *A* event (k) and its sequence number ($n1$) into a token on the place *out_seqn*. Sequence numbers are increased again progressively by transition *incr_seqn*, until the pattern transition is fired or a new *A* event is found on the input sequence. When a B event is found, the pattern transition fires (possibly several times) and the corresponding matches are written into the output place *every A −> B*.

For the same initial marking considered above we have now obtained the following final marking:

$$M' = \{(1, 2, (1, 1, \text{“}A\text{“}, 1), (2, 2, \text{“}B\text{“}, 1)),\\ (2, 8, (5, 2, \text{“}A\text{“}, 2), (8, 8, \text{“}B\text{“}, 3)),\\ (3, 8, (7, 3, \text{“}A\text{“}, 3), (8, 8, \text{“}B\text{“}, 3)),\\ (4, 12, (10, 4, \text{“}A\text{“}, 4), (12, 12, \text{“}B\text{“}, 4))\}$$

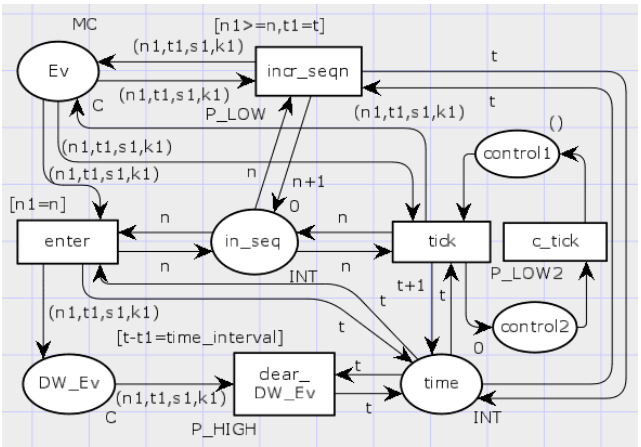


FIGURE 8. PCPN Model for the Sliding Time Interval Data Windows.

2) DATA WINDOWS

As mentioned before, we only describe the PCPN transformation for *Sliding Time Interval Data Windows* (Figure 8). The other *Data Windows*, namely *Batching Event Interval, Batching Time Interval* and *Sliding Event Interval Data Windows* can be obtained in a similar way.

In this case, a place *time* with one token representing the current time is now included, as well as a transition *tick* that increases the time until reaching the time for the next event in the input sequence. The current sequence number under processing is captured again by the token on the place *in_seqn*, as in the previous translations. Place *DW_Ev* (*Data Window Event Place*) will contain at each instant the events corresponding to the current time slide.

Let us see how this PCPN works at each cycle (time instant). Transition *tick* is fired to advance the current time, which can be followed by several firings of transition *clear_DW_Ev* in order to remove the tokens that are too old to remain in this time slide. The high priority of this transition enforces its firing if it is enabled. In the case that there is no event in the input sequence with a time equal to the current time the only enabled transition will be *c_tick*, which allows a new firing of *tick* to advance the current time and start a new cycle. Otherwise, when the following

event in the sequence has an event time equal to the current time, transition *incr_seqn* is fired to increase the sequence number on the place *in_seq* so as to advance in the input event sequence. Transition *enter* is then fired, writing the corresponding event token into the place *DW_Ev*. Should we have more events in the input sequence with the same event time, transition *incr_seqn* is fired again followed by *enter* to deposit these event tokens into the place *DW_Ev*. Notice that *incr_seqn* has a *P_LOW* priority, so we only advance in the sequence after entering the previous event token into *DW_Ev* and we only advance to the following event when its event time is the current time, as mentioned above. Once there are no other events in the sequence with their event time equal to the current time, then it follows that the only enabled transition is *c_tick*, which has the lowest priority in the PCPN (P_LOW2), allowing a new firing of *tick* to advance the current time and start a new cycle.

The specific application of a *Sliding Time Interval Data Window* varies depending on the defined pattern. The actions to be performed must be included just before *tick* fires. In the following section we illustrate the applicability of a *Sliding Time Interval Data Window* by using a particular case study.

IV. CASE STUDY

In this section we analyze a health care case study concerning the monitoring of uterine contractions of pregnant women in a hospital. Obviously, in this case, the end users could not be experts in CEP, EPL or CPN, so our intention is to show, through this case study, how our work could contribute to improve current health care information systems.

As we said before, we focus on the events produced previously to the process of childbirth by considering the monitoring of uterine contractions. Specifically, we only consider the duration —beginning to end of one contraction (seconds)— and the frequency —beginning of one contraction to beginning of the next (minutes).

The patterns of interest are related to the number of times that frequency and/or duration exceeds a predetermined value in a given period of time. Thus, for each patient we only need to indicate her name and the duration of each uterine contraction, with its corresponding timestamp.

In order to define these event patterns for this application domain, the first task to be done by the domain expert is the CEP domain definition making use of MEdit4CEP. As can been seen in Figure 9, this domain contains the *Patient* event type along with its event properties: *ts* (event timestamp in minutes), *id* (patient id) and *contrDuration* (contraction duration in seconds). Moreover, the domain name has been set to *pregnancy*, a textual description has been indicated for the domain and a customized icon has been assigned to the event type, improving usability.

This CEP domain model is automatically validated and stored, and can be transformed into EPL syntax as follows:

```
create schema
Patient(ts integer, id string,
                        contrDuration integer);
```

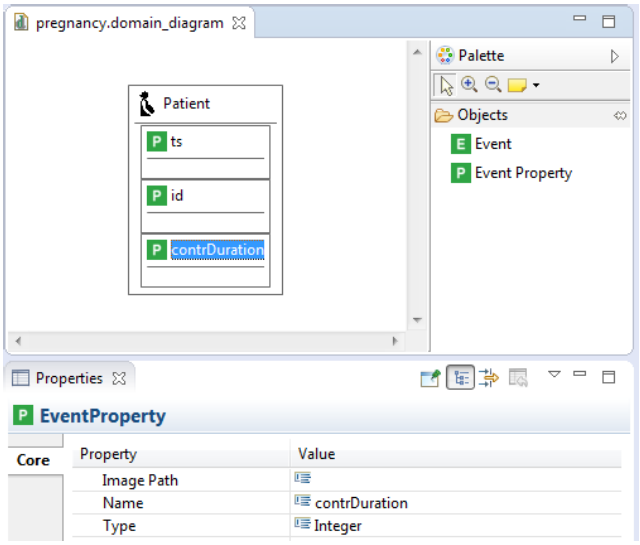


FIGURE 9. Pregnancy domain modeled using MEdit4CEP.

Once designed the pregnancy domain, the event pattern editor has been automatically reconfigured for this domain, i.e. the *Patient* event type has been added as a tool in the Simple Events category of the editor palette (see Figures 10 and 11). This means that end users do not have to worry about how to define event types and their properties, since these are graphically represented when dragging and dropping the tool. In addition, they cannot modify given event types, avoiding the creation of incorrect event patterns for the same domain.

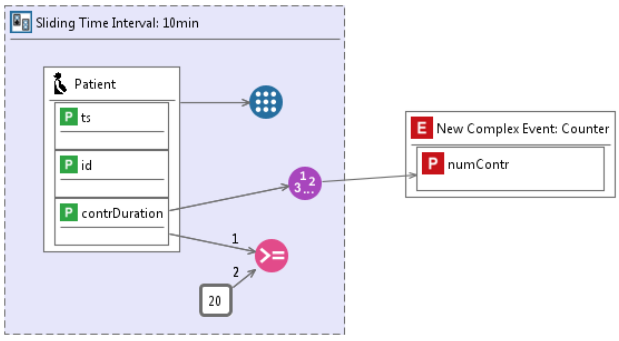


FIGURE 10. Counter event pattern modeled using MEdit4CEP.

Figure 10 shows the *Counter* event pattern modeled using MEdit4CEP. This pattern will counter the number of events with a contraction duration greater than 20 seconds in time slides of 10 minutes. Once the pattern has been designed, it has to be automatically validated and stored after checking that the model is correct and well formed —by using the validation rules. Afterwards, the editor has been automatically reconfigured adding the new complex event type *Counter* as a tool in the Complex Events category of the editor palette. This allows end users to be able to graphically define an event pattern hierarchy by dragging and dropping previously added tools from the Complex Events category into a new event

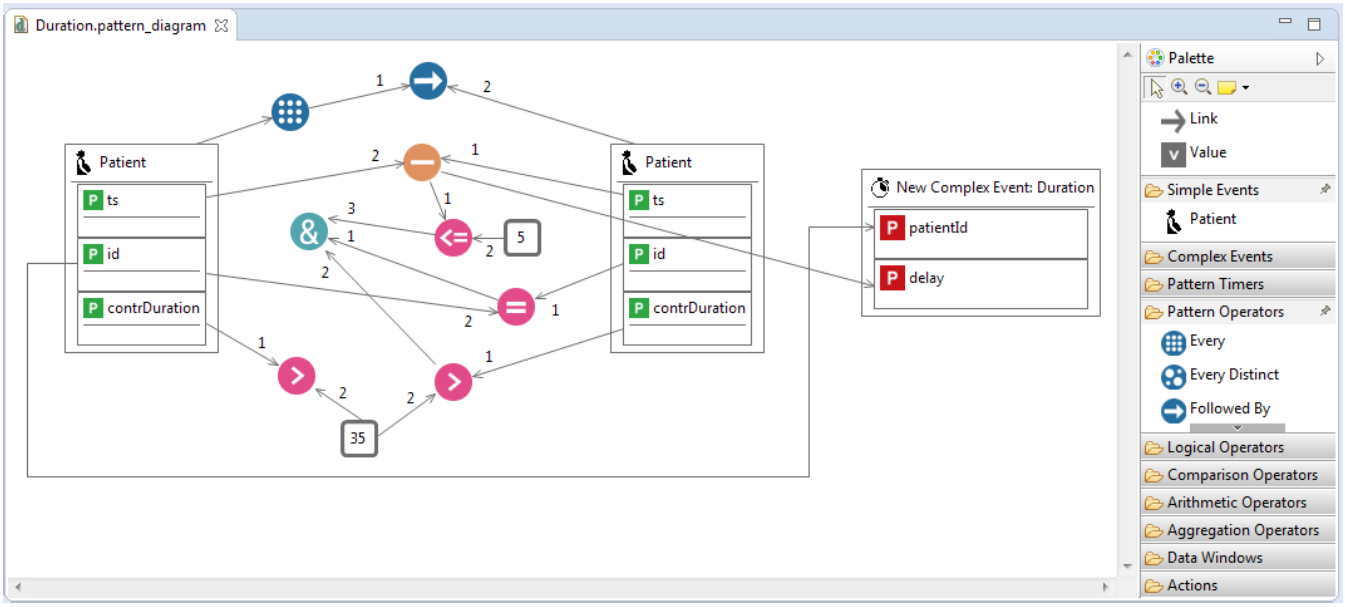


FIGURE 11. Duration event pattern modeled using MEdit4CEP.

pattern model. The EPL code automatically generated for this pattern is as follows:

```
@Name('Counter')
insert into Counter
select count(a1.contrDuration) as numContr
from pattern [(every a1 = Patient(a1.contrDuration
   >= 20))].win:time(10 minutes)
```

Figure 11 illustrates the event pattern *Duration* modeled by the end user. This pattern looks for patients who have had at least two contractions with a duration greater than 35 seconds within a period of 5 minutes. The complex event type *Duration* is then added to the Complex Events category of the editor palette. The EPL code generated for this pattern is as follows:

```
@Name('Duration')
insert into Duration
select a1.id as patientId,
   (a2.ts - a1.ts) as delay
from pattern [((every a1 = Patient(a1.contr
   Duration > 35)) -> a2 = Patient((a2.id = a1.id
   and a2.contrDuration > 35 and (a2.ts - a1.ts)
   <= 5)))]
```

For the PCPN transformation the following color sets, variables and values are declared in CPNTools:

```
colset C=product INT*INT*STRING*INT;
colset INT2=product INT*INT;
var n,t,n1,t1,k1:INT;
var s1:STRING;
val time_interval=10;
```

Then, the corresponding PCPNs for the two modeled event patterns are shown in Figures 12 and 13. The first one is a *Sliding Time Interval Data Window*, in which we have now included the corresponding event processing, which is to count the events on the place *DW_Ev* fulfilling the indicated

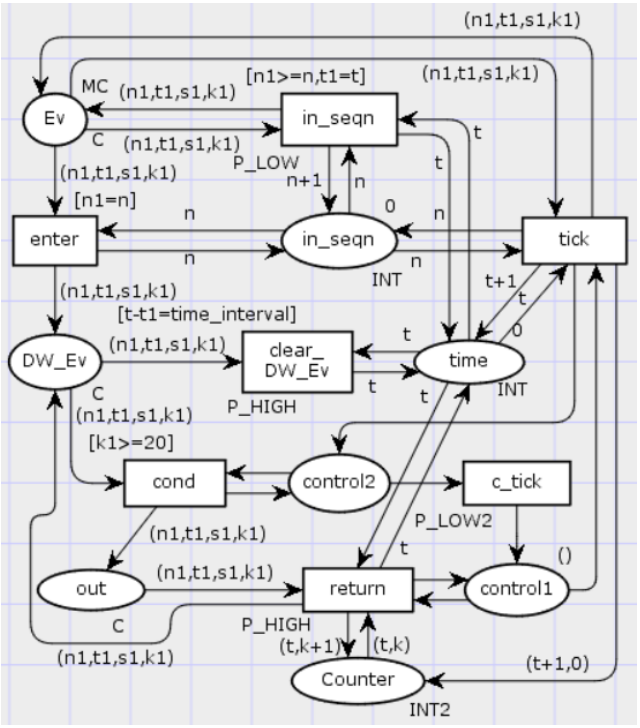


FIGURE 12. PCPN for the Pattern *Counter*.

condition (contraction duration greater than 20) at each time slide. A transition *cond* has been included to select the event tokens fulfilling this condition, which are written into the place *out*. Once all of these event tokens have been transferred to place *out* the transition *c_tick* is fired and then the transition *return* is fired (due to its high priority) to recover all these tokens to the place *DW_Ev* and counter the tokens

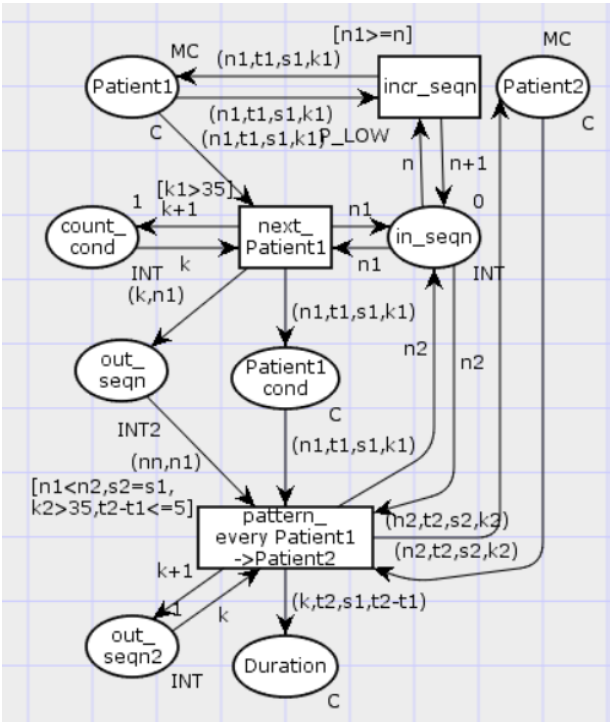


FIGURE 13. PCPN for Pattern *Duration*.

on the current times slide (t). After recovering the tokens on *DW_Ev* the transition *tick* becomes enabled, so as to start a new cycle. Figure 13, the second pattern, corresponds to a "*every A −> B*" case.

Taking, for instance, the following EPL specification as event data:

```
t=t.plus(1 min)
 Patient={id ='Alice', contrDuration =10 , ts=1}
 Patient={id ='Barbara', contrDuration =20, ts=1}
t=t.plus(8 min)
 Patient={id ='Alice', contrDuration =20, ts=9}
 Patient={id ='Barbara',  contrDuration =36, ts=9}
t=t.plus(3 min)
 Patient={id ='Alice', contrDuration =38, ts=12}
t=t.plus(2 min)
 Patient={id ='Barbara', contrDuration =39, ts=14}
t=t.plus(2 min)
 Patient={id ='Alice', contrDuration =40,ts=16}
t=t.plus(8 min)
 Patient={id ='Barbara', contrDuration =42,ts=24}
 Patient={id ='Carla', contrDuration =10,ts=24}
t=t.plus(1 min)
 Patient={id ='Alice', contrDuration =45,ts=25}
t=t.plus(4 min)
 Patient={id ='Alice',  contrDuration =36, ts=29}
t=t.plus(2 min)
 Patient={id ='Barbara', contrDuration =20, ts=31}
```

which corresponds to the initial marking:

$$M = \{(1, 1, \text{"}Alice\text{"}, 10), (2, 1, \text{"}Barbara\text{"}, 20), (3, 9, \text{"}Alice\text{"}, 20), (4, 9, \text{"}Barbara\text{"}, 36), (5, 12, \text{"}Alice\text{"}, 38), (6, 14, \text{"}Barbara\text{"}, 39), (7, 16, \text{"}Alice\text{"}, 40), (8, 24, \text{"}Barbara\text{"}, 42), (9, 24, \text{"}Carla\text{"}, 10), (10, 25, \text{"}Alice\text{"}, 45), (11, 29, \text{"}Alice\text{"}, 36), (12, 31, \text{"}Barbara\text{"}, 20)\}$$

meaning a woman "Alice" with a contraction duration of 10 seconds at instant 1, a woman "Barbara" with a contraction duration of 20 seconds at instant 1 too, woman "Alice" again with a contraction duration of 20 seconds at instant 9 and so on.

The final marking obtained for the first pattern at the place *Duration* follows is

$$M' = \{(1, 14, \text{"}Barbara\text{"}, 5), (2, 16, \text{"}Alice\text{"}, 4), (3, 29, \text{"}Alice\text{"}, 4)\}$$

This marking corresponds to patient "Barbara" who has had two contractions longer than 35 seconds in a period of 5 minutes, where the last contraction has been at time 14 and the previous one 5 minutes before, that is, at time 9; and to patient "Alice", who has as well matched the pattern in two occasions, but in these cases the last contractions have been at times 16 and 29, respectively, and both in a period of 4 minutes.

The final marking obtained for the second pattern at the place *Counter* follows:

$$M' = \{(1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1), (9, 3), (10, 3), (11, 2), (12, 3), (13, 3), (14, 4), (15, 4), (16, 5), (17, 5), (18, 5), (19, 3), (20, 3), (21, 3), (22, 2), (23, 2), (24, 2), (25, 3), (26, 2), (27, 2), (28, 2), (29, 3), (30, 3), (31, 4)\}$$

Therefore, at the time slides of instants 1 to 8 we have only one event fulfilling the condition, at times 9 and 10 we have two, at times 11 to 18 one event and so on. Notice that we have not stored the specific events on an output place, but the inclusion of such a place would be immediate, as an output place of transition *return*. Since we have used a 10-min sliding window in the pattern *Counter*, 31 complex events have been created as a result of analyzing the events coming from instant 1 until 31. It is noteworthy that if the end user had graphically modeled by mistake this pattern with a batch windows, a unique complex event had been created after finishing every 10-min window, i.e. only 4 events. Thus, this output validation would be an appropriate mechanism to detect semantic errors in event patterns and notify them to end users by using MEdit4CEP.

In this case study the initial marking of the obtained PCPN has been introduced manually, and the interpretation of the final marking has also been done by hand. It is evident that the end user who defines the event patterns will not likely be an expert in Petri nets models and EPL, so he would not be able to simulate the Petri net model and translate back the found errors, so we plan to integrate in the MEdit4CEP tool some new features to allow him to define specific initial situations or generic conditions that should hold in general, so as to translate them into initial markings and/or PCPN extensions to check that the designed event patterns are correct. All of this corresponds to phase 4 of the methodology indicated in the Introduction, and will be the matter of further research in this area.

V. RELATED WORK

The most related work is [18], in which Weidlich et al. have defined a model of event processing networks using colored Petri nets (CPN) with priorities and time (PTCPNs). In that paper, the EPN (Event Processing Networks) architecture is shown as the overall system and therefore they propose a general translation of this concept. Besides, in that paper no formal translation is defined for each specific pattern nor for their combination. An only example, which is implemented in the ETALIS framework [19], is provided to illustrate the technique they use, but no explanations are provided about the way in which the combination ''every + followed by'' expressed in its equivalent ETALIS pattern is translated. The ''every + followed by'' is one of the patterns that will produce problems with timestamps in the tokens used in TCPNs, but this is not covered by that paper. For that reason, we have not considered Timed Colored Petri Nets (Timed CPNs), although the use of timestamps can make it easier the translation for the patterns isolated. For instance the translation of Time Interval Data Windows using the timed model is clearly easier. In this way, the most important difference with respect to that paper is that we provide a framework able to compound several patterns together in a compositional way, thus allowing the automatization of a CEP formalization as well as the integration of this framework in the MEdit4CEP tool, so that users can analyze the behavior of compositional patterns using a bottom-up approach. Our aims are then focused on a specific set of patterns used in the CEP engine, which might be graphically modeled by non-experts on EPLs, since this is the idea followed by the MEdit4CEP tool. The PCPN can then be obtained from this graphical specification, so that it can be immediately used in CPNTools for the validation phase.

In [20] we can also find a methodology to model CEP using Timed Net Condition Event System (TNCES) [21] and its Application to a Manufacturing Line is shown as an example. NCES is another Petri Net derived formalism, based on Condition Event Systems which provide a modular modeling formalism for discrete event dynamic systems. The modules of each of the devices are interconnected by means of their input/output behavior to form the uncontrolled system model. TNCES is the timed extension of NCES based on arc-timed Petri nets. The main differences with respect to our paper is that we use a different formalism of Petri Nets and that we plan the integration of the complete framework in the MEdit4CEP tool.

Another formal approach for the modeling of complex event systems has been proposed by Hinze and Voisard [22], in which a parameterized event algebra (EVA) is defined to support adaptable event composition, including temporal restriction, by the notion of relative time. A temporal logic, the TESLA language [23], has been defined by Cugola and Margara. TESLA is a highly expressive and flexible language in a rigorous framework, by offering content and temporal filters, negations, timers, aggregates, and fully customizable policies for event selection and consumption. A timed automata formalization of complex event systems can be found in [24] and [25], where the Sase+ pattern language is introduced. Sase + defines a precise semantics in terms of timed automata with similar results to the work introduced in TESLA. Another formalization using timed automata is presented by Ericsson et al. in [26], in which the events and rules specified for CEP applications are analyzed for design errors using the tool REX [27], implemented by Ericsson and Brendtsson. The paradigm used in this case for CEP patterns is the event condition action (ECA) [28]. REX as MEdit4CEP aims at aiding final user to define the CEP systems but with the difference that REX targets at helping non-experts in formal methods to define the properties that a pattern should satisfy, whereas MEdit4CEP assumes the non-expertise in the CEP pattern language itself, thus targeting a wider group.

There are also many other languages for processing real-time data, such as CQL [24], [29], ESL [30] and streaQuel [31]. These EPLs can be classified into the following language styles [10]: stream-oriented, rule-oriented and imperative. Further information about existing EPLs can be found in the survey by Cugola and Margara [3].

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a PCPN model that together with CPNTools makes possible the modeling, simulation, analysis and semantic validation of the situations of interest needed to be detected in a CEP-based system for a particular application domain by using event patterns. By extending MEdit4CEP, a model-driven solution for real-time decision making in SOA 2.0, with this PCPN model we allow end users not only to easily and graphically define these patterns and validate them syntactically but also semantically, before transforming them automatically into a particular EPL.

Summarizing, we have not only an event pattern graphical representation, but also the capability to perform formal analysis, and therefore semantic analysis, by means of the PCPN model obtained and the CPN Tools. This formal analysis is twofold. On the one hand, users can interact with the model itself by performing a step by step debugging, since the tool allows to simulate the model. With this in mind, users can specify a concrete scenario by providing the initial marking to check whether the model works as expected. By doing this, users can observe the results of the individual steps of the simulation, which represent the different EPL operators, as we can observe at the end of the case study where a user can detect whether the preferred operator has been used, that is, if the specified pattern behaves as expected.

On the other hand, there are certain advantages of performing automatic simulations. An automatic simulation allows us to actually execute the EPL code and compare the obtained output, that is, we can compare whether the results obtained from a given input are the same when we execute the EPL code in the Esper EPL online tool and in CPN Tools.

As future works, firstly we intend to extend the transformations to more event patterns elements provided by

MEdit4CEP. Secondly, regarding the use of the technique by non-experts, we are planning an automatic translation from the models created by using the graphical tool MEdit4CEP, which does allows the user to easily define patterns, to PCPNs following the methodology described in this paper. This automatic translation will be integrated in MEdit4CEP, and this extension will also consider the validation of certain desired behaviors only using the interface.

Furthermore, the results obtained should be provided to the designer in a comprehensive form, so he can interpret them easily. Moreover, there are other aspects to be also considered using the advantages offered by the PCPN formalism. For instance, taking the same ideas as we used in [32] and via an automatic generation of initial markings we could accomplish a quantitative analysis of the model, so as to predict the system behavior and discover potential problems.

Finally, we can also profit from the verification capabilities offered by CPNTools, using the state space generated to check certain properties of interest. This final goal has an important limitation, which is the state space explosion, as a consequence of the enormous amount of events that we usually have in these systems.

Other possibilities can also be explored, like discovery of data-flow errors, using similar ideas as Trcka et al. [33], with questions expressed in terms of a temporal logic, or the use of unfolding techniques of Petri nets in order to prove the PCPN soundness, as Liu et al. [34] have done in the context of workflow nets.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions which allowed them to improve the paper. Boubeta-Puig thanks the hospitality received by the Real-Time and Concurrent Systems Research Group at the University of Castilla-La Mancha, Spain, when visiting them, where part of this work was developed.

REFERENCES

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

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

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

[4] 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.

[5] M. Papazoglou, *Web Services and SOA: Principles and Technology*, 2nd ed. Essex, U.K.: Pearson Education, 2012.

[6] J. Boubeta-Puig. (2016). *MEdit4CEP Tool*, accessed on Nov. 3, 2016. [Online]. Available: https://ucase.uca.es/medit4cep/

[7] T. Stahl, M. Völter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. Hoboken, NJ, USA: Wiley, 2006.

[8] K. Jensen and L. M. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Berlin, Germany: Springer-Verlag, 2009.

[9] CPN Tools. (2016). *CPNTools Homepage*, accessed on Nov. 3, 2016. [Online]. Available: http://www.cpntools.org/

[10] O. Etzion and P. Niblett, *Event Processing in Action*. Greenwich, CT, USA: Manning Publications Co., 2010.

[11] EPTS. *Event Processing Technical Society, Event Processing Glossary-Version 2.0*, accessed on Nov. 3, 2016. [Online]. Available: http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf

[12] EsperTech. (2016). *Esper—Complex Event Processing*, accessed on Nov. 3, 2016. [Online]. Available: http://www.espertech.com/esper/

[13] K. M. Chandy, *Event Processing: Designing IT Systems for Agile Companies*, 1st ed. New York, NY, USA: McGraw-Hill, 2010.

[14] J. Boubeta-Puig, G. Ortiz, and I. Medina-Bulo, "ModeL4CEP: Graphical domain-specific modeling languages for CEP domains and event patterns," *Expert Syst. Appl.*, vol. 42, no. 21, pp. 8095–8110, Nov. 2015.

[15] W. M. P. van der Aalst and C. Stahl, *Modeling Business Processes: A Petri Net-Oriented Approach* (Cooperative Information Systems Series). Cambridge, MA, USA: MIT Press, 2011.

[16] K. Jensen, *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use* (Monographs in Theoretical Computer Science). Berlin, Germany: Springer-Verlag, 1997.

[17] J. D. Ullman, *Elements of ML Programming*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1998.

[18] M. Weidlich, J. Mendling, and A. Gal, "Net-based analysis of event processing networks—The fast flower delivery case," in *Proc. 34th Int. Conf. Appl. Theory Petri Nets Concurrency*, 2013, pp. 270–290.

[19] Etalis. *Event-driven Transaction Logic Inference System*, accessed on Nov. 3, 2016. [Online]. Available: https://code.google.com/archive/p/etalis/

[20] 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. 10th IEEE Int. Conf. Ind. Informat. (INDIN)*, 2012, pp. 380–385.

[21] M. Rausch and H.-M. Hanisch, "Net condition/event systems with multiple condition outputs," in *Proc. IEEE Symp. Emerg. Technol. Factory Autom.*, Oct. 1995, pp. 592–600.

[22] A. Hinze and A. Voisard, "EVA: An event algebra supporting complex event specification," *Inf. Syst.*, vol. 48, pp. 1–25, Mar. 2015.

[23] G. Cugola and A. Margara, "TESLA: A formally defined event specification language," in *Proc. 4th ACM Int. Conf. Distrib. Event-Based Syst. (DEBS)*, 2010, pp. 50–61.

[24] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *Proc. ACM-SIGMOD*, New York, NY, USA, 2008, pp. 147–160.

[25] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman, "On supporting Kleene closure over event streams," in *Proc. ICDE*, 2008, pp. 1391–1393.

[26] A. Ericsson, P. Pettersson, M. Berndtsson, and M. Seiriö, "Seamless formal verification of complex event processing applications," in *Proc. Inaugural Int. Conf. Distrib. Event-Based Syst. (DEBS)*, 2007, pp. 50–61.

[27] A. Ericsson and M. Berndtsson, "REX, the rule and event eXplorer," in *Proc. Inaugural Int. Conf. Distrib. Event-Based Syst. (DEBS)*, 2007, pp. 71–74.

[28] K. R. Dittrich, S. Gatziu, and A. Geppert, "The active database management system manifesto: A rulebase of ADBMS features," in *Rules in Database Systems* (Lecture Notes in Computer Science), vol. 985. Berlin, Germany: Springer-Verlag, 1995, pp. 3–20.

[29] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *VLDB J.*, vol. 15, no. 2, pp. 121–142, Jun. 2006.

[30] Y. Bai, H. Thakkar, H. Wang, C. Luo, and C. Zaniolo, "A data stream language and system designed for power and extensibility," in *Proc. 15th ACM Int. Conf. Inf. Knowl. Manage. (CIKM)*, 2006, pp. 337–346.

[31] S. Chandrasekaran *et al.*, "TelegraphCQ: Continuous dataflow processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 2003, p. 668.

[32] V. Valero, H. Macià, and G. Dıaz, "Quantitative analysis of the publish/subscribe paradigm in the context of Web service resources with timed colored Petri nets," in *Proc. 29th Eur. Simulation Modeling Conf. (ESM)*, 2015, pp. 73–80.

[33] N. Trčka, W. M. P. van der Aalst, and N. Sidorova, "Data-flow anti-patterns: Discovering data-flow errors in workflows," in *Proc. 21st Int. Conf. Adv. Inf. Syst. Eng. (CAiSE)*, 2009, pp. 425–439.

[34] G. Liu, W. Reisig, C. Jiang, and M. Zhou, "A branching-process-based method to check soundness of workflow systems," *IEEE Access*, vol. 4, pp. 4104–4118, 2016.

HERMENEGILDA MACIÀ received the degree in mathematics from the University of Valencia and the Ph.D. degree in computer science from the University of Castilla-La Mancha in 2003. She is currently an Associate Professor with the Department of Mathematics, Computer Science School of Albacete, University of Castilla-La Mancha, Spain. She has authored research articles in reputed journals of mathematics and computer science. Her main research interests include the theoretical study and applications of formal methods, such as process algebras and Petri nets, considering timed, probabilistic, and stochastic extensions.

VALENTÍN VALERO received the degree in mathematics from the Complutense University of Madrid in 1987, and the Ph.D. degree in mathematics from the Department of Computer Science, Complutense University of Madrid, in 1993. He is currently a Full Professor of Distributed Systems and Operating Systems with the Computer Science School of Albacete, University of Castilla-La Mancha, Spain. Since 1987, he has been a member of the Computer Science Department, University of Castilla-La Mancha. His current research areas are in the field of concurrency, specifically in formal models for analysis and design of concurrent systems, and real-time systems.

GREGORIO DÍAZ received the Ph.D. degree in 2006. He was an Assistant Professor for several years with the same university. Since 2009, he has been an Associate Professor of Computer Science with the University of Castilla-La Mancha, obtaining the tenure distinction in 2011. His research goals are aimed to make software more reliable, more secure, and easier to design. His primary technical interests include software engineering and related areas, including contract specification, program monitoring, testing, and verification. His research combines strong theoretical foundations with realistic experimentation in the area of Web services and cloud computing.

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 focuses on the integration of complex event processing in event-driven service-oriented architectures, the 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.

GUADALUPE ORTIZ received the Ph.D. degree in computer science from the University of Extremadura, Spain, in 2007. From 2001 to 2009, she was an Assistant Professor and a Research Engineer with the Computer Science Department, University of Extremadura. In 2009, she joined the Department of Computer Science and Engineering, University of Cádiz, as a Professor. She has authored numerous peer-reviewed papers in international journals, workshops, and conferences. Her research interests embrace aspect-oriented techniques as a way to improve Web service development, with an emphasis on model-driven extra-functional properties and quality of service, and service context-awareness and their adaptation to mobile devices. Her research focuses on trending topics, such as the complex event processing integration in service-oriented architectures. She has been a member of various program and organization committees of scientific workshops and conferences over the last years and acts as a Reviewer for several journals.