MEdit4CEP-CPN: An approach for complex event processing modeling by prioritized colored petri nets

Juan Boubeta-Puig a,*, Gregorio Díaz b, Hermenegilda Macià b, Valentín Valero b, Guadalupe Ortiz a

a Department of Computer Science and Engineering, University of Cádiz, Avda. de la Universidad de Cádiz 10, 11519 Puerto Real, Cádiz, Spain
b School of Computer Science, University of Castilla-La Mancha, Campus Universitario s/n, 02071 Albacete, Spain

A R T I C L E  I N F O

Article history:
Received 3 January 2017
Received in revised form 6 November 2017
Accepted 21 November 2017
Available online 22 November 2017
Recommended by M. Weidlich

Keywords:
Formal modeling
Petri nets
Event-based system
Complex event processing
Event processing language
Model-driven engineering
Business process management

A B S T R A C T

Complex Event Processing (CEP) is an event-based technology that allows us to process and correlate large data streams in order to promptly detect meaningful events or situations and respond to them appropriately. CEP implementations rely on the so-called Event Processing Languages (EPLs), which are used to implement the specific event types and event patterns to be detected for a particular application domain. To spare domain experts this implementation, the MEdit4CEP approach provides them with a graphical modeling editor for CEP domain, event pattern and action definition. From these graphical models, the editor automatically generates a corresponding Esper EPL code. Nevertheless, the generated code is syntactically but not semantically validated. To address this problem, MEdit4CEP is extended in this paper by Prioritized Colored Petri Net (PCPN) formalism, resulting in the MEdit4CEP-CPN approach. This approach provides both a novel PCPN domain-specific modeling language and a graphical editor. By using model transformations, event pattern models can be automatically transformed into PCPN models, and then into the corresponding PCPN code executable by CPN Tools. In addition, by using PCPNs we can compare the expected output with the actual output and can even conduct a quantitative analysis of the scenarios of interest. To illustrate our approach, we have conducted an air quality level detection case study and we show how this novel approach facilitates the modeling, simulation, analysis and semantic validation of complex event-based systems.

© 2017 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

Complex Event Processing (CEP) [1] provides users with facilities for analyzing and correlating large volumes of data in the form of events enabling them to detect relevant or critical situations for a particular domain in real time. To meet this objective, the conditions describing the situations of interest must be specified as event patterns. Patterns are implemented by using the languages provided by CEP engines, the so-called Event Processing Languages (EPLs), and once the patterns are defined they can be deployed in the CEP engine in question.

One of the main advantages of CEP is the diversity of scenarios and domains which can benefit from its use. However, there is a common handicap: event patterns should be defined by domain experts, but most domain experts are not proficient in programming languages and programmers are not skilled in programming the patterns for unknown domains. To solve this problem, we successfully proposed MEdit4CEP [2], a model-driven solution for real-time decision making in Event-Driven Service-Oriented Architecture (SOA 2.0) [3]. This solution consists of a graphical modeling editor for CEP domain definition and a graphical modeling editor for event pattern and action definition. Code is then automatically generated from these graphical models. In addition to automatic code generation, MEdit4CEP offers two key features: on the one hand, domain experts can graphically model domains and patterns; on the other hand, the event pattern editor can automatically be reconfigured for different CEP domains, which will have been previously modeled by domain experts.

Therefore, domain experts have the support of a graphical tool to define event types and event patterns, avoiding the need to learn any programming language. The EPL pattern code is then automatically generated from the graphical models and deployed in the subsequent CEP engine. MEdit4CEP provides a syntactic validation of the patterns designed, so we can ensure that the generated code is syntactically correct. However, MEdit4CEP does not provide semantic validation.

In order to provide a semantic validation, we worked on the transformation of CEP operators into Prioritized Colored Petri Nets (PCPN) [4], generating the transformations from a number of Esper...
EPL operators [5] to PCPN, which were validated through a case study.

In this paper, we go one step further by proposing MEdit4CEP-CPN, an extension of MEdit4CEP by means of the PCPN formalism, which provides a Domain-Specific Modeling Language (DSL) and a graphical editor for automatically transforming event pattern models into PCPN graphical models. These models are then validated and transformed into code executable by Petri Nets software.

Our DSL is composed of a PCPN metamodel and validation rules, which can be used to edit and syntactically validate PCPN tool-independent models. The DSL also provides the transformation rules required to automatically transform a model conforming to Model4CEP [6] (our event pattern metamodel for defining patterns in a user-friendly way) into a model conforming to the proposed PCPN metamodel. This transformation to PCPN is based on the mapping defined in the previous work [4], with improvements in order to cover the concatenation of patterns.

In addition, the DSL provides a set of model-to-text transformation rules so that the models conforming to the PCPN metamodel are now automatically transformed into input code for CPN Tools. We can thus utilize all the CPN Tools functions to analyze the PCPN obtained. Specifically, we can obtain results from simulations in order to check the expected pattern behavior, but we can also perform a quantitative analysis of the PCPN so as to generate predictions in hypothetical scenarios.

Thus, the mapping of EPL to PCPN, which is a formalism often used in Business Process Management (BPM) analysis [7,8], provides us with a unified framework for the analysis of event-driven business processes by integrating Event-Based Systems (EBSs), CEP and process control-flow operations in an all-in-one solution. On the one hand, the integration of a CEP engine with an EBS, characterized by connecting event producers and event consumers through a message-oriented publication/subscription middleware, enables us to correlate and aggregate events in order to discover and respond to event patterns. On the other hand, combining existing techniques for CEP and BPM allows us to both monitor the execution of business processes and analyze finished process instances [9]. In this way, the EBSs are suitable for managing events created by processes.

The contributions of this paper are, therefore: (1) a DSL for defining PCPN models and validating and transforming them into code executable by software supporting Petri Nets; (2) a graphical editor for this DSL which allows the user to graphically model the PCPNs and to proceed with all the transformation and validation steps through the editor; and (3) the integration of this editor with MEdit4CEP. As a result, end users can now execute the following steps using MEdit4CEP-CPN: (1) they can graphically model CEP patterns for the domain in question; (2) they can validate the pattern syntax; (3) they can then automatically transform the graphical patterns into a PCPN model, which can also be edited by using our graphical editor; (4) they can validate the PCPN model syntax; (5) subsequently, they can automatically generate code for Petri Nets software; (6) they can validate the pattern semantics and address quantitative analysis; and (7) they can finally automatically generate the EPL code and deploy it in the final system architecture. Until now, only steps (1), (2) and (7) were possible; with the contributions of this paper we achieve steps (3), (4), (5) and (6), which have been properly integrated with the existing functionalities.

Our approach has been validated through a case study for air pollutant monitoring. This case study exemplifies how the event patterns for detecting air quality changes are (1) graphically modeled and syntactically validated, (2) automatically converted into PCPN and (3) semantically validated and quantitatively analyzed.

The rest of the paper is organized as follows. Section 2 explains the required background on CEP, MEdit4CEP and PCPN to facilitate the understanding of this paper. Section 3 presents the model-driven approach in a nutshell, from the graphical definition of patterns to the generation and analysis of the corresponding PCPNs by using CPN Tools, and the generation of their EPL implementation code. The PCPN DSL and graphical editor are then presented in Section 4. Section 5 describes the PCPN modeling for the CEP operators required in the Air Quality Level (AQL) case study, which is then presented and analyzed in Section 6. Relevant related works are detailed in Section 7 and, finally, Section 8 draws some conclusions and future research lines.

2. Background

This section first provides a background explanation on CEP technology. Then, MEdit4CEP is described. Finally, the PCPN formalism we use for modeling CEP-based systems is explained.

2.1. Complex event processing

CEP [1,10] is a technology which emerged recently with the aim of detecting relevant or critical situations (complex events) in real time. It is used to analyze and correlate huge amounts of data in the form of events. A situation is an event occurrence or an event sequence that requires an immediate reaction [10]. Within this scope, events can be classified into two main categories: simple events are those which are indivisible and happen at one point in time, while complex events provide additional semantic significance, which summarizes a set of other previous events [11]. In fact, events can be derived from other events by matching the so-called event patterns, that is, templates which describe the conditions required to deal with a situation. These patterns are implemented by using EPLs, which 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 [12].

In this context, a software capable of matching these patterns over continuous and heterogeneous event streams is required. This software is a CEP engine, which is also in charge of raising real-time alerts when a pattern is met.

Remarkably, we have chosen to use Esper engine, and therefore Esper EPL, for a number of reasons. First, Esper is a highly efficient open-source CEP engine: it can process over 500,000 events/s [13]. Second, Esper EPL is a rich high-level processing language which is more complete than other EPL languages. Furthermore it supplies a wider range of temporal and pattern operators for the definition of situations of interest. For the sake of brevity, we refer to Esper EPL simply as EPL throughout the rest of the paper. Moreover, Esper provides the Esper EPL online tool [14] for simulating events as well as implementing and detecting event patterns. CEP is performed in 3 stages, as depicted in Fig. 1:

1. Event capture — this consists of receiving events which will later be analyzed using CEP technology.
2. Analysis — this stage involves processing and correlating the information in the form of events according to the previously defined patterns in order to detect critical or relevant situations in real time.
3. Response — this refers to notifying the system, software or device in question when detecting a particular situation of interest.

To conclude, the main advantage of using CEP technology is that relevant or critical situations can be identified and reported in real time. Thus, latency in decision-making is reduced in comparison to traditional software methods for event analysis. Furthermore, CEP presents additional benefits such as decision quality improvement, faster and (semi-)automatic reply, information overload prevention and human workload reduction [15].
2.2. **MEdit4CEP**

MEdit4CEP [2] is a model-driven solution aiming to provide support for real-time decision making in SOA 2.0. Its main aim is to facilitate domain experts in defining situations of interest to be detected, as well as real-time notification alerts. The main advantage of using MEdit4CEP for the definition of event patterns is that it completely conceals implementation details from domain experts.

More specifically, this solution is composed of (1) a model-driven approach for CEP in SOA 2.0, (2) a graphical modeling editor for CEP domain definition and a graphical modeling editor for event pattern and action definition, (3) automatic EPL code generation.

Table A.9 in the Appendix summarizes all the language concepts supported by MEdit4CEP for the graphical and user-friendly definition of event patterns. It is worth noting that we have extended the MEdit4CEP supported operators by including the **GroupBy** operator (see Fig. A.26), which allows us to group events depending on one or more event property names. This operator has been added to MEdit4CEP’s tool palette.

The key feature of the event pattern editor is its ability to reconfigure itself for different CEP domains if necessary. These CEP domains should be modeled by domain experts. Subsequently, the editor dynamically reconfigures the tool palette – Simple Events and Complex Events categories, see Table A.9 – accordingly. Thus, the user is provided with a graphical interface adapted to the particular context required.

Since the event pattern models are automatically transformed into a particular EPL making use of Model-Driven Development (MDD) techniques [16], MEdit4CEP provides the following benefits: EPL technical aspects are hidden from end users and productivity is improved since models are easier to maintain than code is. Furthermore, the automatic generated code is free of syntactic errors.

**Example 1.** Let us assume a set of sensor stations that take a Carbon Monoxide (CO) reading each minute, and send the data to a central monitoring system. We need to be warned when the following 4 event patterns are detected by the Esper CEP engine:

- **COGreater**: It detects the first CO reading greater than 0.03 ppm.
- **COEveryGreater**: It detects all the CO readings greater than 0.028 ppm.
- **COAggregation**: It computes in 3-minute sliding windows the aggregate values (min, max, count, sum and average) for the CO readings greater than 0.028 ppm, grouping the information by each station.
- **COMaxBatch**: It computes in 3-minute batch windows the maximum value of the CO readings greater than 0.028 ppm, grouping the information by each station.

Next, we show how to use MEdit4CEP to graphically model these event patterns for this sensor station CEP domain, and, in turn, automatically generate the patterns implementation in Esper EPL.

The CEP domain is modeled with a unique COEvent simple event type, which has two event properties: the name of the sensor station (st) that takes a reading at a specific time and the CO value (co) of the reading taken by this sensor. Fig. 2(a) shows this domain model as obtained using MEdit4CEP. The EPL implementation of this CEP domain model can also be automatically obtained (see Fig. 2(b)).

Once the domain model has been created in MEdit4CEP, the editor palette is automatically adapted and the four event patterns previously introduced can easily be modeled. As an example, Fig. 3(a) shows the modeled COGreater pattern, which creates a complex event called COGreater when it detects the first CO reading greater than 0.03 ppm. This pattern model has been automatically transformed into the EPL code listed in Fig. 3(b).

2.3. **Prioritized colored Petri Nets**

A Petri Net (PN) is a bipartite directed graph, consisting of two types of node, places (circles) and transitions (rectangles). These are connected by arcs that can only link places to transitions and vice versa (see Fig. 4). Places usually represent states or system conditions while transitions are the actions or events that produce changes in the system state. There are some well-known extensions of the simple model, using time and/or data in the model. In Colored PN (CPN) [17,18] places have an associated color set (a data type), which specifies the set of permitted token colors at this place. CPNs are supported by a widely used tool, CPN Tools [19], which allows us to create, edit, simulate and analyze CPNs. The notation described below is the one used in this tool.

Thus, a place may have as a color set, for instance, the set of integer numbers INT, a Cartesian product of two or more color sets as \( INT^2 = INT \times INT \), a singleton color set (UNIT), etc. A place
can then be annotated with a set of tokens (its marking), which are usually depicted by small dots or simply by indicating the number of dots. Each token on a place has an attached data value (color), which belongs to the corresponding place color set. In CPN Tools the current number of tokens on every place is drawn in green in the right-hand side of the place circle, and the specific colors of these tokens are indicated by the notation n v, meaning that we have n instances of color v; the symbol ‘+’ is used to represent the union of colors in CPN Tools. A CPN with (possibly empty) markings on its places is called a Marked Colored Petri Net (MCPN).

Arcs can have inscriptions (arc expressions), constructed using variables, constants, operators and functions. The arc expressions must evaluate to a color or multiset of colors in the color set of the attached place. A transition t is binding enabled if there is a binding so that each input arc expression of t is associated to one or more colors in the corresponding input place of this arc expression. Transitions may also have guards that can restrict their firing, as well as 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. Transitions can also have an associated priority, so marked CPNs with priorities in their transitions will be called Marked Prioritized Colored Petri Nets (MPCPN). In the event of a conflict between two transitions that can be fired (executed) at a given time, the transition with the highest level of priority is fired first. Specifically, we use the following priorities: P_HIGH, P_NORMAL, P_LOW, P_LOW1, P_LOW2, \ldots P_LOWn (for a certain n \in N) and P_MIN, following this decreasing order of priority.

Finally, we use the hierarchical features of CPN Tools, so the PCPNs we produce on CPN Tools consist of several pages, where each page contains a part of the whole PCPN. These pages are connected by using the so-called fusion sets, where a fusion set is a set of places used in different pages that are functionally identical and therefore correspond to the same place from a formal viewpoint.

Example 2. Fig. 4 shows the marked PCPN corresponding to the COGreater pattern depicted in Fig. 3(a). Places COEvent and Out both have CO as color set, which is defined as the Cartesian product INT \times INT \times STRING \times REAL. This captures the information of COEvent events. Places Seqn, SeqOut and Time have INT as color set. The initial marking of COEvent is a set of tokens corresponding to a numbered sequence of COEvent events, where the first integer field indicates the sequence number, the second field indicates the event timestamp and the other two fields represent the event property values (station and CO value). Specifically, we have four tokens in COEvent, representing four events with the readings taken from two sensor stations: st1 and st2 at times 1 and 2. Place Time represents the current model time, so it contains one token whose initial value is 0.

Transitions are labeled with their associated guard and priority information (P_NORMAL if empty). Arcs are also labeled with their corresponding expressions. Variables n, m, t, k, x are integers, st is a string and co is real.

This MPCPN goes through the event sequence in COEvent and writes on Out the first token (event) such that the co property is greater than 0.03. Time starts at time 0, but there is no event at time 0, so only transition tick is initially enabled. This transition has the minimum priority, so it can only fire when no other transition is enabled. Its firing increases the time to 1. The first token in COEvent is 1(1, 1, “st1”, 0.03). As it does not fulfill the condition, transition incr_seq fires to increase the sequence number on the Seqn place, which is then updated to 2. The second token on COEvent is 1(2, 1, “st2”, 0.027), which does not fulfill the condition either, so transition incr_seq is fired again to produce the sequence number 3 on Seqn. The next token on COEvent is 1(3, 2, “st1”, 0.031), but neither incr_seq nor cond transitions can fire, because no binding is possible for variable t (the current time is 1 and the timestamp of this token is 2). Transition tick must then fire again to increase the time to 2, after which transition cond is enabled for the token 1(3, 2, “st1”, 0.031). This transition has the P_HIGH priority, so, once enabled, it must be fired immediately. The binding of variable
3. Our proposal in a nutshell

As explained in the introduction, MEdit4CEP does not provide a semantic validation for the event patterns defined by end users. However, it is clear that the designed event pattern models must be validated before the code generated thereby is deployed into the CEP engine. To address this issue, we extend MEdit4CEP to support the semantic analysis of such event patterns. More specifically, as illustrated in Fig. 5, the objective is to enable the user to proceed with the following phases to define and analyze event patterns by using our MEdit4CEP-CPN solution:

1. **Event pattern model definition**: the end user is responsible for graphically defining the event patterns to be detected in a specific application domain.
2. **Event pattern model syntactic validation**: once an event pattern is modeled, the editor can syntactically validate it, checking whether the model conforms to the Model4CEP metamodel and showing the errors to be fixed before continuing. As of this phase we can accomplish a semantic validation through PCPNs (phases 3, 4, 5 and 6) or we can proceed with phase 7 in order to transform the model into EPL code.
3. **Event pattern model transformation to PCPN model**: the event pattern models are then automatically transformed into a PCPN model (see Section 5). For this purpose, a metamodel for PCPN and a set of model-to-model transformation rules have been proposed (see Section 4.1). Thus, a PCPN conforming to the metamodel is generated.
4. **PCPN model syntactic validation**: once the PCPN model has been automatically generated, expert users may modify the PCPN. For instance, they might edit the initial marking in order to check other relevant scenarios. Thus, after editing the PCPN, a syntactic validation is also performed, checking whether the model conforms to the PCPN metamodel and all validation rules (see Section 4.1.1) are satisfied; the errors to be fixed before continuing are then shown.
5. **PCPN model transformation to PCPN code**: the PCPN model is automatically transformed into executable PCPN code, using the model-to-text transformation rules (see Section 4.1.2).
6. **Semantic validation and quantitative analysis**: the expert responsible for simulating and analyzing the PCPN may feed the net with an arbitrary number of initial markings (stream of events), so as to check whether the net is semantically correct or not, while also performing a quantitative analysis. If a semantic error is discovered, we return to phase 1.
7. **Pattern model transformation to EPL code and deployment**: the event pattern model is automatically transformed into EPL code and deployed in the CEP engine. In this work, we generate code for the Esper CEP engine, but new transformation rules could easily be created to generate code for any Colored Petri Nets software.

---

1. PCPN code refers to a proprietary PCPN file format that can be executed by a specific software.
Thus, we have a top-down approach in which users start by defining what they want to model (event patterns) and the system finishes by providing the implementation code. Remarkably, phases 3 to 6 are the main contributions of this paper.

4. PCPN DSL and graphical editor

In this section, we provide a DSL for defining PCPNs as graphical models, validating and transforming them into code executable by Petri Net software. Moreover, we present our graphical editor that permits to address the execution of phases (3), (4), (5) and (6) described in our approach (see Section 3).

4.1. DSL

We have defined a graphical DSL for PCPN with three parts: (1) the abstract syntax consisting of both a metamodel – a model describing PCPN concepts and the relationships between them – and validation rules to check whether a model is well formed; (2) the concrete syntax, i.e. the set of useful graphical symbols for drawing diagrams; and (3) the model-to-model and model-to-text transformation rules for software automatic generation.

The DSL has been implemented using Eclipse Epsilon [20], a project which works out of the box with Eclipse Modeling Framework (EMF). It provides a family of languages and tools for model validation (Epsilon Validation Language, EVL), model-to-model transformation (Epsilon Transformation Language, ETL), template-based code generation (Epsilon Generation Language, EGL) and graphical model editor creation (EuGENia, which is a front-end for Graphical Modeling Framework (GMF)), among others. Remarkably, ETL is a rule-based model-to-model transformation language capable of implementing many-to-many model transformations, which is a key requirement in our proposal.

4.1.1. PCPN metamodel

Fig. 6 shows the main metaclasses of our PCPN metamodel and the relationships between them, which are described in Table 1, following a top-down and left–right order.

As explained in Section 3, the user can edit the PCPN graphical model generated in phase 3 (event pattern transformation to PCPN model). Thus, a syntactic validation is performed in phase 4 to check the resulting PCPN after these possible modifications. The following validation rules, implemented in EVL, are then applied to check the PCPNs:

- The declaration of a new ColorSet can only use other previously declared color sets.
**Table 1**

<table>
<thead>
<tr>
<th>Metaclass</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PcPan</td>
<td>The main metaclass of the metamodel, so the root of every PCPN is an instance of PcPan, which can contain pages (Page), groups (Group), fusions (Fusion) and a globbox (Globbox). Every PcPan is identified by a unique name.</td>
</tr>
<tr>
<td>Page</td>
<td>A container element where a bipartite directed graph can be modeled. This graph consists of named nodes (Transition and Place), which can be linked by arcs (Arc). Moreover, a page can contain auxiliary elements (Label, Ellipse and Box) without any semantic meaning, but they can ease the readability of the net.</td>
</tr>
<tr>
<td>Transition</td>
<td>A transition net that can contain a guard (Guard) and can have an associated (Priority).</td>
</tr>
<tr>
<td>Place</td>
<td>A place net that can have a named initmark (Initmark). A place has an associated color set (ColorSet).</td>
</tr>
<tr>
<td>Arc</td>
<td>A connection that can only link places (Place) with transitions (Transition) and vice versa. This is annotated with a text inscription.</td>
</tr>
<tr>
<td>Initmark</td>
<td>The initial tokens (Mark) for a Place. It is necessary to specify its number = 1, by default – and value. A mark has an associated color set (ColorSet).</td>
</tr>
<tr>
<td>Mark</td>
<td>A named token for a Place. It must have a name and an associated type (ColorSet).</td>
</tr>
<tr>
<td>Group</td>
<td>A set of named nodes (Transition and Place) on a Page that are members of a group.</td>
</tr>
<tr>
<td>Fusion</td>
<td>A named set of places (Place) so that anything that happens to each place in a set also happens to all the other places in the set. The places are then functionally identical.</td>
</tr>
<tr>
<td>Globbox</td>
<td>A single named element that contains all net declarations. There are different declaration types: Var, Alias, Ml, Block and ColorSet.</td>
</tr>
<tr>
<td>Var</td>
<td>A variable, i.e. an identifier whose value can be changed during the execution of the net. Its type is a color set.</td>
</tr>
<tr>
<td>Alias</td>
<td>Another name for a color set. An alias has exactly the same values and properties as a previously declared color set.</td>
</tr>
<tr>
<td>Ml</td>
<td>An identifier, which is an alphanumeric sequence of letters, digits, apostrophe and underscores starting with a letter. It is useful for declaring priorities (Priority) and constants.</td>
</tr>
<tr>
<td>Block</td>
<td>A container element for grouping declarations.</td>
</tr>
<tr>
<td>ColorSet</td>
<td>A data type (color set) which can be classified into two categories: SimpleColorSet and CompoundColorSet.</td>
</tr>
<tr>
<td>Simple ColorSet</td>
<td>A simple color set: Enumerated and BasicColor.</td>
</tr>
<tr>
<td>Enumerated</td>
<td>A simple color set that is explicitly named as identifier in the declaration. It consists of operands (EnumeratedOperand).</td>
</tr>
<tr>
<td>Enumerated Operand</td>
<td>A string identifier that must be alphanumeric.</td>
</tr>
<tr>
<td>BasicColor</td>
<td>A simple color set whose type can be: Unknown (i.e. the type has not been chosen yet), Boolean, Integer, LargeInteger, Real, String and Unit.</td>
</tr>
<tr>
<td>Compound ColorSet</td>
<td>A compound color set: ListColor, IndexColor, RecordColor, ProductColor and UnionColor.</td>
</tr>
<tr>
<td>List Color</td>
<td>A value sequence (listOperands) whose color set must be the same type. The minimum and maximum length of a list is expressed by lowerBound and upperBound integer attributes.</td>
</tr>
<tr>
<td>Index Color</td>
<td>An index-specifier. The minimum and maximum length of an index is expressed by lowerBound and upperBound integer attributes.</td>
</tr>
<tr>
<td>Record Color</td>
<td>A fixed-length color set whose set of values is identical to the Cartesian product of the values in previously declared color sets. Each of the component color sets (CompoundColorOperand) may be a different type and each is identified by a unique label (operandId), so each field is position-independent.</td>
</tr>
<tr>
<td>Product Color</td>
<td>As RecordColor, this is identical to the Cartesian product of the values in previously declared color sets. The difference is there are no labels to differentiate the component color sets (CompoundColorOperand), so each field is position-dependent, what eases net readability.</td>
</tr>
<tr>
<td>Union Color</td>
<td>A disjoint union of previously declared color sets. Each of the component color sets (CompoundColorOperand) may be a different type and each is identified by a unique label (operandId).</td>
</tr>
</tbody>
</table>

- Every **Place** must have a name and an associated type (ColorSet) that must be defined previously.
- Every **Transition** needs a name and a priority. Non-default priorities must be previously declared.
- Every **Arc** links a **Place** to a transition or vice versa. An **Arc** cannot be used to connect either two places or two transitions, even if they are the same.
- Every **Arc** requires an associated **Inscription**. The variables used in these inscriptions must be previously defined.
- The resulting type of arc inscription must be the same as the type associated with the place from which the arc leaves or arrives.
- There are no isolated transitions, i.e. all transitions have incoming or outgoing arcs.
- **Transition Guards** must be Boolean expressions. Only the variables that appear in the arc inscriptions of the incoming arcs can be used in these Boolean expressions.
- All places of a **Fusion** set must have the same color set.
- For those **Places** having an initial marking expression, the resulting marking evaluation must produce a value consistent with the place color set.

### 4.1.2. Model transformation and code generation

Fig. 7 shows all the model transformations now available in the MEdit4CEP-CPN approach. The new contributions with respect to MEdit4CEP are depicted in the right hand side of the figure: (1) a set of model-to-model transformation rules allowing us to automatically transform graphical event pattern models, which conform to Model4CEP, into graphical PCPN models, and (2) a set of model-to-text transformation rules to automatically transform PCPN models into code executable by CPN software.

In order to conduct the automated mappings from CEPPo- 
main together with EventPattern models to PCPN models, we have proposed the EventPatternToPcpn module, which is implemented in ETL.

Listing 1 illustrates an ETL transformation rule. The following factors should be noted:
Fig. 7. MEdit4CEP–CPN model transformations.

- The rule is applied to transform the CEPDomain into the Page Pcpn, where cpnet is a global variable representing the whole Pcpn output model.
- The rule will only be applied if the existsPage operation is not satisfied.
- The name for the PCPN net is set to the domain name.
- Using the equivalent operator := , the events defined for a Domain are transformed into a PCPN product color by applying the appropriate implemented rule responsible for its conversion.

1 rule CEPDomain2Page
2 
3  
4 for (page in Pcpn.pages)
5  
6 out.print(getPage(page));
7 
8 for (fusion in Pcpn.fusions)
9  
10 out.print(getFusion(fusion));
11 
12 </cpnet>
13 
14 </workspaceElements>

Listing 2: An excerpt of the PcpnToCPNTools transformation using EGL.

As previously mentioned, although we currently generate XML documents supported by CPN Tools, new transformation rules could easily be created to generate code for any Colored Petri Nets software. Thus, thanks to the use of our model-driven approach (see Fig. 7), the end user needs to graphically design each event pattern only once before it can be automatically transformed into any particular Petri Net language, such as CPN Tools XML or Petri Net Markup Language (PNML).

4.2. Graphical editor

By using Epsilon and GMF, we have created a graphical editor for our DSL that permits to address phases (3), (4), (5) and (6) described in our approach (see Section 3). The MEdit4CEP-CPN editor integrates the PCPN editor with the MEdit4CEP editor. Users are thus provided with all functionalities in a single editor. A new option has been added to the Event Pattern menu: Transform All Existing Patterns To PCPN (see Fig. 9). The editor palette shown in the right-hand side of Fig. 10 has 28 tools for modeling PCPNs that are classified into 6 categories: General Elements, Declarations, Page Elements, Place Elements, Transition Elements, and Auxiliary Elements.

Furthermore, a new menu called PCPN has been added with the following options: New (creates a new PCPN from scratch); Duplicate Existing PCPN (facilitates the PCPNs creation from another one); Open (opens one of the previously modeled PCPNs); Save and Validate (saves the active PCPN model and validates it, listing any problems encountered); Transform PCPN To XML (transforms the chosen syntactically validated PCPN model into code); and Delete (eliminates one PCPN). This menu and its options are depicted in Fig. 10, which illustrates the Pcpn model automatically obtained from the pattern model shown in Fig. 9.

5. PCPN modeling

There is no formal semantics for Esper EPL and the only information we have about the operator behavior is the informal description that can be found on the reference site [5]. In this paper, then, we do not define a complete denotational semantics for EPL operators in terms of PCPN, because there is no way to tackle this with a corresponding EPL semantics.

Thus, taking as reference the Air Quality Level Case Study (see Section 6) we provide a PCPN translation for the EPL operators required in this specific case study, whose behavior is validated by confronting the outputs obtained in the PCPN with those produced by the Esper EPL online tool [14]. In a previous work [4], we presented a PCPN translation for a number of EPL operators (event detection, event, followed by and sliding time interval data windows), illustrated by means of a simple healthcare case study. We only provided a translation for the indicated pattern operators isolatedly, i.e. concatenation was not possible, time was only introduced for time data windows and transition priorities were used in a different way. The changes now implemented allow us to link several patterns in a row, so the complex events produced by one pattern can be used as input by the next one. For that purpose, we have made changes to the use of priorities and time control is introduced in all the PCPNs. The PCPNs obtained are compatible with CPN Tools, and can thus be opened immediately. We use the hierarchical features of CPN Tools, so that each pattern is transformed into a PCPN on a separate page (see Table 1).
Fig. 8. Model-to-model and model-to-text transformations.

(a) Pcpn model obtained from CEP Domain. 

(b) Code executed by CPN Tools and obtained from the Pcpn model (a).

Fig. 9. A screenshot of the COGreater pattern modeled with the MEdit4CEP-CPN editor.
Let us now examine the PCPN structure for the EPL operators presented in this paper, taking Example 1 as reference and its corresponding PCPN shown in Example 2. Each type of event (stream) has an associated place, which can be replicated as many times as needed by the pattern operands. The color sets of these event type places are defined taking the event properties defined in their corresponding schemas (CEP Domain), and two additional fields representing an event sequence number and timestamp. Specifically, we have a Cartesian product in which the first component stands for the sequence number (integer), the second component stands for the event timestamp (integer), and the remaining components correspond to the event properties. Thus, the tokens at any event place are annotated with a sequence number and this numbering must be consistent with time elapsing, i.e. for all pairs of events $e_i, e_j$ on an event place with pairs of sequence numbers and timestamps $(i, t_i), (j, t_j)$ we have $i < j \Rightarrow t_i \leq t_j$.

We then annotate 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, especially in certain pattern operators like Batching Time Interval Data Windows. Indeed, we would need this timestamp field for them to keep the initial event time information. Furthermore, we would also need to control time elapsing by using a time place and a tick transition in order to have the data windows at the times required.

Events are then represented by tokens at their corresponding event type places. We assume that new event tokens come into their places as time elapses, i.e. they must be processed according to their sequence number and timestamp fields and the PCPN obtained must be able to process an infinite stream of events (an ordered sequence of events). The complex events generated by the application of event patterns are also represented by specific event type places, with color sets defined according to the pattern schemas that define the specific complex events. These complex event places will have an associated sequence number place with a single integer token that allows us to produce a numbered sequence of events on that place.

**Example 3.** Taking the CEP domain model shown in Fig. 2(a), the MEdit4CEP-CPN editor has automatically transformed it as a PCPN page (EventsPage) containing the COEvent place. Additionally, we have included a specific initial marking, as can be seen in Fig. 11(a). This page has been then automatically transformed into the CPN Tools declaration listed in Fig. 11(b).

Each pattern operator is modeled by a separate PCPN, in which we essentially have a transition representing the pattern application and possible other control transitions and control places that allow us to apply the specific filters the pattern operator requires. Guards and priorities are used in all of these transitions in order to enforce the correct pattern behavior. Conditional expressions in event patterns are translated as transition guards constructed by using the variables declared for event properties. They are written in CPN-ML, a functional programming language based on Standard ML [21]. A global place called Time is used to keep track of the current simulation time and a global transition tick is used to increase the time. This transition has the lowest priority ($P_{\text{MIN}}$), so it only fires when no other actions can be performed. In addition,
time can only increase until the greatest timestamp of the tokens on the input event stream place. Thus, time can be increased again as new events come into this place. This allows us to deal with a possibly infinite and discontinuous stream of events coming into the system dynamically, as expected in this kind of system.

Notice that when two event pattern operators are applied in a sequence, we obtain their associated PCPN separately, but now the output stream of the first pattern acts as the input stream of the second one. Therefore, we only need to join these places into a single one (a fusion place in terms of CPN Tools).

Finally, in the PCPNs presented in this paper, we do not include the specific actions that can be performed as a response to the situations detected, such as sending an e-mail, firing an alarm, etc. These actions can easily be included in the final PCPN by introducing new transitions representing these actions, whose precondition places would be the complex event places related to their execution.

In the following subsections, we present the transformation of the EPL patterns described in Example 1 into both EPL code and PCPN model. We also explain how the latter is transformed into executable PCPN code.

5.2. Event detection

As previously mentioned, the COGreater pattern (see Fig. 9) searches for the first event on an input stream fulfilling a certain condition. The PCPN model for this pattern obtained with MEDit4CEP-CPN is depicted in Fig. 10, before its transformation to CPN Tools. The corresponding CPNP representation in CPN Tools was depicted in Fig. 4 and its behavior was described in Example 2, by using a specific initial marking. In this PCPN, variable st corresponds to property st and variable co to property co. Time elapsing is captured by transition tick, which has the minimum priority, so that time elapsing can only be done when no other transition can be performed. The current model time is indicated by the token on Time. Transition incr_seq is used to increase the sequence number on place Seqn. This transition will be fired several times until finding an event fulfilling the condition, but only as far as we have further events in the place COEvent with a timestamp equal to the current time. Once we have the sequence number (Seqn) for the event fulfilling the condition, due to its high priority, transition cond must be fired, writing the output event in the place Out. Transition cond will not fire again, since it removes the token on Seqn. When there are no further events on the place COEvent with a timestamp equal to the current time, incr_seq is disabled and transition tick must be fired in order to increase the model time.

Table 2 shows the initial marking we have considered on COEvent and the output obtained on place Out.

5.3. Every pattern operator

The every pattern operator selects every event belonging to the specified type that fulfills the indicated condition (if a condition has been defined). Fig. 12 shows the COEveryGreater pattern that returns all the COEvent events that have a CO value greater than 0.028 ppm.

The PCPN for the COEveryGreater pattern is depicted in Fig. 13. This PCPN is similar to that of Fig. 10, but replaces the transition guard with the new one and includes a new arc from the cond transition to the Seqn place, which allows us to continue with the event search. Notice the use of place SeqnOut to number the tokens on Out.

In this PCPN, transition tick fires first to increase the model time to 1. Then, due to its P_HIGH priority, transition cond is fired, yielding the first output token on Out: (1, 2, "st1", 0.03) Once this token has been produced on Out, the sequence number on Seqn is updated to 2. Thus, the next search starts with the second token (event) on COEvent, which does not fulfill the property. However, we have still one token on COEvent with timestamp 1, so incr_seq fires to update the sequence number on Seqn to 3. Transition tick must now fire to increase time, because the third event on COEvent has timestamp 2. As the PCPN evolves, both the third and fourth events fulfill the condition, so both will finally be annotated on Out.

Table 3 shows the initial tokens we have considered on COEvent and the output obtained in Out, with all the events fulfilling the condition.

Finally, if the select clause only chooses a subset of properties from the input stream, we replace the color set of the Out place by the color set of the corresponding complex event type, and the inscriptions on the arc from cond to Out will be changed accordingly.

5.4. Data windows

Bearing in mind the patterns required for the Air Quality Level Detection case study, we restrict our attention to Time Interval Data Windows, presenting both their Sliding and Batching versions.

Fig. 14 shows the COAggregation pattern with a sliding time interval data window, which, in 3-minute time sliding windows, computes the aggregate values (min, max, count, sum and average) for events with a CO value greater than 0.028, grouping the information by each station.
The PCPN for the COAggregation pattern is depicted in Fig. 15. Table 4 contains the COEvent input tokens we have considered and the first five output tokens obtained on place Totals. A screenshot of CPN Tools presenting the complete output is shown in Fig. 16.
In the top part of this PCPN, we can identify the PCPN structure of the every pattern. The tokens produced by the every pattern are written into the DW place (Data Window). This place is intended to contain the tokens (events) that must be processed at the current sliding time data window. Transition start_pdw is used to start the processing of one sliding time data window at the current model time, producing one initial token on Totals for a specific station. Place C2 is used to number the tokens on Totals as they are produced by firing start_pdw. After the firing of start_pdw transition, proc_dw is enabled if we have further tokens on DW for the same station. Notice that due to its P_NORMAL priority, it fires before start_pdw. On each firing of proc_dw, the information on the token in Totals for the current time and station is updated. When there are no more tokens on DW for this station, this token on Totals will contain the aggregate values for that station at the current time. Both transitions start_pdw and proc_dw annotate the consumed tokens on place PDW, which is then a store of the processed tokens at the current time. Finally, on each firing of the tick transition, ret_ev returns the tokens on PDW that must remain in the current slide to DW. For this purpose, place C1 contains the latest firing time of proc_dw.

For instance, for the initial marking indicated in Table 4, after the first event \((1,1,\text{"st1"},0.03)\) is written into DW at time 1, and transition incr_seq fires to update the sequence number on Seqn to 3. Transition start_pdw is then fired to write the initial token for the st1 station and time 1 into the Totals place. The current time \((1)\) is then annotated on C1. In addition, this token \((1,1,\text{"st1"},0.03)\) is written into PDW. No more processing can be done at time 1, so tick fires to increase time to 2. Then, both ret_ev and cond can fire, to return the token from PDW to DW and insert the third and fourth events on DW (cond fires twice), respectively. At this point, we have the three tokens corresponding to the current time slide, \((1,1,\text{"st1"},0.03), (2,2,\text{"st1"},0.031), (3,2,\text{"st2"},0.029)\) on DW, so start_pdw fires and generates one initial token on Totals, either for st1 or st2. Let us assume the binding is made with one of the tokens of st1, in this case proc_dw is enabled and fires to process the other token on DW for st1. The processing continues by firing start_pdw again to produce the initial token for st2 on Totals. Next, transition tick must be fired to increase the time to 3 and a new time slide starts.
The PCPNs for batching time interval data windows are obtained in a similar way. A pattern COMaxBatch is defined in Fig. 17, in which the group-by clause has been specified again to compute the aggregate values for a specific property (station name).

Fig. 18 shows its associated PCPN and Table 5 presents the COEvent input tokens and the output obtained in the Out place. Notice the use of a select transition in the PCPN to provide the maximum co value for each group at every batch window.

A transition \( t_{\text{start}} \) has now been included, which only fires for the first event fulfilling the condition in order to annotate its timestamp into the s_time place. This time value is used as the starting point for the intervals. For instance, for batching time data windows of 3 time units, if the starting time is 2, the first window contains the aggregate values for the events fulfilling the condition with timestamp in the interval [2, 4]. Transitions \( \text{start} \_\text{pdw} \) and proc_dw are used in a similar way as in the case of the sliding time data windows.

Thus, taking the initial marking indicated in Table 5, once the first event is written into DW at time 1, transition \( t_{\text{start}} \) fires to annotate the current time 1 on s_time. Transition tick is then fired to increase time to 2, after which cond fires twice, writing tokens third and fourth on DW. No more processing can be done at time 2, so tick fires again to set the model time to 3. The fifth token does not fulfill the condition, so incr_seq fires, after which cond inserts the sixth token on DW. No more tokens with timestamp 6 exist on COEvent, so the first batching time data window can be constructed. Transition \( \text{start} \_\text{pdw} \) is enabled and must be fired to produce the first token of either st1 or st2 on Totals. Next, proc_dw fires to process the other token for this same station, thus updating the associated token on Totals. The same sequence of firings (\( \text{start} \_\text{pdw} \) and proc_dw) is then performed for the other station to make up the first batching time data window. At this point transition select fires to produce the output token with the properties indicated in the pattern definition. Notice the \( P_{\text{LOW1}} \) priority of select so as to wait until all tokens on DW have been processed before moving them to Out.

Remarkably, this \( P_{\text{LOW1}} \) priority level for transition select should be changed to another \( P_{\text{LOW}} \) level if there is a concatenation of patterns, i.e., the output stream of one pattern is the input stream of another pattern and so on. In this case, in order to ensure that all previous processing due at the current time has been done when a select is performed, we assign decreasing levels of priorities to these select transitions. Thus, for the leftmost PCPN or those running in parallel with it in which we have a select transition, we assign level \( P_{\text{LOW1}} \) to this transition. For another PCPN receiving data from these PCPNs, we would assign level \( P_{\text{LOW2}} \) to transition select and so on.

6. Case study: Air quality level detection

The aim of this section is to provide a real scenario where CEP-based patterns are required. Subsequently, we will show how the pattern models are automatically transformed into their corresponding PCPNs, allowing us to validate their behavior. A short introduction justifying the need for real time air pollution monitoring is first presented. The air quality levels set by the United States Environmental Protection Agency (EPA) [22] are then explained. Next, we present the EPL pattern definition and finally, a data simulation is shown to facilitate the understanding of these patterns.

6.1. Motivation

Air quality is currently a source of great concern across the world, since unhealthy air quality levels can seriously affect our health. The most relevant pollutants are Particulate Matter (\( \text{PM}_{2.5} \) and \( \text{PM}_{10} \)), Carbon Monoxide (CO), Ozone (\( \text{O}_3 \)), Nitrogen Dioxide (\( \text{NO}_2 \)) and Sulfur Dioxide (\( \text{SO}_2 \)). Monitoring these pollutants in real time is key to preventing outbreaks and worsening of respiratory and other related diseases.

There are several indexes for reporting air quality around the world. Each index provides details of air pollution levels by using different scales. However, currently, there is no international standard. For illustration purposes, we decided to base our case study on the air quality levels defined by the EPA, since they provide sufficient information about how each air quality level affects different risk groups. Nevertheless, patterns can be defined for any other index just by adjusting the intervals considered for each level.

6.2. EPA air quality levels

The EPA provides information on the ranges of each pollutant in a particular air quality level. Based on the EPA Technical information, a classification is made calculating the average value of a pollutant across 1 h, 8 h or 24 h, depending on the type of pollutant. For instance, for \( \text{NO}_2 \), the average value in a 1 h period is required, but for CO, the 8-hour period is used. Once we have this average value, we can report the level air quality by taking the range to which the value belongs (see Table 6).

The EPA also defines a general level for air quality, the Air Quality Index (AQI), which is based on the maximum level of each pollutant,
Fig. 18. PCPN for the batching time interval DW with the group-by-clause.

### Table 6

<table>
<thead>
<tr>
<th>Air quality category</th>
<th>Pollutants</th>
<th>NO₂ (ppb)</th>
<th>SO₂ (ppb)</th>
<th>CO (ppm)</th>
<th>O₃ (ppm)</th>
<th>PM₂.₅ (µg/m³)</th>
<th>PM₁₀ (µg/m³)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Level</td>
<td>1 h</td>
<td>1 h</td>
<td>8 h</td>
<td>8 h</td>
<td>24 h</td>
<td>24 h</td>
</tr>
<tr>
<td>Good</td>
<td>1</td>
<td>0–53</td>
<td>0–35</td>
<td>0.0–4.4</td>
<td>0.000–0.054</td>
<td>0.0–12.0</td>
<td>0–54</td>
</tr>
<tr>
<td>Moderate</td>
<td>2</td>
<td>54–100</td>
<td>36–75</td>
<td>4.5–9.4</td>
<td>0.055–0.070</td>
<td>12.1–35.4</td>
<td>55–154</td>
</tr>
<tr>
<td>Unhealthy for Sensitive Groups</td>
<td>3</td>
<td>101–360</td>
<td>76–185</td>
<td>9.5–12.4</td>
<td>0.071–0.085</td>
<td>35.5–55.4</td>
<td>155–254</td>
</tr>
<tr>
<td>Unhealthy</td>
<td>4</td>
<td>361–649</td>
<td>186–304</td>
<td>12.5–15.4</td>
<td>0.086–0.105</td>
<td>55.5–150.4</td>
<td>255–354</td>
</tr>
<tr>
<td>Very Unhealthy</td>
<td>5</td>
<td>650–1249</td>
<td>305–604</td>
<td>15.5–30.4</td>
<td>0.106–0.200</td>
<td>150.5–250.4</td>
<td>355–424</td>
</tr>
<tr>
<td>Hazardous</td>
<td>6</td>
<td>1250–2049</td>
<td>605–1004</td>
<td>30.5–50.4</td>
<td>&gt;0.200</td>
<td>250.5–500.4</td>
<td>425–604</td>
</tr>
</tbody>
</table>

so as to obtain one of six air quality levels: Good, Moderate, Unhealthy for Sensitive Groups, Unhealthy, Very Unhealthy, Hazardous.

For each air pollutant and level, the corresponding health risks are given. For instance, having a NO₂ 1 hour-avg value of 275 ppb and a CO 8 hour-avg value of 8.4 ppm implies the detection of the following air quality levels: Unhealthy for Sensitive Groups for NO₂ and Moderate for CO. Since the Unhealthy for Sensitive Groups level is more dangerous than the Moderate one, the global air quality level (AQI) for this location will be Unhealthy for Sensitive Groups.

### 6.3. Air quality pattern definition

Let us now see the patterns modeled to detect the AQI level at a particular location, where we assume data are received according to the AirMeasurement domain modeled by using the MEdit4CEP-CPN editor (see Fig. 19(a)). This CEP domain model is automatically validated and stored, and can be transformed into EPL syntax (see Fig. 19(b)).

Once the AirMeasurement domain is designed, the event pattern editor is automatically reconfigured for this domain, i.e. the AirMeasurement event type is added as a tool in the Simple Events category of the editor palette. Thus, end users need not to worry about how to define event types together with their properties, since these are graphically represented when dragging and dropping the available tool. Moreover, they cannot modify predefined event types, avoiding the creation of incorrect event patterns for the same domain.

Fig. 20(a) shows the design of a pattern that computes the average value for NO₂ at every location based on the NO₂ measurements received during the last hour. Thus, from all the simple events of AirMeasurement for a location the average value for NO₂ is obtained, and a new complex event with the stationId and the computed average value is created and inserted into the flow NO₂_Avg, so as to have all NO₂ average values obtained across the time. As detailed in Table 6, pollutant values are taken as an average value per 1-hour, 8-hour or 24-hour period. These average values are computed as they are received by using time sliding data windows.

Once the pattern has been modeled, it is automatically validated by executing the syntactic validation rules. The EPL code automatically generated for the NO₂_Avg pattern is shown in Fig. 20(b).

In parallel, we monitor the NO₂_Avg events, to check the level of NO₂. For this purpose, we have defined 6 additional patterns to detect when NO₂ is good, moderate, unhealthy for sensitive groups, unhealthy, very unhealthy and hazardous. For instance, Fig. 21(a) shows the modeled NO₂_Good pattern. It is detected when the average value of NO₂ is greater or equal to 0.0 and smaller than 54.0. In this case, a new complex event with the stationId, the level name (NO₂_Good) and a level number (1 has been assigned for NO₂ good) is created and inserted into the PollutantLevel flow. The EPL code generated for this pattern is shown in Fig. 21(b).

NO₂_Moderate, NO₂_UnhealthyForSensitiveGroups, NO₂_Unhealthy, NO₂_VeryUnhealthy and NO₂_Hazardous patterns are defined analogously according to the intervals for average NO₂ values described in Table 6. The corresponding complex events will
be inserted into the PollutantLevel flow, with levelNumber 2, 3, 4, 5, and 6, respectively.

Regarding the AQI level, the EPA considers the final AQI level as the highest of all the pollutant levels. Thus, if all pollutants except NO2 are at level 2 or less, but NO2 is at level 4, the AQI level will be 4. To detect this situation, we modeled the AirQualityLevel pattern (see Fig. 22(a)), which selects the maximum level detected during 5-minute batching windows for a particular station and establishes this level as the air quality level for the station, inserting it in the AirQualityLevel flow.

6.4. Air quality event simulation

To illustrate the behavior of the patterns when the system is running, we first present a short simulation, considering only two sensor stations (st1 and st2) with NO2 and SO2 values.

Table 7 shows the measurements taken every 5 min for NO2 and SO2 for the 1-hour window from 8:00 to 8:59 h.

As an illustration of the output obtained for these input values in the Esper EPL online tool, we have obtained the following events at 8:55 h:

At: 08:55
-----------------
NO2_Avg=(stationId='st1', value=60.833333333333336)
PollutantLevel=(stationId='st1',
  levelNumber=2, levelName='NO2_Moderate')
SO2_Avg=(stationId='st1', value=10.75)
PollutantLevel=(stationId='st1',
  levelNumber=1, levelName='SO2_Good')

At: 09:00
-----------------
AirQualityLevel=(stationId='st1', level=2)
AirQualityLevel=(stationId='st2', level=4)

Thus, for station st1, the NO2 average is 60.83 ppb, and its level is 2 (Moderate) according to Table 6, while SO2 average is 10.75 ppb, with a level of 1 (Good). For station st2, NO2 average is 363.25 ppb, so its level is 4 (Unhealthy), while SO2 average is 10.08 ppb, with a level of 1 (Good).

At 9:00 h, the maximum level per sensor station is finally obtained:

At: 09:00
-----------------
AirQualityLevel=(stationId='st1', level=2)
AirQualityLevel=(stationId='st2', level=4)

As can be seen from the results, and based on the previous data, the system detects that the air quality level is 2 (Moderate) for station st1 and 4 (Unhealthy) for station st2.
Fig. 20. NO2_Avg pattern.

(a) Pattern model.
(b) Pattern implementation in EPL.

@Name('NO2_Avg')
insert into NO2_Avg
select
  a1.stationId as stationId,
  avg(a1.no2) as value
from pattern
  [(every a1 = AirMeasurement)]
  .win:time(1 hours)
group by a1.stationId

Fig. 21. NO2_Good pattern.

(a) Pattern model.
(b) Pattern implementation in EPL.

@Name('NO2_Good')
insert into PollutantLevel
select a1.stationId as stationId,
  1 as levelNumber,
  'NO2_Good' as levelName
from pattern
  [(every a1 =
    NO2_Avg((a1.value >= 0.0
    and a1.value < 54.0)))]

Fig. 22. AirQualityLevel pattern.

(a) Pattern model.
(b) Pattern implementation in EPL.

@Name('AirQualityLevel')
insert into AirQualityLevel
select a1.stationId as stationId,
  max(a1.levelNumber) as level
from pattern
  [(every a1 = PollutantLevel)].win:
  time_batch(5 minutes)
group by a1.stationId
Table 7
Air pollution simulated data.

<table>
<thead>
<tr>
<th>St.</th>
<th>Pol.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>8:00</td>
</tr>
<tr>
<td>st1</td>
<td>NO₂</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>SO₂</td>
<td>2</td>
</tr>
<tr>
<td>st2</td>
<td>NO₂</td>
<td>171</td>
</tr>
<tr>
<td></td>
<td>SO₂</td>
<td>12</td>
</tr>
</tbody>
</table>

Fig. 23. The obtained AirQuality PCPN model.

By applying both the EventPatternToPcpn transformation and the PcpnToCPNTools transformation (see Section 4.1.2), we have obtained the PCPN model depicted in Fig. 23. This PCPN consists of 15 pages: 2 pages to compute average values of NO₂_avg and SO₂_avg, 6 pages for each pollutant level (PL_NO₂_1 to PL_NO₂_6, and PL_SO₂_1 to PL_SO₂_6), and a final one to determine the air quality (AirQuality). Each time unit in this PCPN corresponds to 5 min. As an illustration, Fig. 24 provides a detailed view of the pattern AirQualityLevel defined in Fig. 22.

The simulation results for the initial marking corresponding to the data shown in Table 7 are partially shown in both Figs. 23 and 24. The output obtained is shown beside the AirQualityLevel_out place on the AirQualityLevel page (see Fig. 24), where the last two elements (sequence numbers 23 and 24) correspond to the output obtained at 9:00h. We can see that stations st1 and st2 have reached levels 2 and 4, respectively, i.e. the same results that we obtained by using the Esper EPL online tool.

Once we have obtained the corresponding PCPN, we can edit it by feeding the net with random values to analyze new hypothetical scenarios. Thus, making use of the MEdit4CEP-CPN editor, we have added some new places and transitions with the aim of randomly generating 1000 input events at the AirMeasurement place, based on the 24 events of Table 7 and considering new input events at each station every 5 min. For simplicity, we have considered that only NO₂ and SO₂ values change, using a normal distribution $\mathcal{N}(0, 10)$ with mean 0.0 and standard deviation 10.0 to produce variations in the previous values for NO₂ and a normal distribution $\mathcal{N}(0, 3)$ for SO₂. Once the PCPN was modified, we checked it by applying the validation rules (phase 4 of our methodology, PCPN model syntactic validation). Subsequently, the PCPN model transformation (phase 5) and semantic validation and quantitative analysis (phase 6) were applied. Fig. 25 shows the last output values obtained in CPN Tools, feeding the PCPN in this way. We can see that the tokens with sequence numbers 999 y 1000 correspond to
The AirQualityLevel page showing the result obtained for the analyzed input.

The last output values obtained for the random input.

<table>
<thead>
<tr>
<th>Station</th>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
<th>Level 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>st1</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>498</td>
</tr>
<tr>
<td>st2</td>
<td>10</td>
<td>7</td>
<td>483</td>
<td>0</td>
</tr>
</tbody>
</table>

time 500, i.e. 41 h and 40 min (2500 min) after the first input event, for which stations st1 and st2 have levels 3 and 4, respectively. Table 8 shows how many times levels 1 to 4 were reached at each station – levels 5 and 6 were not reached in any station.

The results of the simulation show the benefits of MEdit4CEP-CPN. Firstly, it allows us (1) to model a pattern quickly using the edit capabilities and (2) to easily check the pattern syntax. Secondly, it provides pattern-to-CPN automatic transformation, facilitating the PCPN model syntactic inspection. Thirdly, PCPN model edition is facilitated to incorporate the automatic generation of randomly-distributed events. Finally, it can be used in cooperation with CPN Tools to study the pattern semantic behavior by means of simulation.

A video on how the MEdit4CEP-CPN tool works and the simulation results are downloadable at https://data.mendeley.com/datasets/n4cf3x22jj/draft?a=b1023a48-71ed-4822-9f93-a62d983e38fa

7. Related work

Weidlich et al. [23] used Colored Petri Nets (CPN) with Priorities and Time (PTCPNs) to define a model of event processing networks. They present EPN (Event Processing Networks) architecture as the overall system and therefore propose a general translation of this concept, using as illustration an example implemented in the ETALIS framework [24]. In their work, they use time features of CPNs. We, however, preferred to use untimed CPNs with a direct time control, thus avoiding the problems related to time aspects in CPN Tools mentioned in Section 2.3.

Carle et al. [25] also used untimed colored Petri nets to detect critical situations or hazards in the aerospace field. They defined a situation description language called chronicles capturing the detection of simple events, sequence, disjunction conjunction and absence of a chronic operator. A translation of the chronicles language is provided in terms of untimed colored Petri nets with inhibitor arcs and fusion places. They also use an event sequencer to feed the model, but timed events and data windows are not considered, as they only present the situations described by the chronicles language.

Ahmad et al. [26] present a methodology to model CEP using Timed Net Condition Event System (TNCES) [27] and its application to a Manufacturing Line is shown as an example. NCES is a Petri Net derived formalism based on Condition Event Systems, providing a
Fig. A.26. Model4CEP metamodel extended with the GroupBy metaclass.

modular modeling formalism for discrete event dynamic systems. TNCES is the timed extension of NCES based on timed-arc Petri nets [28–30]. Thus, the main difference to our work, other than the Petri Nets formalism used, is that we integrate the PCPN translation into the MEdit4CEP tool, so as to automatically obtain the PCPNs from the event pattern graphical specification created by using this tool.

Hofstede et al. [31] define a formal process model query language for business process queries, which is based on semantic relationships between tasks in process models, independent of any particular process modeling notation. As a future work, they intend to define a technique for APQL queries evaluation by using Petri Nets.

Another formal approach for the modeling of complex event systems was proposed by Hinze and Voisard [32], 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 [33], was defined by Cugola and Margara. TESLA is a highly expressive and flexible language in a rigorous framework, 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 [34] and [35], 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. [36], in which the events and rules specified for CEP applications are analyzed for design errors using the tool REX [37], implemented by Ericsson and Brendtsson. The paradigm used in this case for CEP patterns is the event condition action (ECA) [38]. REX as MEdit4CEP aims to help final users to define the CEP systems. The difference is that REX aims to help inexpert users of formal methods to define the properties that a pattern should satisfy, whereas MEdit4CEP assumes a lack of expertise in the CEP pattern language itself, thus targeting a wider group.

In reference to works considering Petri nets metamodels, Gomez et al. [39] propose a metamodel for modeling Petri nets in the domain of biological data processing; the models conforming to this metamodel can be automatically transformed into the XML code executable by CPN Tools. However, this work has some limitations compared to our solution. First, as stated by the authors, the model design is closer to CPN Tools concepts (e.g. color and position of the graphical elements) than to the conceptual Petri net concepts (see, for instance, the DiagramElement metaclass). Second, Petri net modeling is addressed by using a tree model editor (not a graphical one). In addition, a CPN model can contain only one page, while a PCPN model conforming to our metamodel can contain more than one. Another limitation of Gomez’s metamodel is that it does not allow to model prioritized CPNs and a limited subset of color set types are considered. Even though the latest limitation has been solved with the new version of the metamodel available at https://github.com/abelgomez/cpntools.toolkit, this version retains the other limitations previously mentioned. Petri net modeling is close to CPN Tools concepts and conducted through a tree model editor, and a CPN model can contain only one page.

Westergaard et al. [40] implemented Access/CPN, a framework providing CPN Tools with two interfaces. One is written in Standard Markup Language, which is useful for analysis methods. The other interface is written in Java and provides an object-oriented representation of CPN models, whose object model (metamodel) is implemented by using Eclipse Modeling Framework (EMF). However, the latest version released is not actually up-to-date and, although the latest version available from Subversion (https://svn.win.tue.nl/repos/cpntools/AccessCPN/trunk/) has better support for 4.0 features of CPN Tools, it is still not complete, as stated by Westergaard. In addition, Petri net modeling is addressed by using a tree model editor (not a graphical one with nodes and links), as in the work by Gomez et al. [39].

Kindler created ePNK (http://www.imm.dtu.dk/~ekki/projects/ePNK/index.shtml), a platform for developing Petri net tools based on the PNML transfer format. According to the Petri Net Kernel [41], this platform supports the definition of Petri net types and new plug-in functionalities. Moreover, ePNK provides both a tree editor...
<table>
<thead>
<tr>
<th><strong>Table A.9</strong> MEdit4CEP palette tools.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Category</strong></td>
</tr>
<tr>
<td>Link Value GroupBy</td>
</tr>
<tr>
<td><strong>Simple Events</strong></td>
</tr>
<tr>
<td><strong>Complex Events</strong></td>
</tr>
<tr>
<td><strong>Pattern Timers</strong></td>
</tr>
<tr>
<td><strong>Pattern Operators</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Logical Operators</strong></td>
</tr>
<tr>
<td><strong>Comparison Operators</strong></td>
</tr>
<tr>
<td><strong>Arithmetic Operators</strong></td>
</tr>
<tr>
<td><strong>Aggregation Operators</strong></td>
</tr>
<tr>
<td><strong>Data Windows</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td><strong>Actions</strong></td>
</tr>
</tbody>
</table>
8. Conclusions and future work

In this work, we propose and implement MEdit4CEP-CPN, a MEdit4CEP-based approach extended by a PCPN formalism, which supports the modeling, simulation, analysis and both syntactic and semantic validation of complex event-based systems. Thus, it makes these tasks transparent to domain experts thanks to the use of model-driven techniques.

In particular, our approach provides domain experts with the ability to graphically model the event patterns (situations of interest) to be detected for a particular CEP domain. It validates the pattern syntax, automatically transforms the graphical pattern models into a PCPN model, generates its corresponding PCPN code executable by CPN Tools, validates the pattern semantics, and generates the ESP EPL code to be deployed in the final event-based system. This is achieved by defining a novel DSL and a graphical editor for PCPN, including the necessary model-to-model and model-to-text transformations to be integrated with the existing MEdit4CEP approach.

As future work, we plan to include some additional features in the MEdit4CEP-CPN tool, such as new CEP operators and their support in terms of PCPNs. We also plan to use other hierarchy tools, such as substitution transitions and port-sockets of CPN Tools in order to obtain more well-structured models. In addition, we intend to extend our analysis by producing hypothetical markings in order to automatically test expected outputs versus those obtained by using guided Arrange Act Assert (AAA) strategies. These initial markings could be obtained either from real data scenarios or they could be produced automatically by using certain probability distribution, such as a normal or negative exponential. Our case study uses a normal distribution, but we plan to conduct a deeper analysis of different scenarios by using real data obtained from a set of sensors distributed in several cities in Andalusia, Spain. Moreover, we will be able to produce pollution models from these real data in order to feed the PCPNs by using probability distributions computed from these data. With these probabilistic models, we will be able to perform a deeper analysis and thus obtain predictive results, such as estimated times to detect a specific pattern, the probabilities of detecting situations of interest at specific points of time, occurrence frequency of certain event patterns and average waiting times between particular situations.

Acknowledgments

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 TIN2016-81978-REDT, and in part by the University of Cádiz under Project PR2016-032. Boubeta-Puig would like to thank the Real-Time and Concurrent Systems Research Group for their hospitality when visiting them at the University of Castilla-La Mancha, Spain, where part of this work was developed.

Appendix. MEdit4CEP model-driven solution

See Fig. A.26 and Table A.9.

References


