A Compositional Approach for Complex Event Pattern Modeling and Transformation to Colored Petri Nets with Black Sequencing Transitions

Valentín Valero, Gregorio Díaz, Juan Boubeta-Puig, Hermenegilda Macià, and Enrique Brazález

Abstract—Prioritized Colored Petri Nets (PCPNs) are a well-known extension of plain Petri nets in which transitions can have priorities and the tokens on the places carry data information. In this paper, we propose an extension of the PCPN model with *black sequencing transitions* (BPCPN). This extension allows us to easily model the ordered firing of the same transition using an ordered set of tokens on one of its precondition places. Black sequencing transitions are then presented as a shorthand notation in order to model the processing of a flow of events, represented by one of their precondition places. We then show how black sequencing transitions can be encoded into PCPNs, and their application to model Complex Event Processing (CEP), defining a compositional approach to translate some of the most relevant event pattern operators. We have developed MEdit4CEP-BPCPN, an extension of the MEdit4CEP tool, to provide tool support for this novel technique, thus allowing end users to easily define event patterns and obtain an automatic translation into BPCPNs. This can, in turn, be transformed into a corresponding PCPN, and then be immediately used in CPN Tools. Finally, a health case study concerning the monitoring of pregnant women is considered to illustrate how the event patterns are created and how the BPCPN and PCPN models are obtained by using the MEdit4CEP-BPCPN tool.

Index Terms—Colored Petri nets, CEP, EPL, compositional modeling, model-driven development

1 Introduction

Complex Event Proessing (CEP) technology [1], [2] is used in many different fields to analyze large volumes of information in the form of streams of events, using them to take decisions or derive conclusions. Areas of application include industry and business [3], sensors and IoT [4], air quality control [5], etc. Event patterns are then defined to detect the situations of interest, such as values read from a set of sensors that exceed a certain threshold, unusual inputs from system users, etc. Event patterns are usually written in the so-called *Event Processing Languages* (EPLs), but domain experts are not usually aware of these technologies and languages, which creates an obstacle for user acceptance of EPLs [6]. Therefore, we developed MEdit4CEP [7], which is a model-driven tool for real-time decision-making. MEdit4CEP allows the user to easily define event patterns using a graphical modeling editor. The patterns thus defined can then be automatically transformed into a corresponding EPL code, using Model-Driven Development (MDD) techniques.

Using this approach, we can obtain EPL code, which is syntactically correct, but domain experts still need to know whether the patterns they define are semantically correct, i.e., whether the outputs obtained upon event pattern detection are correct. We take as reference the Esper EPL implementation,[1] but there is no formal semantics for this language, and the event pattern operator semantics is only described textually in the reference manual. Thus, some formalization is required to check whether the patterns created are correct or not. For that purpose, in a previous paper [8] we presented a first Prioritized Colored Petri Net (PCPN) transformation of EPL patterns, with the goal of using CPN Tools [9] to check the semantic behavior of the patterns defined in an unlimited variety of situations. In this previous work, event patterns such as basic event detection, every, followed-by and data windows were considered, but the transformation was limited to certain specific constructions, so compositionality was not possible. We also developed MEdit4CEP-CPN [10], an extension of MEdit4CEP that allows the user to define event patterns using the MEdit4CEP graphical editor, and then automatically transform them into PCPNs, thus obtaining the input files that can be immediately opened with CPN Tools. MEdit4CEP-CPN was then used in [11] as a computational intelligence model-driven tool for the quantitative analysis of events of interest in the context of a specific application, the Sick Building Syndrome.

In this paper, we present a different approach to obtain a compositional PCPN model for event patterns defined using

- *Valentín Valero, Gregorio Díaz, and Enrique Brazález are with the Department of Computer Science, Albacete Research Institute of Informatics, Universidad de Castilla-La Mancha, 02071 Albacete, Spain. E-mail: {valentin.valero, gregorio.diaz, enrique.brazalez}@uclm.es.*
- *Juan Boubeta-Puig is with the Department of Computer Science and Engineering, University of Cádiz, 11519 Puerto Real, Cádiz, Spain. E-mail: juan.boubeta@uca.es.*
- *Hermenegilda Macià is with the Department of Mathematics and Albacete Research Institute of Informatics, Universidad de Castilla-La Mancha, 02071 Albacete, Spain. E-mail: Hermenegilda.Macia@uclm.es.*

Manuscript received 21 Oct. 2020; revised 5 Mar. 2021; accepted 7 Mar. 2021. Date of publication 11 Mar. 2021; date of current version 18 July 2022. (Corresponding author: Gregorio Diaz.) Recommended for acceptance by M. Whalen. Digital Object Identifier no. 10.1109/TSE.2021.3065584

1. `http://esper.espertech.com`

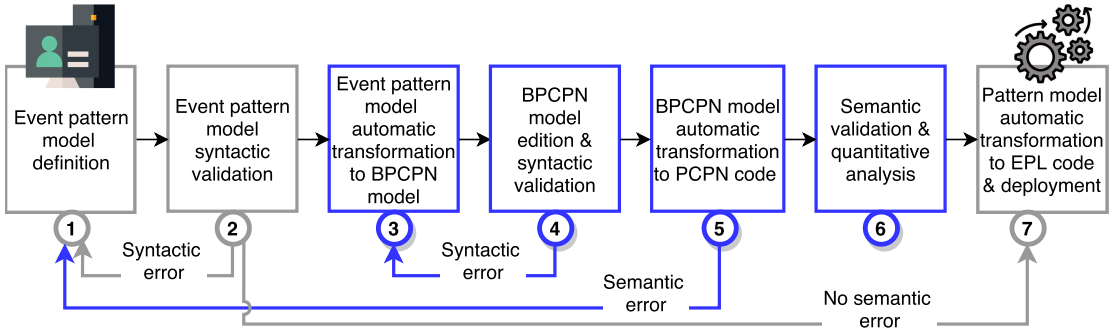

Fig. 1. Phases of the BPCPN approach.

the MEdit4CEP-CPN graphical editor. This new approach is based on an extension of PCPNs with black sequencing transitions (BPCPNs). Black sequencing transitions are a new type of transition that we introduce in PCPNs with the aim of processing an ordered sequence of events from a flow. These black sequencing transitions can be encoded by using normal places and transitions of PCPNs, which allow us to provide compact models for the event pattern operators, simplifying the complexity of the models obtained.

The goal, therefore, is to finally produce automatically the PCPNs corresponding to the specified event patterns, and analyze them by using CPN Tools. Petri nets are a graphical formalism, so with CPN Tools we can easily visualize, edit, simulate and analyze the PCPNs obtained.

MEdit4CEP-CPN has also been modified, to cover the compositional encoding of the patterns operators in a BPCPN intermediate model, and the final enconding of this BPCPN model into a PCPN model that can be used in CPN Tools.

In particular, as illustrated in Fig. 1, our BPCPN approach has the following phases: (1) *Event pattern model definition*: the end user can graphically define the event patterns to be detected in a particular application domain. (2) *Event pattern model syntactic validation*: the editor can syntactically validate the modeled pattern and show the errors to be fixed before continuing. (3) *Event pattern model automatic transformation to BPCPN model*: the event pattern models are then automatically transformed into a BPCPN model. (4) *BPCPN model edition & syntactic validation*: domain experts may edit the obtained BPCPN model, for instance, modifying the initial marking to check other application scenarios. Afterwards, the model is syntactically validated, showing the errors to be fixed before continuing. These errors refer to the structural composition of the BPCPN elements such as places and transitions, specially for the new black transitions introduced in this paper. (5) *BPCPN model automatic transformation to PCPN code*: the generated BPCPN model is automatically transformed into executable PCPN code. (6) *Semantic validation & quantitative analysis*: at this point we can analyze the generated PCPN by feeding it with initial markings (event streams) to check whether the outputs obtained are semantically correct or not. In addition, we can perform a quantitative analysis by feeding the PCPN with random inputs and studying the obtained outputs. In the case that a semantic error is detected, due to a wrong definition of the pattern, we must return to phase 1. (7) *Pattern model automatic transformation to EPL code & deployment*: the event pattern model is automatically transformed into EPL code and deployed in a CEP engine.

In this approach, therefore, end users start by defining the event patterns they need to model and the final output is the implementation code. Note that phases 3 to 6 are the main contributions of this paper which can then be summarized as follows:

- The extension of the PCPN model with black sequencing transitions, as a simple way to process an ordered flow of events (tokens) by the consecutive firings of these transitions.
- The transformation of BPCPN models into equivalent PCPN models, which preserve the net system information and its behavior.
- A BPCPN compositional semantic model for the following EPL operators: simple event search, every and followed-by. These are the basic operators required to detect certain situations of interest, such as temperatures exceeding a certain value or levels of a certain pollutant that exceed the normal values.
- An extension of the MEdit4CEP-CPN tool to provide support for the creation, edition, syntax analysis and automatic transformation of event pattern models into a BPCPN model, and the automatic transformation of BPCPN models into PCPN models executable in CPN Tools.

The extensions to PCPNs with black sequencing transitions are novel, and have demonstrated utility, because they allow a compositional semantics of EPL patterns. It is worth noting the importance of the BPCPN compositional semantics, since we can provide an automatic translation to every pattern constructed using these EPL operators: simple event search, every and followed-by. As we mention in the related work section, there are other works on this subject that provide translations for some EPL operators, but none of them was able to provide a compositional transformation, and they were restricted to the applications of simple patterns constructed using the operators.

The rest of the paper is structured as follows. Section 2 presents a summary of PCPNs and the MEdit4CEP-CPN tool. Black transitions are formally introduced in Section 3. The PCPN translations for the considered EPL operators are then presented in Section 4. The tool support is described in Section 5, together with the transformation validation performed using a set of test cases. A use case is presented in Section 6. Important related works are detailed in Section 7. Finally, Section 8 draws some conclusions and future work lines.

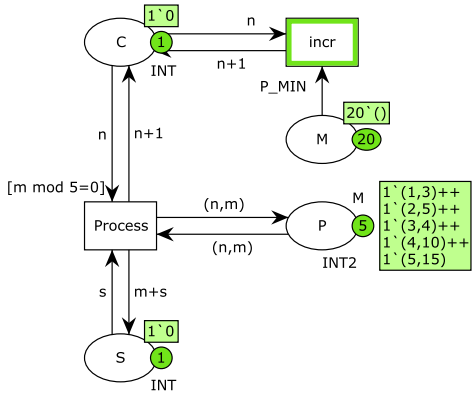

Fig. 2. Graphical view of a marked PCPN.

2 Background

In this section, we present the background to both the PCPN formalism and the MEdit4CEP-CPN tool.

2.1 Prioritized Colored Petri Nets

Petri Nets (PN) [12] are a graphical formalism for concurrent system modeling. PNs are directed graphs with two types of nodes: places and transitions. Places are drawn as circles with a label, and a natural number (number of tokens) inside the circle. This number is the so-called *marking* of the place. Places are then used to represent system states, the number of processes waiting in a queue, for instance, or whether a machine is ready to operate, etc. Transitions are drawn as rectangles with a label, representing events or actions that produce state changes. Arcs can only connect places with transitions (pt-arcs) or transitions with places (tp-arcs), see Fig. 2.

Definition 1 (Petri Net). *A PN is a triple (P, T, A), where P is the set of places, T the set of transitions, such that $P \cap T = \emptyset$, and A is the set of arcs, $A \subseteq (P \times T) \cup (T \times P)$. The set of nodes is $X = P \cup T$. For any node $x \in X$ we define the preconditions and postconditions of x, denoted by ${}^\bullet x$ and $x^\bullet$ respectively, as follows: ${}^\bullet x = \{y \in X \mid (y, x) \in A\}$, $x^\bullet = \{y \in X \mid (x, y) \in A\}$.*

PNs have been extended in many different ways. In particular, in Colored Petri Nets (CPNs) [13], [14] data information attached to tokens is used to extend the simple model. The places of a CPN have an associated *color set*, which is a data type specifying the set of token colors allowed at this place. Among the different tools providing support for CPNs, we use CPN Tools [9], which allows us to easily create CPN models, using a graphical editor. We follow the same notation used in CPN Tools to draw the CPNs, to denote markings, guards, etc. The CPN models created with CPN Tools can be modified, simulated and analyzed by using the simulator engine of CPN Tools and the analysis features, which include boundedness and liveness analysis and state space exploration. The CPN models created with CPN Tools are saved in native XML files, so the files we produce with our transformation are compliant with this format.

A place of a CPN can, for instance, have as its color set the singleton color set $(UNIT)$, which corresponds to the classical tokens of ordinary PNs, without any attached information, but can also have other color sets. For example, the color set INT denotes the set of integer numbers, $REAL$ denotes the set of real numbers, $STRING$ denotes the set of strings, etc. Furthermore, new color sets can be defined as a Cartesian product, a record[2] or the union of existing color sets. For instance, color set $INT2 = \textbf{product}\ INT \times INT$ is defined as the Cartesian product of two integers, whereas $RI2 = \textbf{record}\ f1 : INT * f2 : STRING$ is a record with two fields $f1$ and $f2$, and the color set $UI2 = \textbf{union}\ t1 : INT + t2 : INT2$ is defined as the union of INT and $INT2$ with two possible fields, either *t1* or *t2*. Thus, any token with color set *UI2* must either be an integer or a pair of integers.

In the case of record and union color sets, field labels are used to identify the specific field in which we are operating. For instance, in the case of *RI2*, an element of this color set could be $\{f1 = 3, f2 =" ST"\}$. In the case of *UI2*, $t1(7)$ denotes an integer token with value 7, while $t2(2, 4)$ denotes an *INT2* token with value (2,4).

In CPN Tools, place markings are drawn in green in the right-hand side of the place circle, using the following syntax: $n'v$ denotes n instances of color v, and symbol '++' is used to represent the union of colors in CPN Tools.

Arcs usually have inscriptions, arc expressions, which evaluate to a multiset of colors when we assign values to the variables in the arc expression (*binding*). We say that a binding of $t \in T$ is *enabled* if there are tokens on the precondition places of t matching the expression evaluation. These tokens are then removed when t is fired, and new tokens are produced in $t^\bullet$ according to the arc expressions of the outgoing arcs.

There are two additional important features of CPNs, transition guards and priorities. Guards are Boolean expressions that must evaluate to true with the selected binding for the transition to be fireable. Transitions can also have an associated priority, so we use the term Marked Prioritized Colored Petri Nets (MPCPN) to refer to the marked CPNs with priorities associated with their transitions. In the event of a conflict, the transition with the highest level of priority is fired first. In this paper, we use the following priority levels: P_MAX, P_HN, P_NORMAL, P_MIN, following this decreasing order of priority.

Let us now present the PCPN structure and semantics more rigorously, taking as reference the definitions and CPN semantics described in [13] for the model without priorities, and also the work by van der Aalst *et al.* [15], which includes a description of the use of priorities in CPN Tools.

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*

- *P is a finite set of* places, *where each place $p \in P$ has an associated color set Σ_p. Let Σ be the set of color sets used in N: $\Sigma = \cup_{p \in P} \Sigma_p$.*
- *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 $v \in V$. We can then construct expressions using the multiset operators, constants, variables and functions defined over the color sets in Σ. We use $EXPR_V$ to denote the set of expressions.*

2. These are similar to a product, but their fields are named.

- $G : T \longrightarrow EXPR_V$ *is the* guard function, *which assigns a Boolean expression to each transition, i.e.,* $Type(G(t)) = Bool, \forall t \in T$.
- $E : A \longrightarrow EXPR_V$ *is the* arc expression function, *which assigns an expression to each arc, constructed using variables, constants, operators and functions. Arc expressions must evaluate to the same color set of the place they are attached to, i.e., for every arc* $(p,t) \in A$, *we must have* $Type(E(p,t)) = \Sigma_p$, *and for every arc* $(t,p) \in A$, $Type(E(t,p)) = \Sigma_p$.
- $\pi : T \longrightarrow \mathbb{N}$ *is the* priority function, *which assigns a priority level to each transition, where low values correspond to high priorities.*

Definition 3 (Markings). *Given a PCPN* $N = (P, T, A, V, G, E, \pi)$ *and a place* $p \in P$, *a marking* $M(p)$ *is defined as a multiset of colors in* p *(which can be empty), according to its associated color set. A marking* M *of* N *is then defined taking a marking for all its places. The corresponding marked MPCPN is denoted by* (N, M).

We now address the notion of binding, then the enabling condition and finally the firing rule for MPCPNs.

Definition 4 (Bindings). *Let* $N = (P, T, A, V, G, E, \pi)$ *be a PCPN. A binding of a transition* $t \in T$ *at marking* M *is a function* b *that maps each variable* v *appearing in the inscriptions of one or several precondition and postcondition arcs of* t *into a value* $b(v) \in Type(v)$. *This corresponds to a token color or a part of it that must be present in the precondition place of* t *corresponding to that arc, according to the indicated inscription.*

For instance, for the transition $Process$ *in Fig. 2 and the integer variable* n, *a binding is an integer value that must correspond to one token value on place* C *and the first component of one token value on place* P *at the moment of firing transition* Process.

$B(t)$ *will denote the set of all possible bindings for* $t \in T$. *For an expression* $e \in EXPR_V$, $e\langle b \rangle$ *will denote the evaluation of* e *for 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.

In the following definition we capture the conditions under which $(t, b) \in BE$ can be fired.

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

Hence, the transition must be binding-enabled, its guard must be true, and there is no other transition with greater priority fulfilling these conditions.

Notice that we can have several bindings enabling the same transition, so the firing of a transition is non-deterministic. The new marking obtained after the firing of a transition is computed as follows: for every place $p \in {}^\bullet t$ 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 $E(t,p')$ and the binding selected.

Definition 6 (Firing Rule). *Let* (N, M) *be a MPCPN and an enabled binding element* $(t, b) \in BE$. *With the firing of* (t, b) *we obtain a new marking* M', *defined as follows:* $\forall p \in P : M'(p) = M(p) - E(p,t)\langle b \rangle + E(t,p)\langle b \rangle$, *where symbols* $'+, -$ *respectively denote the union and subtraction of multisets.*

Example 1. Let us consider the MPCPN depicted in Fig. 2. Places C and S have INT as color set, P has $INT2 =$ **product** $INT \times INT$ as color set and M the singleton color set $UNIT$. All of these places have their initial markings indicated in green in their right-hand sides. Both places C and S initially have one integer token with value 0, M has 20 singleton tokens and P has 5 tokens with values $\{(1,3), (2,5), (3,4), (4,10), (5,15)\}$, which could correspond, for instance, to a numbered sequence of integer events, where the first number indicates the ordering and the second the event value. Transitions are labeled with their associated guard (empty when it is true) and priority information (empty if P_NORMAL), and arcs are labeled with their corresponding expressions. All the variables used in the expressions (n, m, s) are integer.

This MPCPN goes through the event sequence in P adding up all the event values divisible by 5. Place C acts as a sequence counter so as to process the event tokens on P in order. Place M in the figure restricts the number of firings of *incr*, since it needs and then removes one token from M when it is fired.

The first transition to be fired is $incr$, because transition *Process* (which has greater priority than *incr*) can only be fired when there is one token in P with a sequence number coinciding with the counter on C, and the value m of that event token in P is divisible by 5. Thus, after the initial firing of $incr$, the new marking of C is $1'1$, and transition $Process$ cannot be fired, since the only possible binding at this marking is $n = 1, m = 3$, which makes its guard be evaluated to false. Transition $incr$ is then fired again, leading us to the marking $1'2$ in C, which allows the firing of transition $Process$ with the binding $n = 2$, $m = 5$, and $s = 0$. Due to its higher priority, transition $Process$ must be fired, replacing the value of the token on S with 5. Transition *Process* also updates the counter in C, by increasing it by one, so that the following event can be processed immediately if the guard condition is fulfilled. Otherwise, transition *incr* is fired (possibly several times) until either reaching a sequence number for which transition *Process* can be fired or M is empty, in which case the MPCPN stops. The final marking obtained in S is therefore $1'30$ for the initial marking indicated in the figure.

In general we will have to deal with large PCPNs, so we split the models into separate pages, in which we draw different parts of the model. Each page contains a subset of the whole model (places, transitions and arcs), and the *glue* of these pages to conform the whole model are the so-called *fusion* places. These fusion places appear in different pages, but they represent the same place from a formal viewpoint. In CPN Tools, all fusion places corresponding to the same place will have the same fusion label, which is drawn in blue at the left-hand side bottom corner of these places. Further details about Coloured Petri Nets and CPN Tools can be found in [16].

2.2 MEdit4CEP-CPN

MEdit4CEP-CPN [10] is our model-driven PCPN extension of the MEdit4CEP approach [7]. This extension allows domain experts to model, simulate, analyze and both syntactically and semantically validate CEP-based systems. Model-driven techniques are used in MEdit4CEP-CPN to facilitate all these tasks and make them transparent to end users.

More specifically, our approach allows domain experts to graphically model the situations of interest (event patterns) to be detected for a specific application domain. It is in charge of validating the pattern syntax, automatically transforming the graphical pattern models into a PCPN model, generating its corresponding PCPN code executable by CPN Tools, validating the pattern semantics, as well as generating the Esper EPL code to be deployed in the Esper CEP engine, one of the well-known open source engines [17]. This was addressed by creating a new Domain-Specific Language (DSL) and a graphical editor for PCPN that includes a set of model-to-model and model-to-text transformations to be integrated with the MEdit4CEP approach. Note that a DSL consists of three parts: (1) the abstract syntax composed of both a metamodel, i.e., a model describing language concepts and relationships between them, and validation rules that check whether the model is well formed, i.e., whether it conforms to its metamodel; (2) the concrete syntax, i.e., the set of graphical symbols useful for creating model diagrams; and (3) a set of transformation rules for automatically transforming models into other models or implementation code.

A complete explanation of the MEdit4CEP-CPN tool is outside the scope of this paper, but more information is available at http://dx.doi.org/10.17632/n4cf3x22jj.2, including the tools, an air quality case study, and a screencast demonstrating the use of the tool.

3 Colored Petri Nets With Black Sequencing Transitions

Black sequencing transitions are now introduced as a shorthand notation for the ordered firing of a transition following an ordered sequence of tokens on one of its precondition places (flow F). A place *init* will then always be present for these black transitions, being used to restrict the starting sequence number for each firing of the transition.

Definition 7 (BPCPNs). *A prioritized colored Petri net with black sequencing transitions (BPCPN) is a PCPN $N = (P, T, A, V, G, E, \pi)$ in which we have two types of transitions: normal (T_n) and black (T_b), $T = T_n \cup T_b$, $T_n \cap T_b = \emptyset$, fulfilling the following conditions:*

- *For every black sequencing transition $t \in T_b$:*
  (a) There are two precondition places $Init_t, F_t \in {}^\bullet t$, with a (product) color set in which the first field is an integer (it may be the only field, in which case the color set would be INT*). We call this number the* token sequence number *and there cannot be two tokens on F_t with the same sequence number. Thus, we will, in general, have a collection of numbered tokens on F_t, starting from 1. In addition, only one token can be taken from these places when t is fired.*
  The other fields in these places, if present, will contain data information related to the application of interest.
  (b) The flow place F_t contains the tokens (events) to be processed by t, and it is also a postcondition place of t: $F_t \in t^\bullet$, $E(F_t, t) \subseteq E(t, F_t)$, i.e., the tokens consumed from F_t are written back into F_t by its firing, but some new tokens can be produced on F_t.
- *There is a place* control, *such that for all $t \in T$ we have that $(control, t) \in A$, $(t, control) \in A$, $E(control, t) = 1'()$, $E(t, control) = 1'()$, i.e., all transitions (including the black ones) are loop-connected with the* control *place. Place* control *initially contains a single UNIT token.*[3] *In principle, we would not need this in a classical occurrence sequence semantics of BPCPNs, since transitions could only fire in a row, but we need this place in the transformation from BPCPNs to PCPNs to avoid the interference of other transitions when a black transition is fired.*

For the sake of brevity, hereon in we will refer to black sequencing transitions simply as black transitions. *They will be drawn in black, and we will usually omit subindex t in places* Init *and* F *when its associated transition is clear from the context.*

Definition 8 (Firing Rule for BPCPNs). *Let $N = (P, T, A, V, G, E, \pi)$ be a BPCPN and M a marking of N. Bindings of normal transitions are defined in exactly the same way as for PCPNs, but bindings of black transitions must satisfy the following condition:*

Let $t \in T$ a black transition and b a binding of t according to Definition 4.

- *Let r, n be respectively the integer values of the first field of the tokens on both places $Init$ and F derived from binding b. Then, we must have $r \leq n$, and no other token on F with a sequence number n' can be bound such that $r \leq n' < n$.*

When this condition holds, the binding element (t,b) is enabled and transition t can be fired. The new marking is obtained as indicated in Definition 6.

Fig. 3 shows a simple BPCPN example, with a black transition t, which is used to process a sequence of tokens on F, returning on place *End* those tokens for which the second (integer) field is divisible by 5, adding 10 (the value of the token on P) to this second field and increasing its first field by 1.

Transformation 1. BPCPNs are encoded into PCPNs by applying the transition substitution illustrated in Fig. 4 for every $t \in T_b$. Transition t is replaced by a new place S and three new transitions: $t1, t2$ and $next$. Transitions $t1$ and *next* have normal priority, and $t2$ has maximum priority. The guard of $t1$ is defined as follows: $G(t1) = G(t)\ andalso\ n \geq r$, where n is the variable used for the first field of the arc from F to t and r is the variable used in the first field of the arc from $Init$ to $t1$. In the same way, the guard of $t2$ is defined as follows: $G(t2) = G(t)\ andalso\ n = r$. The following structural connections are then established: ${}^\bullet t1 = {}^\bullet t$, $t1^\bullet = \{S\} \cup {}^\bullet t \setminus \{control, Init\}$, ${}^\bullet t2 = \{S\} \cup {}^\bullet t \setminus \{Init, control\}$, $t2^\bullet = t^\bullet$, ${}^\bullet next = next^\bullet = \{S\}$.

3. Encoded as INT with value 0 in CPN Tools, due to a problem with the fusion of UNIT places in CPN Tools.

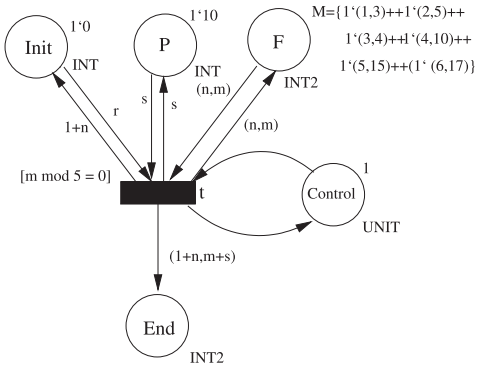

Fig. 3. A simple BPCPN.

Notice that we do not require t to have an outgoing edge to $Init$. In this case, we are restricting the firing of t in the given BPCPN to just one instance per token on $Init$, so if we only have one token on $Init$, t will fire at most once. Thus, the arc from $t2$ to $Init$ would not be included in the translation, as defined by the previous structural connections.

The arc inscriptions for the new arcs[4] are defined as follows, where E_1 refers to the arc expression function in the BPCPN and E_2 refers to the arc expression function of the PCPN obtained as result of the transformation:

- $\forall p \in {}^\bullet t : E_2(p, t1) = E_1(p, t)$.
- $\forall p \in t1^\bullet \setminus \{S\} : E_2(t1, p) = E_1(p, t)$
  Transition $t1$ only checks that t can fire, so it keeps the tokens on p.
- $E_2(t1, S) = E_1(Init, t)$, $E_2(S, next) = i$ (resp. $E_2(S, next) = (i, c)$ when the color set of *Init* is $INT * C$), $E_2(next, S) = 1 + i$ (resp. $E_2(next, S) = (1 + i, c)$), $E_2(S, t2) = E_1(Init, t)$.
- $\forall p \in {}^\bullet t \setminus \{Init, control\} : E_2(p, t2) = E_1(p, t)$.
- $\forall p \in t2^\bullet : E_2(t2, p) = E_1(t, p)$.

The markings of the original places are maintained, and S initially has no tokens. Transition t is now split into two transitions, $t1$ and $t2$, and their guards are extended in order to enforce the firing for the first sequence number fulfilling the guard but greater than the starting number indicated in the selected token on *Init* (see Fig. 4). Transition $t1$ has the guard $n \geq r$ *andalso* g, where g is the guard of t, while $t2$ has the guard $n = r$ *andalso* g. Thus, transition $t1$ can only fire when black transition t can fire in the given BPCPN, i.e., there is at least one token in F fulfilling the condition indicated on the guard of t with a sequence number greater than or equal to r. Furthermore, the priority of $t1$ is exactly the same as for t, so it competes with exactly the same transitions. Transition $t1$ is then only used to check that transition t could be fired in the BPCPN. Once $t1$ has been fired, transition *next* can be fired zero or more times until reaching a sequence number that allows the firing of $t2$, which is then fired, due to its maximum priority.

The place *control* is required to avoid interference with the other transitions in the model. All transitions in the original model must be loop-connected with this control place,

4. Those not connected with UNIT places.

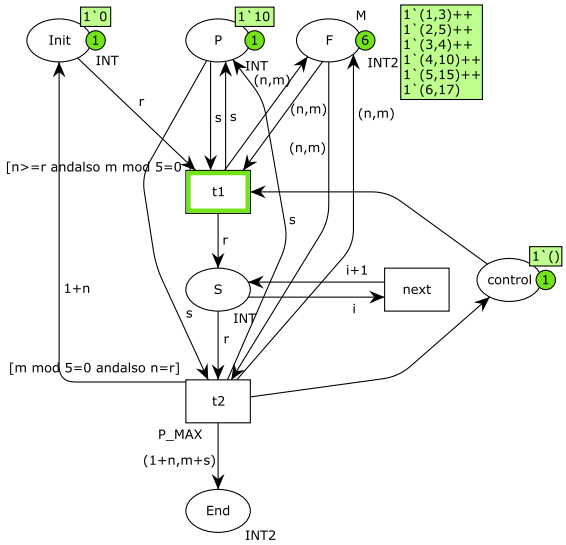

Fig. 4. Black transition PCPN encoding.

and the translation of this specific connection for the black transitions is as indicated in Fig. 4.

This transformation preserves the transition sequences (traces) of the given BPCPN, where the black transitions in the sequence are replaced by the firings of their corresponding transitions $t1$, $next$ (possibly several times) and $t2$, as the following Proposition and Corollary demonstrate.

Proposition 1. *Let (N, M_0) be a marked BPCPN and (N', M'_0) its corresponding marked PCPN, as defined in Transformation 1. Let σ_1, σ_2 be two transition sequences in which there is no black transition and t a black transition in N.*

Then, $\sigma_1.t.\sigma_2$ is a transition occurrence sequence of (N, M_0) if and only if $\sigma_1.t_1.next^{n-r}.t2.\sigma_2$ is a transition occurrence sequence of (N', M'_0), where n is the sequence number of the token used in F for the firing of t and r the value of the token on $Init$ that has been used for this firing.

Proof. $\Rightarrow$: σ_1 can be executed in the PCPN obtained, since we have changed no other transition connections. Let M be the marking obtained in the BPCPN when all transitions in σ_1 have been fired. Transition t can be fired at this marking, so there is at least one token on all its precondition places that make it possible to fire t. Specifically, there is one token on *Init* with a starting sequence number lower than or equal to the sequence number of a token on F. We have the same marking on these places on the corresponding PCPN, so $t1$ can be fired. In fact, its priority is the same as for t in the given BPCPN. The firing of $t1$ removes the token on the place *control*, thus avoiding any other transition firings of transitions in the given BPCPN. The only possible behavior at this point is the firing of either *next* or $t2$. Transition $t2$ has the maximum priority, so it will be fired if enabled, which only occurs when the current sequence number on S is exactly the same as a token on F. Thus, transition *next* will keep firing until this sequence number is reached, which must be the case because the firing of $t1$ was only possible if such a token existed on F. Finally, transition $t2$ returns the token to the place *control* and writes the necessary tokens on the postcondition places of t, so the marking reached on these

places is that obtained with the firing of t in the given BPCPN. From this marking, it is now trivial that the transition sequence σ_2 can be fired on both Petri nets.

$\Leftarrow$: For the converse, we have again that σ_1 can be fired on both Petri nets, and now the firing of $t1$ on the PCPN is only possible if we have tokens on all its precondition places allowing its firing. Thus, there is at least one token on F with a sequence number greater than or equal to the value on one token on $Init$, so t can be fired on the given BPCPN by using the token on $Init$ and the token on F fulfilling the black transition semantics (minimum sequence number such that the guard on t is true). In fact, after $t1$ is fired, transition *next* fires until reaching that same sequence number on S, and then transition $t2$ can fire on the PCPN using the same token on F as t in the given BPCPN. Thus, the resulting marking on its postconditions is the same for both Petri nets. Finally, σ_2 can now be executed on both Petri nets again. □

Corollary 1. *Let (N, M_0) be a marked BPCPN and (N', M'_0) its corresponding marked PCPN, as defined in Transformation 1. We have that σ is a transition occurrence sequence of (N, M_0) if and only if σ' is a transition occurrence sequence of (N', M'_0), where σ' is obtained by replacing every black transition t in σ with its corresponding subsequence $t1.next^{n-r}.t2$.*

Thus, with this transformation we obtain a PCPN whose behavior is the same as that of the given BPCPN. We can therefore analyze the obtained PCPN model using CPN Tools, and the results can immediately be mapped into the original BPCPN. The places of the original BPCPN are maintained in the PCPN, so the state exploration analysis or the simulation results related to markings are mapped trivially. The same occurs for white transitions. In the case of black transitions, properties about them can be analyzed by checking the firings of transitions $t2$ produced by the transformation, since the firing of a black transition t in a given BPCPN can occur if and only if the corresponding $t2$ fires in the obtained PCPN.

The computing complexity is not meaningfully affected by this transformation. The only effect on the complexity would be caused by the consecutive firings of *next* transition (see Fig. 4). Nevertheless, these firings are limited to the tokens (events) that follow to the sequence number indicated in *Init* that do not fulfill the guard condition, until reaching the following sequence number that allows the firing of transition $t2$.

Black transitions can therefore be encoded into PCPN subnets. At first sight, we might think of encoding them by using *substitution transitions*, the other mechanism, together with *fusion places*, which can be used in CPN Tools to construct hierarchical models. With substitution transitions we can create large models by using multiple layers of detail. Thus, starting from a high-level model we can refine the model by replacing some transitions with subnets that preserve their inputs/outputs. However, black transitions do not have the same behavior as normal transitions, so the substitution we apply is not a simple refinement in which the behavior represented by some normal transitions is detailed by their associated subnets. In addition, at present, CPN Tools does not allow the combined use of both fusion places and I/O places, so even the use of the subpages capability is not possible, and thus all the pages we obtain are at the same level.

4 BPCPN Models for CEP

In this section, we present the BPCPN compositional models for the following Esper EPL event operators: *simple event search*, *every* and *followed-by*, with the restriction that an every operator cannot be applied to an inner every operator. These operators are used to deal with the detection of single events, repetitions of events, and streams of events that fulfill certain conditions. In addition, we can create several separate event patterns, so that the complex events produced by some patterns can be used as input for subsequent patterns. The restriction introduced regarding the use of the *every* operator has few consequences, because the use of a nested *every* is uncommon in EPL patterns, and if it were really required we could obtain the same effect with two separate event patterns, where the second pattern takes the events produced by the first one as input.

Each pattern operator is translated in separate BPCPN pages, which are similar to the CPN Tools pages. We also use fusion places to join the pages, with the same interpretation as in CPN Tools.

4.1 Input Event Flow

The event input flow will be modeled by the tokens on a place *Input*, whose color set is defined taking the following fields in a product color set:

- Event sequence number,
- Event type, numerically encoded, according to the event types used in the patterns,
- Event timestamp: instant at which the event occurred,
- Union of all possible color sets for the event types used in the patterns. The property names and data types for each event type color set are obtained from either the schemas that define these specific event types or in the case of complex events these are defined by the select clauses that establish their structure.

In fact, all (complex) event flows follow this same structure: $INT \times INT \times INT \times C$, where the first three integer fields represent, respectively, the sequence number, event type and timestamp.

Timestamps will always be positive, i.e., no event comes at time 0. C is the color set of the associated event type, or a union of them. We assume that event sequence numbers are consecutive as the events enter into a flow and they are consistent with time elapsing, i.e., and for all pairs of events $e_i = (i, t_i, x_i, c_i), e_j = (j, t_j, x_j, c_j)$, we have $i < j \Rightarrow x_i \leq x_j$. It is worth noting that all the events that have the same timestamp have occurred simultaneously, so they could have entered in the flow in any order. Then, we consider equivalent all the possible ways for them to enter into the flow. However, once the events are in the input event flow with their associated sequence numbers, they will be processed according to these numbers.

Example 2. Let us consider the following EPL schemas:

```
create schema A(tm integer, ma integer);
create schema B(tm integer, mb integer);
```

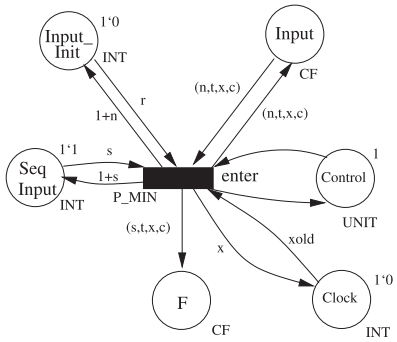

Fig. 5. BPCPN Encoding for Input Event Flow.

The corresponding declarations in CPN Tools for these input event types would be:

```
colset CA=record tm:INT*ma:INT;
colset CB=record tm:INT*mb:INT;
colset C=union pA:CA+pB:CB;
colset CF=product INT*INT*INT*C;
```

The event timestamp is then 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. This is because the clock of the CPN Tools simulator engine does not move steadily, but only moves its value forward when no transition can be fired at the current time. It then jumps forward to the nearest value for which at least one transition becomes enabled. An additional problem we found when using timed PCPNs is that the transition guards are not immediately reevaluated when time moves forward; a transition must fire for transition guards to be reevaluated, which produced deadlocks in the models. The solution required a tick transition to control time elapsing, so we finally preferred an untimed model and so have direct control of the model time.

Events are then represented by tokens on their corresponding places, and new event tokens come into the input event flow as time elapses, so we have a potentially infinite stream of events. Events from the Input event flow place must then be processed in order, so a black transition is first used to feed the flow place F, as shown in Fig. 5. This BPCPN is then encoded in a separate page, and places *Control*, *F* and *Seq_Input* are fusion places, which will be used in the pattern operator BPCPN pages.[5] Place F acts therefore as the event stream used in EPL for the processing of events, so we only insert new events on it when all the events that were previously on it have already been processed. The lowest priority level *P_MIN* is then used in transition *enter* to allow the following event to enter only when no other processing can be done, and the global clock value (integer token on place *Clock*) is updated to the timestamp of this last entered token.

4.2 Pattern Operators

A pattern is defined by the composition of a set of pattern operators and a *select* clause to choose the information to be provided to the user from the events that have been processed. For each pattern operator, a corresponding BPCPN page is defined, so the BPCN for a pattern is obtained in a compositional way, and the link between these BPCPN pages are the fusion places. However, there is a final step in which the BPCPN obtained is modified in order to apply the select clause and include the events produced by the pattern in the flow place F. These newly produced events are inserted in F to allow its further use by other patterns that could be applied later.

5. We omit the fusion labels in BPCPN models, since they are the same as the place names.

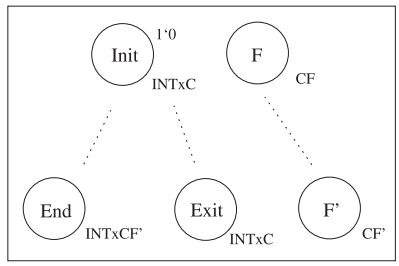

Fig. 6. Event pattern BPCPN structure.

The BPCPNs for the pattern operators have the following structure (Fig. 6):

- They process the events coming into the event flow place F, with color set $CF = INT \times INT \times INT \times C_F$, where C_F is the union of all the event types used by the patterns.
- There is an *Init* place, with a color set of the form INT or $INT \times C$, for a certain flow color set $C = INT \times INT \times INT \times C_0$. We will use the color sets of the form $INT \times C$ in the case of the *followed-by* operator, specifically in the righ-hand side argument, where the C-type events gathered by the left-hand side argument are passed to the second argument.
- There is a place *Exit*, with the same color set as *Init*, which only receives one token when the pattern operator has terminated (if it terminates). The first integer field contains the position at which we should continue processing the event flow in the case of either an *every*or *followed_by* external operator acting on the result of this operator. The second field, if present, contains exactly the same value of the second field of the token used to start the BPCPN on place *Init*.
- The events produced by the pattern operator are written in an output event flow place F', but when *Init* has a color set $INT \times C$, these events are concatenated with the color field of the second component of the token used from *Init*. Thus, the color set of F' is $CF' = INT \times INT \times INT \times C_{F'}$, where $C_{F'} = C_0 \times C'$ (resp. $INT \times INT \times INT \times C'$, when *Init* has color set *INT*), where C' is the color set of the events produced.
- There is a place *End*, with color set $CEnd = INT \times CF'$, which receives the tokens written in F'. The first field of these tokens on place *End* is a sequence number that indicates the current position on F plus one for the event produced. This sequence number is exactly the same as that written in *Exit* when the pattern terminates, but notice that *Exit* is not always

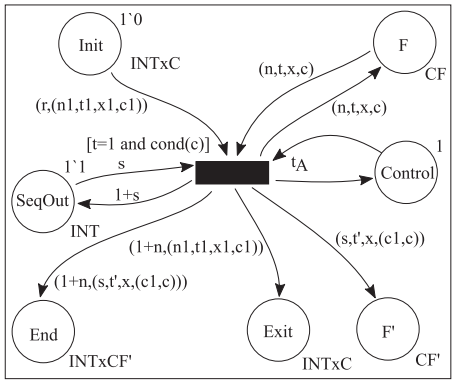

Fig. 7. BPCPN for event detection.

reachable, since some operators do not terminate *(every)*. Place *End* is used by the *followed-by*operator to store the events produced by the left-hand argument.

We denote the patterns by the following abstract syntax:

$$((pattern\_op, Init, F, F', Exit, End), sel),$$

where function sel represents the *select* clause in the Esper EPL syntax, which is only applied at the end of the transformation, taking some fields of its argument events. *Pattern_op* represents the event pattern operator, with the following syntax:

$$\begin{aligned} pattern\_op ::= {} & Event\_Type(cond) \mid \\ & every(pattern\_op) \mid \\ & pattern\_op \rightarrow pattern\_op \end{aligned}$$

where *cond* is a predicate constructed by using the event type properties. *Event_type* denotes an event type (schemas declared or complex event types), so we are searching for the first event of that type fulfilling the indicated condition, after which the pattern operator terminates. The *every* operator repeats the pattern operation indicated as argument, with the restriction that an *every* operator cannot be applied over another inner *every* operator, i.e., patterns such as the following are not permitted:

$$\begin{aligned} & every(every(A(cond)) \\ & every(A(cond) \rightarrow (B(cond1) \rightarrow every(C(cond2)))) \end{aligned}$$

In consequence, an *every* operator never terminates. With the followed-by operator ($\rightarrow$) for every event produced by the left-hand side operator we search for all the subsequent events that match with the right-hand side operator. This operator only terminates when both arguments have terminated.

The translation of a pattern $((pattern\_op, Init, F, F', Exit, End), sel)$ is done compositionally. We first obtain the BPCPN for the input event flow (Fig. 5), and then we obtain the BPCPN for $(pattern\_op, Init, F, F', Exit, End)$, after which the *sel* function is applied by modifying the arc inscriptions leading to the final place F', as well as its color set. In this final step, we also insert the newly produced events in F, so that they can be used in further patterns. Recall that place F is a fusion place, being the same place as in Fig. 5, and the same occurs with places *Control* and *Seq_Input*, the latter is used to insert the newly produced complex events in F in the correct order.

4.2.1 Event Detection

Our base case is $Event\_Type(cond)$, which searches for the first event of the indicated type that fulfills a certain condition on event flow place F. Fig. 7 shows the BPCPN for this search, where we are looking for the first event of type A (type 1) that fulfills the predicate condition *cond* and produces as result an event $(s, t', x, (c1, c))$, where s is the sequence number, obtained from place *SeqOut*, t' is the type of event produced, x is the time of the event processed and $(c1, c)$ is the concatenation of the event coming from *Init* (if *Init* has the color set $INT \times C$, otherwise we would write (s, t', x, c)).

Black transition t_A will fire for the first event on F that has type 1 (A) and fulfills the indicated condition. The position on F after finding such an event would be $1 + n$, so this is the value on the first fields of both *Exit* and *End*.

Example 3. Let us consider again the EPL schemas:

```
create schema A(tm integer, ma integer);
create schema B(tm integer, mb integer);
```

We can have both event types (A, B) in the input event flow, but we only want property *ma* of the first A event such that $ma\%5 = 0$. We then write the following EPL code:

```
@Name('FirstA')
insert into C
select a1.ma as mc
from pattern [a1= A(ma % 5 = 0)];
```

The output event is encoded with event type 3. It is produced at the time of the first detected event fulfilling this condition and will be the first event in the output flow. The color set CF now includes this new complex event type:

```
colset CA=record tm:INT*ma:INT;
colset CB=record tm:INT*mb:INT;
colset CC=record mc:INT;
colset C=union pA:CA+pB:CB+pC:CC;
colset CF=product INT*INT*INT*C;
```

Fig. 8 shows the CPN page that encodes this BPCPN, in which we have also applied the *select* clause to the tokens produced to F', thus changing its type and color set to CF. We have also included the events (tokens) produced by the pattern in F since, as mentioned above, all tokens produced by a pattern must be included in the flow. This only requires including these tokens in the arcs returning to F from the final transitions that produce such tokens. The sequence numbers for these tokens are obtained from fusion place *SeqInput* (Fig. 5).

In this CPN, for the following initial marking on the place *Input* (Fig. 5):

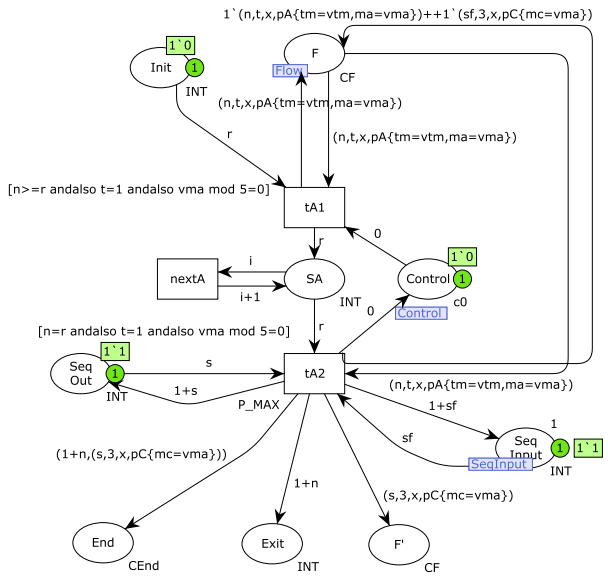

Fig. 8. CPN page of FirstA pattern.

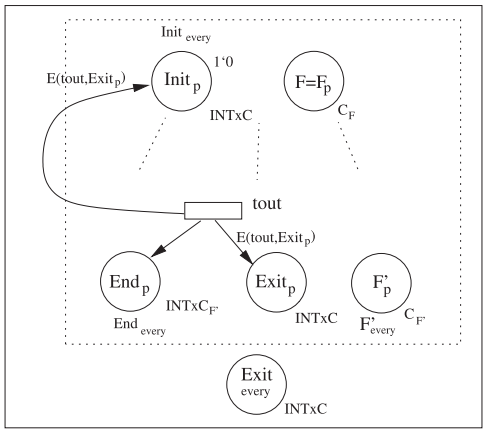

Fig. 9. BPCPN for pattern *every*.

$$M_0 = 1'(1,2,1,pB\{tm=1, mb=10\}) ++ \\ 1'(2,1,1,pA\{tm=1, ma=20\}) ++ \\ 1'(3,2,2,pB\{tm=2, mb=30\}) ++ \\ 1'(4,1,3,pA\{tm=3, ma=0\})$$

we would obtain $1'(1,3,1,pC\{mc=20\})$ as the output token on F'. Evidently, this is also the output that we would obtain by using Esper EPL Online tool for this same pattern and input events.

Regarding the correctness of this translation, notice that the obtained BPCPN fulfills the structural properties indicated at the beginning of this section. The token produced in place *End* has as first field the following position in F to start a new search $1+n$, and the second field is the event produced by the operator. In the same way, the first field of place *Exit* is also $1+n$, and the second field is the second field of the token consumed from *Init*. Place *Exit* is marked as this operator terminates when t_A is fired.

4.2.2 Every operator

Fig. 9 illustrates the BPCPN translation of the every pattern operator: $(every(p), Init_{every}, F, F'_{every}, Exit_{every}, End_{every})$. We

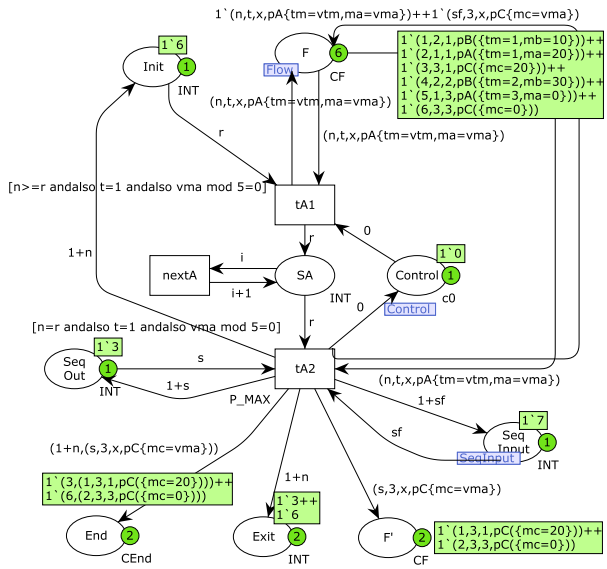

Fig. 10. CPN page for pattern *everyA*.

first compositionally construct the BPCPN for $(p, Init_p, F, F'_p, Exit_p, End_p)$, with $Init_p = Init_{every}, End_p = End_{every}$, $F'_p = F'_{every}$, where no *every* operator can appear in p. Place $Exit_p$ must then have at least one precondition transition (black or normal). Thus, we can include new arcs from these transitions (*tout* in the figure) to $Init_p$, with the same inscription as those from these transitions to $Exit_p$. Thus, the new index in $Init_p$ would be the one written in the first field of the token in $Exit_p$, and the every operator follows the processing from that position in the flow. Notice that $Exit_{every}$ is unreachable (an every operator cannot terminate), but it must be produced as an isolated place for the sake of compositionality.

Example 4. Let us consider the previously defined schemas A and B, and the following Esper EPL pattern to obtain the *ma* values for all A-events for which ma is divisible by 5:

```
@Name('EveryA')
insert into C
select a1.ma as mc
from pattern [every(a1=A(ma % 5 = 0))];
```

Fig. 10 shows one of the CPN Tools pages produced for pattern *EveryA*, which corresponds to the modifications performed in page A to include the arc from *tA2* to *Init*. The *select* clause has also been applied, and thus the arc from *tA2* to F has been modified in order to insert the obtained events (tokens) in F, using fusion place *SeqInput* (Fig. 5) to assign the correct sequence numbers to them. In this figure, we can see the output obtained in F' for the same initial marking of Example 3, where we can see both events fulfilling the condition. These events have also been included in flow F, with sequence numbers 3 and 6.

Regarding the correctness of this translation, observe that place $Exit_{every}$ is unreachable and place End_{every} is the same as End_p, so the tokens on it have the contents obtained by the BPCPN of p. On each execution of a transition *tout*, a token is produced on place $Init_{every}$, which has as first field the new position in F to start a new search, as required by the every pattern.

4.2.3 Followed_by Pattern

Let us now consider the pattern $(p_1 \rightarrow p_2, Init, F, F', Exit, End)$. We first obtain the BPCPNs for $(p_1, Init, F, F'_1, Exit_1, End_1)$ and $(p_2, End_1, F, F', Exit_2, End)$. We take $Init_2 = End_1$ in order to connect both BPCPNs and enforce $BPCPN_2$ to start processing flow F at the following position at which $BPCPN_1$ returns each detected event. Thus, for each event coming from $p1$ we will obtain the events matching with $p2$ that follow it. Fig. 11 illustrates the connection between both BPCPNs, where we can see that a transition *Exit* with priority *P_HN* has been included for termination. This transition can only be executed if both argument BPCPNs have terminated. Thus, when one of the arguments does not terminate (it contains an every operator), the *followed-by* operator does not terminate either. Transition *Exit* has priority *P_HN*, the second level of priority after *P_MAX*,[6] with the goal of terminating the inner *followed-by* pattern operators as soon as they finish and process the events produced in the $Exit_i$ places ($i = 1, 2$) in the order they are produced.

Regarding correctness, as place *End* for the *followed-by* operator is the same as End_2, the contents in this place are obtained as result of the execution of $BPCPN_2$, and are structurally correct by compositionality. In addition, for each token produced on End_1, we can execute $BPCPN_2$ using that token, since $End_1 = Init_2$. Thus, $BPCPN_2$ starts the search from the position indicated in the first field of that token, which corresponds to the semantics of the *followed-by* operator. With respect to place *Exit*, it can only be reached when both *BPCPNs* have terminated, and the token produced on *Exit* is the same as that produced by $BPCPN_2$ upon termination. As regards F', this place is F'_2, so the same events (tokens) produced by $BPCPN_2$ are the events (tokens) produced by the followed-by operator. Recall, however, that from the basic event detection the second field of the tokens consumed from $Init_2$ is concatenated with the events produced by $BPCPN_2$, so the final output produced in F' contains the outputs from both $BPCPN_1$ and $BPCPN_2$.

Observe that the events produced by pattern p_1 in F'_1 are stored in $End_1 = Init_2$ (second field), so they can be used in $BPCPN_2$. Specifically, we can use their fields to check some conditions in p_2, which is also allowed in Esper EPL, as the following example demonstrates.

Example 5. Taking again the two previously defined schemas, we can consider the following EPL pattern, which obtains the first A-type event and then looks for a B such that $mb > 1$, but only if the A-event found fulfills $ma \,\%\, 5 = 0$.

```
@Name('AtoB')
insert into AtoB
select a.ma as ma, b.mb as mb
   from pattern [ a=A -> b = B (ma % 5 = 0 and mb > 1)];
```

Fig. 12 shows the final BPCPN for this pattern, once the select function has been applied and the newly produced tokens have been inserted in F. We can see that place End_1 is the $Init_2$ place for the BPCPN corresponding to B. In the same way, End_2 is the *End* place of the full pattern. Hence, value $1 + n$ in the first field of the inscription of the arc from t_A to End_1 is the index used in $Init_2 = End_1$ to start the search for one B-type event fulfilling the indicating condition. The second field of this arc expression is the event produced $(s, 3, x, (tm, ma))$. In the guard condition of t_B, we use the field ma of the A-type event previously found, so this pattern will not terminate if the found A event does not satisfy this condition, even if there are B events with $mb > 1$. The event produced by the B-type event detection operator is $(s, 4, xb, ((tma,\ ma), (tmb, mb)))$, which is then modified by the select function as follows: $(s, 4, xb, (ma, mb))$.

The corresponding PCPN can now be automatically obtained by applying the black transition encoding.

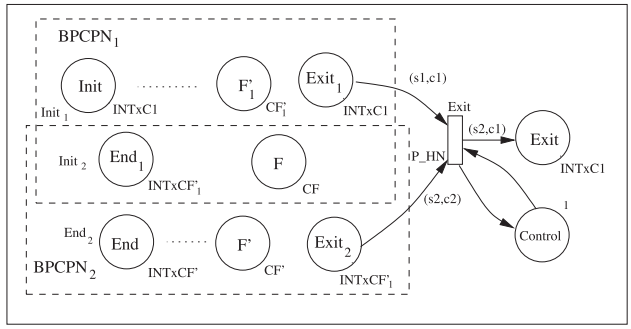

Fig. 11. BPCPN for pattern *Followed_by*.

An interesting observation regarding the general use of the followed_by pattern is that each time a new event enters into F (input event flow) the followed-by pattern may produce none, one or several events as a result of its application, due to compositionality. For instance, we can have several previous tokens on End_1, which would precede a new matching event on the second argument, thus producing some new events.

In the examples presented we have shown the BPCPNs for the pattern operators, which are then transformed into PCPNs using the encoding presented in Section 3. In Section 5.3 we illustrate the complete methodological process by using a set of test cases, and a real-world use case is presented in Section 6.

Regarding the scalability, we have two issues to consider, the model scalability and the state space scalability. With regards to the model scalability, we will have in general a small number of event operators in the pattern definitions, and we typically have patterns that combine one or two followed-by operators with some every operators. The transformation will produce separate PCPN pages for each operator, and the MEdit4CEP-BPCPN tool does not have a limitation in this aspect, and to our knowledge, CPN Tools has no limitation in the number of pages, either. Regarding the state space scalability, the analysis of PCPNs is usually done by simulations, because of the state space explosion. In our case, let us observe that the state space explosion appears due to the events that occur at the same time and thus can be processed in different order. As a consequence, in the state space exploration we obtain many different branches that lead to different markings in the output event flow, which would correspond to the different ways in which these events could have been processed. In practice, however, a CEP engine would only provide us with a single output, obtained for a specific ordering of events, so with a

6. *P_MAX* is only used to encode black transitions.

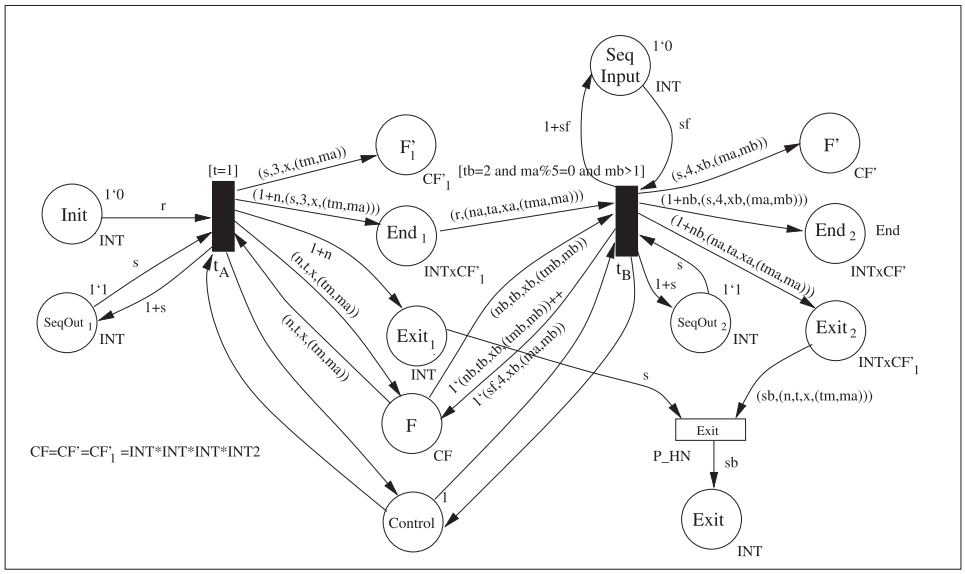

Fig. 12. BPCPN for pattern AtoB.

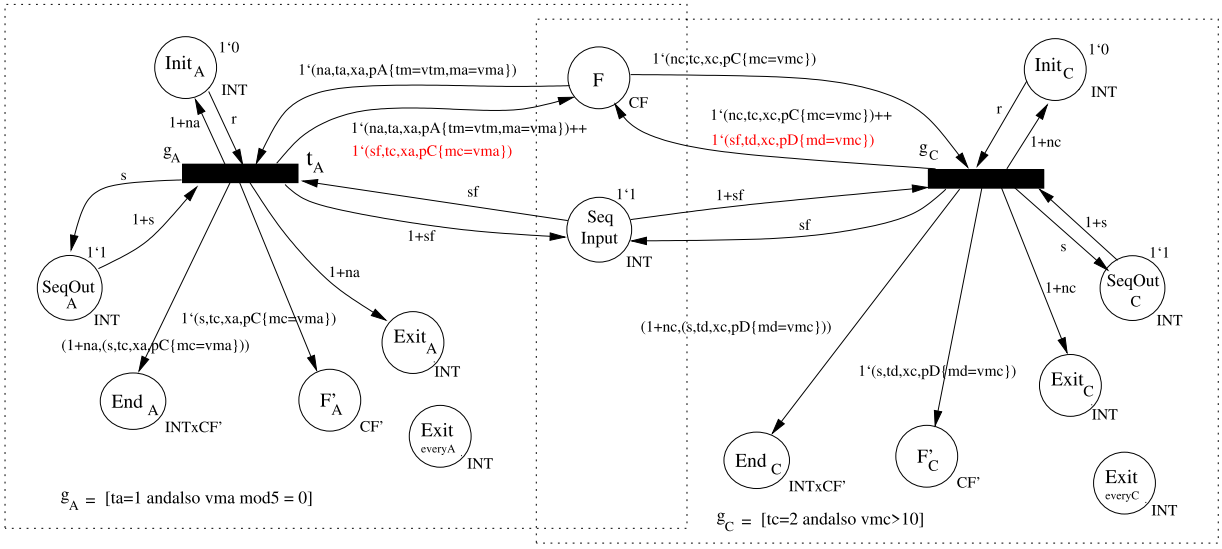

Fig. 13. Pattern composition: EveryC applied on the results of EveryA.

single simulation of the BPCPN model we can quickly and easily obtain the same result.

4.3 Composing Patterns

We now consider the case of a pattern chain in which the complex events produced by previous patterns can be used by a subsequent pattern in addition to the events from the initial input stream.

For that purpose, as previously mentioned, the events produced by a pattern are inserted into flow place F. Let us remember that new events from the place *Input* can only enter F when all other processings have finished.[7] Thus, the newly inserted events on F keep the consistency between sequence numbers and time, since we use the same place *SeqInput* to assign the sequence numbers for these complex events.

The BPCPNs of these patterns are obtained separately, but use the same input flow place F, so events are separately processed from this flow for each pattern. However, the important thing is that events produced by one pattern can be used by further subsequent patterns.

Example 6. Let us consider the *EveryA* pattern presented in Example 4, which produces complex events of type C, which contain a single property *mc*, and consider now the *EveryC* pattern, defined as follows:

```
@Name('EveryC')
insert into D
select c1.mc as md
from pattern [every(c1= C(mc > 10))];
```

This pattern takes the C-events for which *mc* is greater than 10, so it is applied over the results produced by the *everyA* pattern.

Fig. 13 illustrates this pattern composition, showing in dotted boxes the BPCPN pages produced for the A and C event detections in both patterns. In the left-hand side

7. Due to the minimum priority of transition *enter* in Fig. 5.

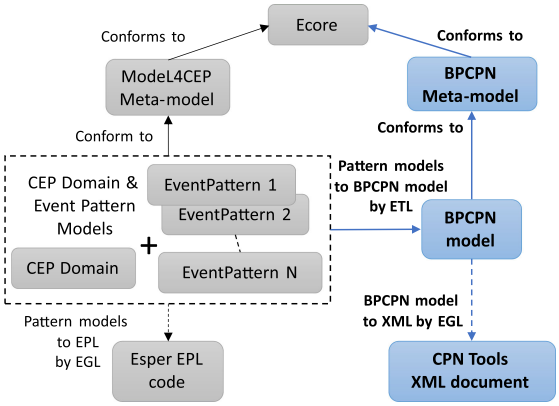

Fig. 14. MEdit4CEP-BPCPN model transformations.

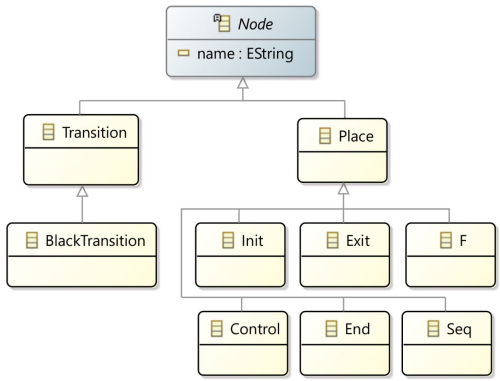

Fig. 15. New entities added to our PCPN metamodel.

of this figure we have highlighted in red the C-events produced by pattern *EveryA* that are inserted in the flow F. These events are then processed by pattern *EveryC* (right-hand side), which produces D-events, which are also inserted in F.

More information about this example can be found in the *Pattern Composition* folder of the Mendeley dataset: http://dx.doi.org/10.17632/scrjyndpv6.1. A *Screenshots* folder is also provided, which contains a pdf file with the BPCPN obtained.

5 MEdit4CEP-BPCPN

In this section we present the MEdit4CEP-BPCPN tool, which has been developed as an extension of MEdit4CEP-CPN (see Section 2.2) to provide support for the creation, edition, syntax analysis and automatic transformation of event pattern models into a BPCPN model and the transformation of the obtained BPCPN model into the corresponding XML code to be executed in CPN Tools, i.e., in a PCPN model that we can analyze and simulate in CPN Tools.

The technology used in this development is Eclipse Epsilon [18], which consists of a family of languages and tools for model-to-model transformation (Epsilon Transformation Language, ETL), model validation (Epsilon Validation Language, EVL), template-based code generation (Epsilon Generation Language, EGL) and graphical modeling editor creation (EuGENia). EuGENia is a front-end for Graphical Modeling Framework (GMF) that automatically generates the models needed to develop a GMF editor from an annotated Ecore Eclipse Modeling Framework (EMF) metamodel.

Fig. 14 illustrates both the specified graphical models and the transformation process now available in the MEdit4CEP-BPCPN approach. In the right-hand side of the figure we can observe the new contributions with respect to MEdit4CEP-CPN. The transformation process consists of two steps: (1) a CEP domain and several event pattern models conforming to the ModeL4CEP metamodel [19] are transformed into a BPCPN model conforming to our BPCPN metamodel by applying a set of ETL model-to-model transformation rules (M2M), and (2) the obtained BPCPN model is transformed into a CPN Tools XML document by applying a set of EGL model-to-text transformation rules (M2T). This document can then be executed by the CPN Tools software.

5.1 BPCPN Metamodel and Validation Rules

In this subsection, we propose our BPCPN metamodel, which is an extension of our previous PCPN metamodel presented in [10]. As illustrated in Fig. 15, the *Transition* metaclass has been specialized by the *BlackTransition* metaclass, which defines the new type of black transition in the model (see Section 3).

Moreover, the *Place* metaclass has been specialized by the metaclasses *Init*, *Exit*, *F*, *Control*, *End* and *Seq*, which correspond to the places with the same names and interpretation as in the BPCPN formalism (see Section 3).

Furthermore, we have implemented the following new validation rules in EVL in order to syntactically validate BPCPN graphical models, which conform to the BPCPN metamodel. For every *BlackTransition* t:

- Transition t must have at least one *Init* and one F precondition place. In the event of having further connections with other *Init* and F places, they would have a normal role from a semantic point of view, so the primary *Init* and F places are those used from a semantic point of view.
- The same F place acting as primary precondition place must also be a postcondition place of t.
- The inscription of the arc connecting t with F contains the inscription of the arc connecting F with t.
- t is loop-connected with the *Control* element, which is unique.

As an illustration, Listing 1 shows an EVL rule that validates if a given black transition has at least one Init and one F precondition places.

5.2 M2M and M2T Transformations

The M2M transformation implemented in ETL includes two different sets of rules. The first set of rules is used to transform a graphical CEP Domain model into a set of declarations. These declarations include the color sets associated with the domain model and all the variables required in the BPCPNs, as defined in Section 4.2 for the pattern operator transformations.

As an illustration, Listing 2 shows an excerpt of a rule to transform a given event into a color set. On the left-hand side of Fig. 16 we show the schema used in Example 2, while the right-hand side shows the variables obtained for Event A by the application of this rule.

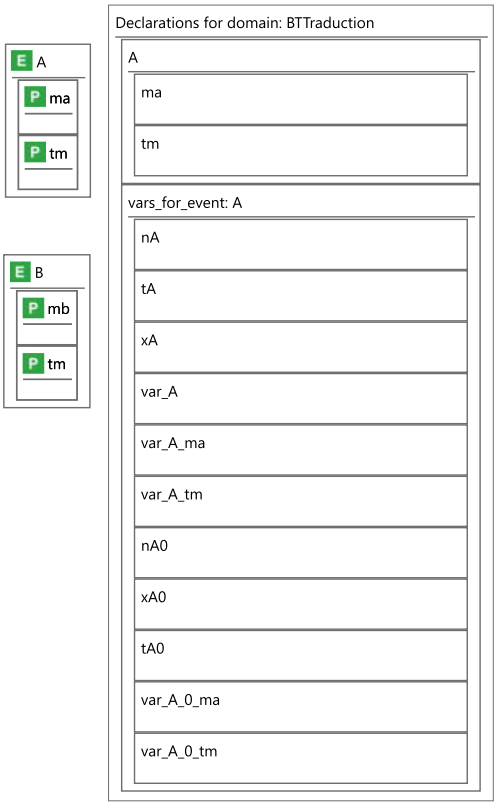

Fig. 16. CEP Domain graphical model for events A and B, and the BPCPN declarations obtained for event A.

The second set of rules is used to transform the event pattern graphical models into the corresponding BPCPN model pages, as indicated in Section 4.2. In this set of rules, we essentially have a transformation rule for each of the considered pattern operators. In addition, some auxiliary functions have been required to explore the different models and create the different elements of the corresponding BPCPN model. The translation rules are applied using a bottom-up approach. For instance, to produce the BPCPN model for a *followed-by* pattern operator we first obtain the BPCPN models of both arguments, which are then connected and modified in order to construct the resulting BPCPN. Listing 3 shows the excerpt for this pattern operator. Lines 10 and 12 of this code obtain the BPCPN models of the arguments. From line 13 to the end these two models are updated using auxiliary fuctions, such as `getEventList`, `draggedEvents`, `updateChildren` and `substituteFP`. In this rule, the `draggedEvents` function explores the model to obtain the list of precedent events when several *followed-by* pattern operators are concatenated.

```
1  context BlackTransition {
2      constraint HasPreconditionPlaces {
3          check : self.incoming.isDefined() and self.incoming.
           source.select(x|x.isTypeOf(Init)).size() >= 1 and
           self.incoming.source.select(x|x.isTypeOf(F)).size() >= 1
4          message : "A black transition must have at least one
           Init and one F precondition places"
5      }
6  }
```

Listing 1. The HasPreconditionPlaces EVL rule.

```
1  rule Event2ProductColor
2      transform e : Domain!Event
3      to rc : BPCPN!RecordColor {
4          //Establish the color set name
5          rc.name = e.typeName;
6          //For each property get its transformation
7          for (prop in e.eventProperties) {
8              var po ::= prop;
9              rc.recordOperands.add(po);
10         }
11 }
```

Listing 2. The Event2ProductColor ETL rule.

In the next step, the BPCPN model obtained is transformed into a PCPN model suitable for execution in CPN Tools. A M2T transformation has been implemented for this purpose. In this case the technology used is EGL, which allows us to generate an XML document conforming the DTD provided by CPN Tools. Using this language, each BPCPN element is then transformed into its corresponding XML element as stated in the DTD. The black transitions are transformed into their corresponding PCPN counterparts, as stated in Section 3. Listing 4 shows an excerpt of the operations applied for this transformation. Notice that this transformation also provides us with the capability to use a 2D matrix for the positioning of the graphical elements.

In Section 5.3, we provide a set of test cases that have been used to validate the correctness of these transformation rules.

It is worth highlighting that, for simplicity in the transformation, all places *Init*, *End*, *Exit* and *Fp* (F') for the pattern operator pages have color set *CEnd*, which is defined as $INT \times List\_CFs$ where *List_CFs* is a list of CF elements. With this implementation, when a place *INIT* has the color set *INT* we only need to write an empty list in the second

```
1  rule FollowedBy2Page
2      transform fb : Eventpattern!FollowedBy
3      to page : BPCPN!Page {
4          page.name = "FollowedBy_Operator_" +
           getGlobalCounter();
5          //Assigning the page type
6          page.type = BPCPN!PageTypeValue#FollowedBy;
7          globalBPCPNet.pages.add(page);
8          //Getting the transformation for the children pages
9          //Left children transformation
10         var childrenPage1 ::= fb.inboundLink.selectOne(l|l.
           order=1).operand;
11         //Right children transformation
12         var childrenPage2 ::= fb.inboundLink.selectOne(l|l.
           order=2).operand;
13             ...
14         //  Updating the children pages
15         //Obtaining the list of events preceding eventX
16         var event1 = childrenPage1.~event;
17         var list1 = getEventList(draggedEvents(event1));
18         var event2 = childrenPage2.~event;
19         var list2 = getEventList(draggedEvents(event2));
20         updateChildren(childrenPage2,list1, list2 );
21             ...
22         //General substitution of second children
23         substituteFP(fb,childrenPage2);
24 }
```

Listing 3. The FollowedBy2Page ETL rule.

```
operation substituteBT( transition : BlackTransition, page :
      Page):Bag{
  ...
  // Creating the new page elements
  var t1 = new Transition() ; t1.name = "t1";
  var t2 = new Transition() ; t2.name = "t2";
  var p = new Priority () ; p.name = "P_MAX"; p.value=0;
  t2. priority = p;
  var s = new Place(); s.name = "S";
  var next = new Transition() ; next.name = "next";
  ...
  // Creating the arcs to connect the new elements
  var arc_t1TOs = new Arc();
  arc_t1TOs.source = t1; arc_t1TOs.target = s;
  arc_t1TOs.inscription = initIncoming. inscription ;
  t1.outgoing.add(arc_t1TOs); s.incoming.add(arc_t1TOs);
  ...
}
```

Listing 4. The substituteBT EGL operation.

```
1  A(condA)
2  every A(condA)
3  A(condA) → B(condB)
4  A(condA) → B(condAB)
5  every A(condA) → B(condB)
6  A(condA) → (every(B(condB))
7  every(A(condA) → B(condB))
8  (every (A(condA)) → (every(B(condB))
9  (A(condA) → B(condB)) → C(condC)
10 A(condA) → (B(condB) → C(condC))
11 A(condA) → every (B(condB) → C(condC))
12 (every(A(condA)) → every(B(condB))) → every C(condC)
13 every((A(condA) → B(condB)) → C(condC))
14 every (A(condA) → B(condB)) → C(condC)
15 A(condA) → (every(B(condB)) → C(condC))
```

Listing 5. Test cases.

field. In the general case of a concatenation of elements obtained by previous followed-by operators, these would appear in the list in the order in which they were obtained.

5.3 Validation

As mentioned in the Introduction section, there is no official formal semantics for the Esper EPL event pattern operators. Thus, the validation of the transformation was performed using a set of test cases, checking the outputs obtained with the PCPN models and the outputs from the Esper EPL online tool. The results obtained for all of these test cases were the same.

Listing 5 contains the set of test cases that have been used to test the MEdit4CEP-BPCPN framework. These test cases cover the three operators that have been considered and serve to illustrate the compositionality of the transformation presented. In these cases, *condA* refers to a Boolean expression that depends on attributes of event type A, while *condAB* refers to a Boolean expression using attributes from both A and B.

The test cases are written in the listing using the abstract syntax defined in Section 4.2. However, for the proof, we have created the patterns using the graphical editor of our tool, MEdit4CEP-BPCPN. All of these test cases can be found at this URL,[8] where the reader can find the graphical models created with MEdit4CEP-BPCPN, and the BPCPNs and PCPNs obtained for them with the test scenarios. As an illustration, we describe Test cases 1, 2 and 11, with the input events shown in Table 1. These correspond to the initial marking of place *Input* in the input event flow CPN Tools page shown in Fig. 17.

Fig. 18 shows the graphical model created for the *FirstA* pattern (Example 3), which corresponds to Test case 1. Notice complex event *Case1*, which takes (select clause) the first field (*ma*) from the first A-type event fulfilling the condition $ma\,\%\,5 = 0$. Its corresponding BPCPN can be obtained by clicking on the BPCPN transformation button. This BPCPN consists of two pages: the input event flow page and the A page. Fig. 19 shows BPCPN page A in MEdit4CEP-BPCPN tool for the *FirstA* operator. This page

TABLE 1
Input Events for the Test Cases

Time	Simple Events
1s	$A = \{ma = 23, tm = 1\}$, $A = \{ma = 55, tm = 1\}$, $C = \{mc = 0, tm = 1\}$
2s	$B = \{mb = 25, tm = 2\}$, $A = \{ma = 50, tm = 2\}$
3s	$C = \{mc = 5, tm = 3\}$, $B = \{mb = 12, tm = 3\}$
4s	$C = \{mc = 10, tm = 4\}$, $A = \{ma = 0, tm = 4\}$
5s	$B = \{mb = 1, tm = 5\}$
6s	$B = \{mb = 5, tm = 6\}$
7s	$A = \{ma = 15, tm = 7\}$, $C = \{mc = 8, tm = 7\}$

corresponds to the implementation of the transformation shown in Fig. 7, the application of the select clause and the insertion of the events in F.

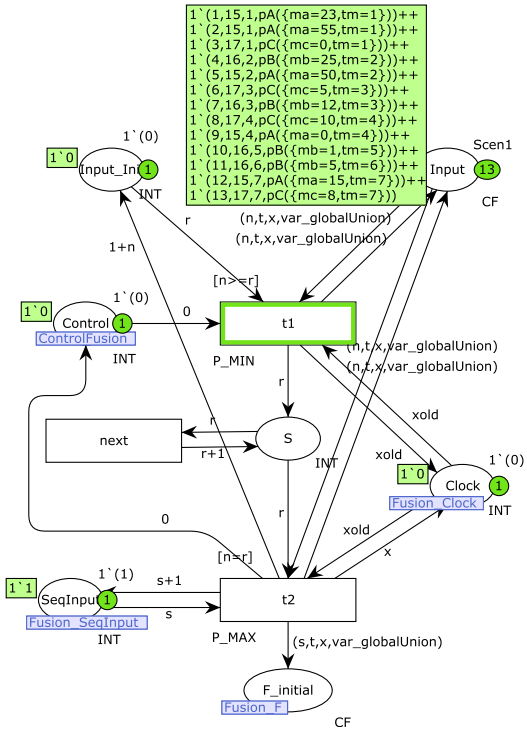

Fig. 17. CPN Tools page for input event flow with the initial marking.

8. http://dx.doi.org/10.17632/scrjyndpv6.1

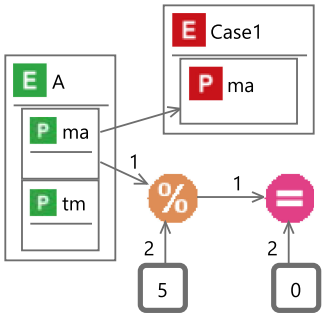

Fig. 18. Test Case 1: graphical model.

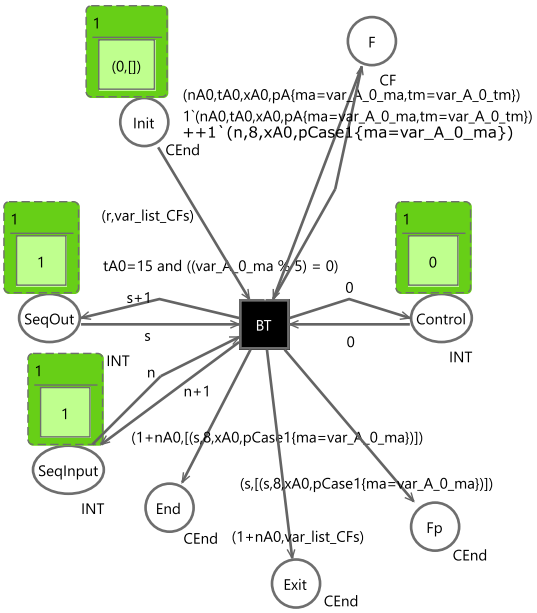

Fig. 19. Test Case 1: BPCPN page A in MEdit4CEP-BPCPN.

As seen from the transition guard in the figure, the number automatically assigned for A-type events is 15. Hence, once the BPCPN is obtained for an event pattern, we can obtain the corresponding PCPN, by clicking on the PCPN transformation button. Fig. 20 shows the A page taken from CPN Tools for this operator. The final marking in place *Fp* indicates that, as a result, the *FirstA* pattern has obtained the *pCase1* event with $ma = 55$. Observe that this complex event is also inserted into flow F (third line in the marking), which would allow further patterns to use it.

Test case 2 consists of three pages, the input event flow, the A page and the every page. Fig. 21 shows the PCPN page obtained for the *every* operator, once applied the select clause and the insertion of the new events in F. This PCPN page only contains the fusion places that conform the operator. In the A page an arc from $t2$ to *Init* has been included with respect to Fig. 20 (inner A-type event detection operator).

Fig. 22 shows the graphical model created in MEdit4CEP-BPCPN for Test case 11. In this case, complex event *Case11* consists of three fields, *ma*, *mb* and *mc*, which are taken from the A, B and C events detected by the pattern, with $condA = ma > 10$, $condB = mb > 2$ and $condC = mc > 0$. We only show the C page of the BPCPN obtained (Fig. 23), which corresponds to searching for the C-type events after the inner followed-by operator. Notice the inscription produced for the arc from place *Init* to transition *BT*, in which the second field contains a list with the two previous A and B events obtained. Finally, Fig. 24 shows the CPN Tools page that implements the external followed-by operator in Test case 11. Recall that the followed-by operator is essentially obtained by merging the BPCPNs obtained for the arguments (see Fig. 11). This CPN Tools

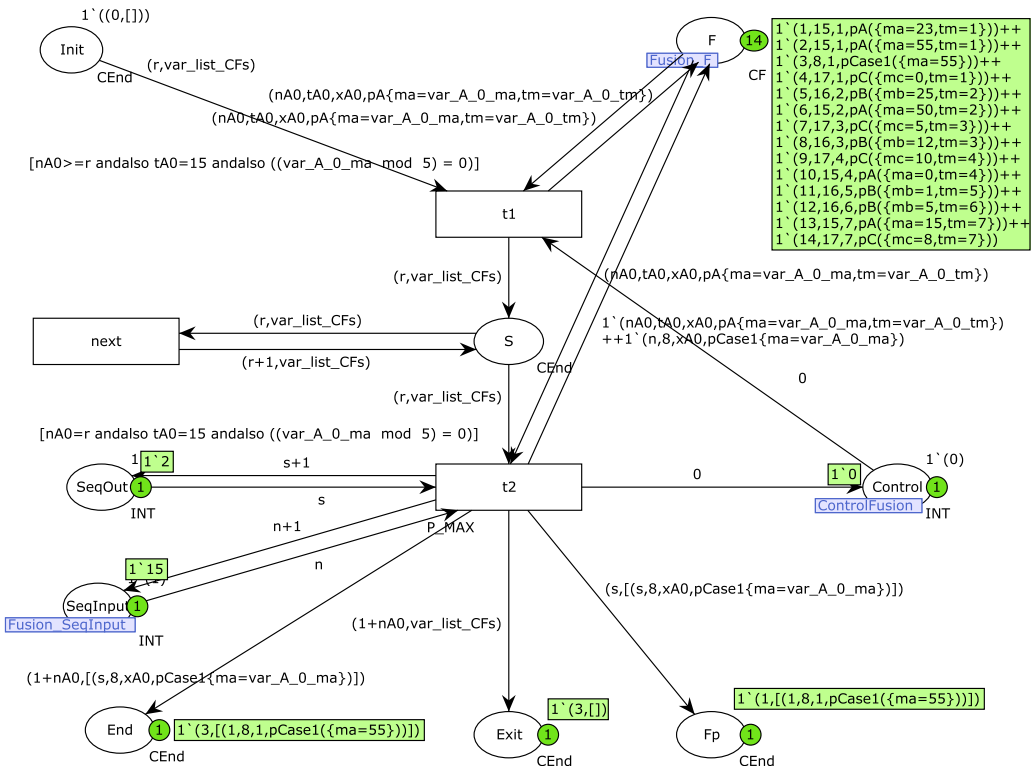

Fig. 20. Test Case 1: page A in CPN Tools.

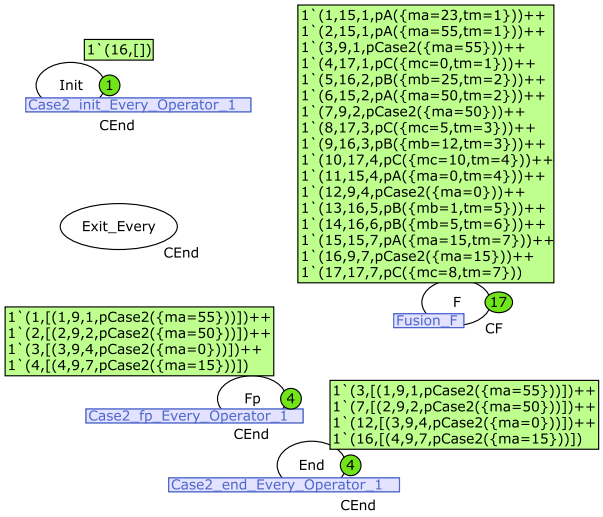

Fig. 21. Test Case 2: CPN Tools page for the every operator.

page contains the *Init*, *F*, *End*, *Exit*, *Control* and *Fp* places required in every pattern operator, most of which are, in this case, fusion places. Notice the token in place *Exit1*, which means that the first argument has finished, while place *Exit2* remains empty, because an every operator never terminates, and thus, the whole (followed-by) operator never terminates either. The marking in place *Fp* contains the three Case11 events found matching the pattern.

NOTE: all the test cases indicated in Listing 5 have been proved and checked with EsperTech EPL On-line. The outputs provided by the PCPNs in CPN Tools are the same as those obtained in Esper EPL On-line (https://www.esperonline.net/).

6 Use Case

Let us consider the health case study concerning the monitoring of uterine contractions of pregnant women in a hospital presented in [8]. In this case, we consider the events produced prior to the process of childbirth considering only the duration —beginning to end of one contraction (seconds)— and the frequency —beginning of one contraction to beginning of the next (minutes). For each patient we have the following event information: patient name, the duration of each uterine contraction and timestamp. This domain is depicted in Fig. 25, which contains the Patient event type along with its event properties: ts (event timestamp, in minutes), name (patient name) and contrDuration (contraction duration in seconds):

```
create schema Patient(ts int, name string,
                      contrDuration int);
```

We have defined two event patterns for this case study. First, we consider the complex event *Duration* depicted in Fig. 26, which searches for patients who have had at least two contractions with a duration of more than 35 seconds within a period of 5 minutes. This situation is an indication of a "she-is-in-labor" situation in maternity units in hospitals. This event pattern corresponds to Test case 5 in Listing 5. The EPL code produced by the MEdit4CEP-BPCPN tool for this event pattern is as follows:

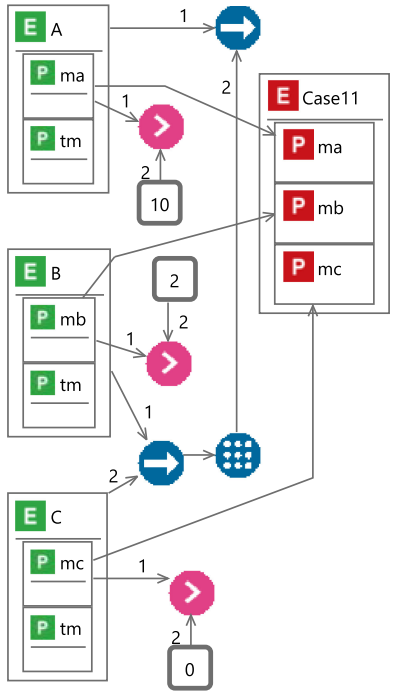

Fig. 22. Test Case 11: graphical model.

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

As regards the second event pattern, we have defined an event pattern to record the time elapsed for patients having contractions with a duration of more than 5 seconds before two contractions fulfilling the conditions of the pattern *Duration*. This pattern, called *TimeOnLabor*, is depicted in Fig. 27, and the EPL code obtained with the MEdit4CEP-BPCPN tool is as follows:

```
@Name('TimeOnLabor')
insert into TimeOnLabor
select a1.name as patientId,
a3.ts-a1.ts as before_delay
from pattern[
every a1=Patient(a1.contrDuration >5)->
(every a2=Patient(a2.contrDuration >35 and
a2.name=a1.name)->
a3=Patient(a3.name=a1.name
and a3.contrDuration>35 and
a3.ts-a2.ts <=5))];
```

In order to analyze both patterns, we have considered a scenario with two patients A and B. The initial marking considered (events) is that shown in Column 1 of Table 2.

The PCPN obtained consists of 75 declarations, including 14 types and 54 variables, and 12 pages with 88 places, 21 transitions and 121 arcs. Notice that these figures grow proportionally to the number of patterns modeled. Simulation took 224 steps (transition firings). The outputs obtained for both

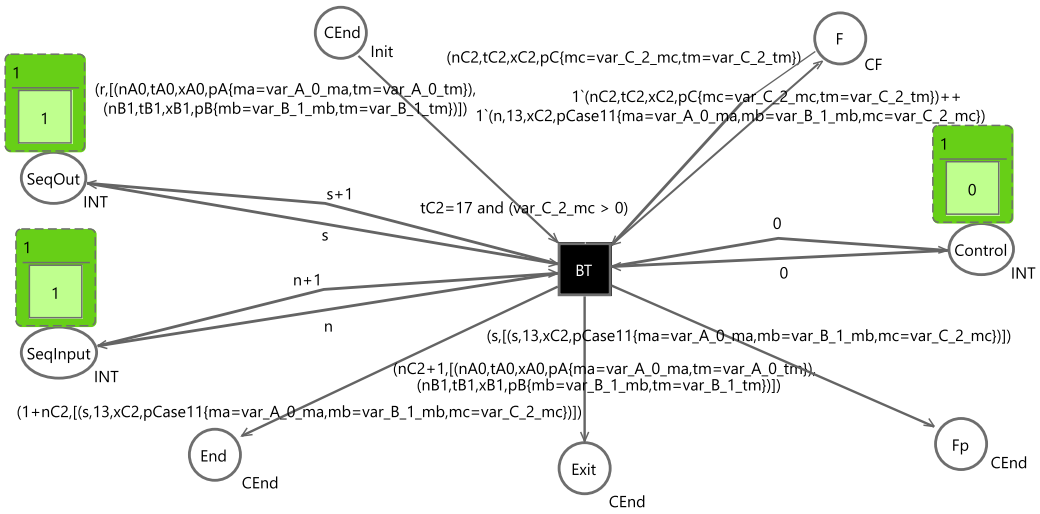

Fig. 23. Test Case 11: BPCPN page C in MEdit4CEP-BPCPN.

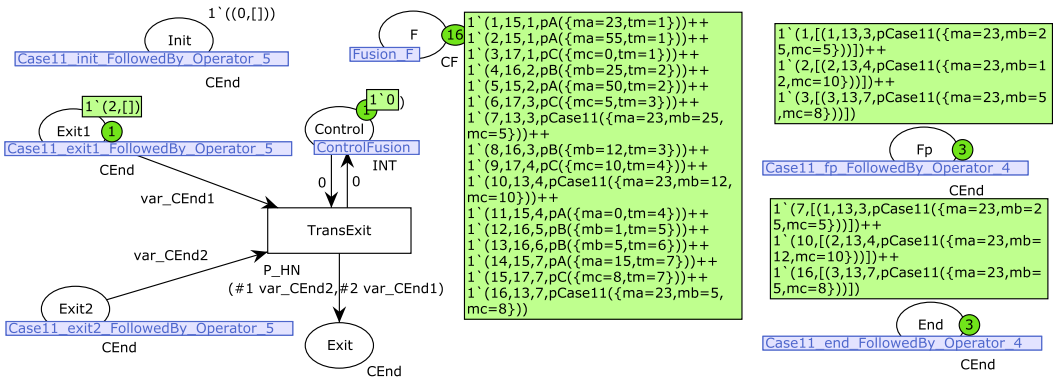

Fig. 24. Test Case 11: CPN Tools page for external followed-by operator.

patterns are shown in Columns 2 and 3 of Table 2. Other simulations led to the same results, but changing the order in which the events were produced at the same instant.

7 Related Work

Petri nets are a powerful graphical modeling tool to represent and analyze the behavior of concurrent systems. PNs have been extended in many different ways to facilitate the modeling of specific application areas. For instance, Hafidi *et al.* [20] use Reconfigurable Timed Net Condition/Event Systems (R-TNCESs) to perform a formal verification of reconfigurable discrete event control systems (RDECSs). Rodidoux *et al.* [21] present a procedure similar to our work using CPNs. In this case, a formal model to specify reliability block diagrams (DRBD) for computer-based systems is used

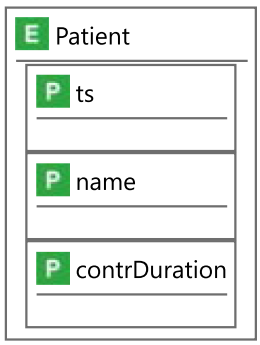

Fig. 25. Patient domain: graphical model.

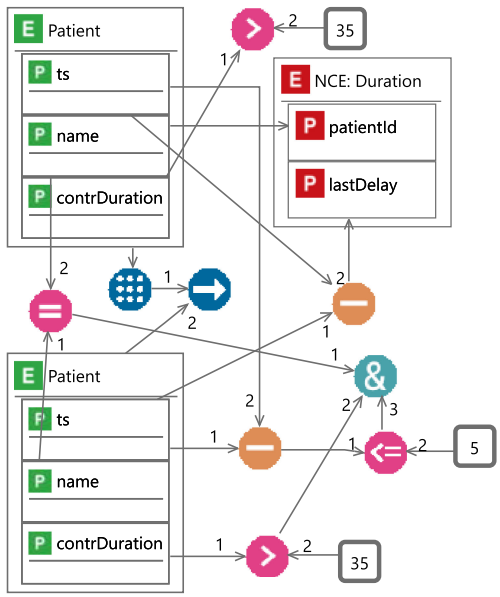

Fig. 26. Duration pattern: graphical model.

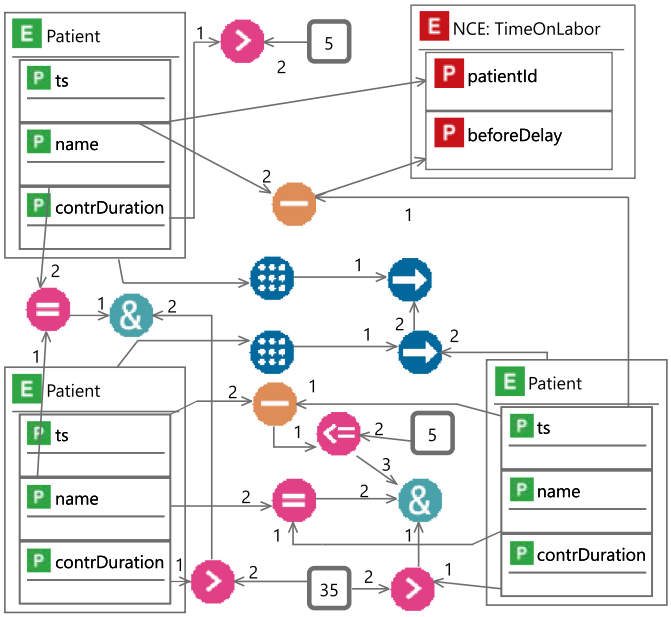

Fig. 27. TimeOnLabor pattern: graphical model.

to verify it dynamically. In this compositional approach proposed by Rodidoux *et al.*, a DRBD model written in RML (Reliability Markup Language) is converted into a CPN model used to analyze the behavioral properties of the DRBD model.

In this paper we have defined an extension of PCPNs with black sequencing transitions, in order to model the ordered firing of these transitions with respect to the tokens consumed from an input flow place. The use of black transitions in PNs is not new. Valero *et al.* [22] used black transitions in a timed extension of Petri nets to denote a class of transitions that must fire when the time interval at which they can fire expires, Niek Tax *et al.* [23] used black transitions to denote invisible actions in the system, while Lorbeer and Padberg [24] used black transitions to denote the enabled transitions in the context of reconfigurable Petri nets. In their paper, Lorbeer and Padberg proposed a hierarchical structure for reconfigurable Petri nets using substitution transitions and a set of rules to reconfigure the global net and the subnets. Substitution transitions are a mechanism to refine the models, as they represent subnets that preserve the transition connections (input and output places). In reconfigurable Petri nets the system structure can therefore change during its execution. Kahloul *et al.* [25] applied reconfigurable object nets (RONs) for the modeling, simulation and analysis of reconfigurable manufacturing systems (RMS). Graph transformation techniques were used in this case for the reconfiguration process.

Thus, the black transitions presented in the present paper have some similarities with substitution transitions, since they represent a PCPN structure that essentially replaces the black transition. However, black transitions do not have the same behavior as normal transitions, so this substitution is not a simple refinement in which the behavior represented by some normal transitions is detailed by their associated subnets.

Regarding the use of formal methods and Petri nets in the field of the CEP technology, Offel *et al.* [26] showed the need for a comprehensive formalization integrating event streams, event queries and evaluation architectures. For instance, Colored Petri Nets with Priorities and Time (PTCPNs) were used by Weidlich *et al.* [27] to define a model of event processing networks (EPNs). They justified the use of PTCPNs due to their capability of expressing concurrency and support for typing events. A general transformation of EPNs to PTCPNs was then presented by the authors, and a specific application to the fast flower delivery case. Weidlich *et al.*'s work was based on a timed model of CPNs, in contrast to our work, in which we use untimed PCPNs to define a compositional model for the most relevant operators of Esper EPL. We have also used M2M techniques to implement a tool supporting this translation.

Carle *et al.* [28] used untimed CPNs to detect critical situations of interest in the aerospace field. They defined the *chronicles* language, which is a situation description language with operators to capture the detection of simple events, sequence, disjunction, conjunction and absence of a chronicle. A translation of the chronicles language was provided in terms of untimed CPNs with inhibitor arcs and fusion places. Despite the use of untimed CPNs, there are significant differences with our work. Carle *et al.* only presented the situations described by the chronicles language, while we present a compositional semantics in a quite different way by using the newly presented black transitions and not using inhibitor arcs.

A different class of Petri nets is used in the work by Ahmad *et al.* [29], in which the authors presented a methodology to model CEP using Timed Net Condition Event System (TNCES) [30]. TNCES is a timed extension of NCES (Net Condition Event Systems) [31], which is a formalism that allows the modular design of discrete event systems. The timed extension considered in TNCES is based on timed-arc Petri nets [32], [33], in which *pt-arcs* have a time interval to restrict

TABLE 2
Initial Marking and Outputs Obtained for *Duration* and *Time-On-Labor* Patterns

Input	Duration	Time-On-Labor
1`(1,2,1,pPatient(name="A",contrDuration=12,ts=1))++ 1`(2,2,1,pPatient(name="B",contrDuration=10,ts=1))++ 1`(3,2,15,pPatient(name="A",contrDuration=45,ts=15))++ 1`(4,2,17,pPatient(name="B",contrDuration=45,ts=17))++ 1`(5,2,21,pPatient(name="A",contrDuration=15,ts=21))++ 1`(6,2,23,pPatient(name="B",contrDuration=38,ts=23))++ 1`(7,2,26,pPatient(name="B",contrDuration=38,ts=26))++ 1`(8,2,30,pPatient(name="A",contrDuration=36,ts=30))++ 1`(9,2,34,pPatient(name="A",contrDuration=45,ts=34))++ 1`(10,2,34,pPatient(name="B",contrDuration=38,ts=34))++ 1`(11,2,37,pPatient(name="A",contrDuration=15,ts=37))++ 1`(12,2,39,pPatient(name="B",contrDuration=38,ts=39));	1`(8,1,26,pDuration(patientId="B",lastDelay=3))++ 1`(13,1,34,pDuration(patientId="A",lastDelay=4))++ 1`(20,1,39,pDuration(patientId="B",lastDelay=5))++	1`(9,2,26,pTimeOnLabor(patientId="B",beforeDelay=25))++ 1`(10,2,26,pTimeOnLabor(patientId="B",beforeDelay=9))++ 1`(14,2,34,pTimeOnLabor(patientId="A",beforeDelay=13))++ 1`(15,2,34,pTimeOnLabor(patientId="A",beforeDelay=33))++ 1`(16,2,34,pTimeOnLabor(patientId="A",beforeDelay=19))++ 1`(21,2,39,pTimeOnLabor(patientId="B",beforeDelay=16))++ 1`(22,2,39,pTimeOnLabor(patientId="B",beforeDelay=13))++ 1`(23,2,39,pTimeOnLabor(patientId="B",beforeDelay=22))++ 1`(24,2,39,pTimeOnLabor(patientId="B",beforeDelay=38))

the age of the tokens to be consumed. In addition to the class of Petri nets used, the main differences with our work are that we define a new type of transition, the black transition, which allows us to simplify the definition of a compositional semantics for the most relevant Esper EPL operators.

Thus, it is important to note that in contrast to these works using different variants and extensions of Petri nets for the modeling of CEP systems, we have been able to define a compositional BPCPN semantics, which means that we can deal with complex pattern definitions, and the translations can be automatically produced using the tool that has been implemented using model-to-model techniques.

A number of works have used different formalisms. Al Bassit *et al.* [34] defined LEAD, a pattern algebra that extends the common set of operators in CEP. These authors defined a set of pattern operators, a rule grammar and a logical execution plan, which is based on a combination of timed colored Petri nets with aging tokens and prioritized Petri nets. Hinze and Voisard [35] presented a parameterized event algebra (EVA) to support adaptable event composition. Cugola and Margara [36] defined the TESLA language, which is a highly expressive and flexible language offering content and temporal filters, negations, timers, aggregates, and fully customizable policies for event selection and consumption. Agrawal *et al.* [37] defined a timed automata formalization of complex event systems, and the Sase+ pattern language was defined by Gyllstrom et al [38]. They defined a precise semantics in terms of timed automata with similar results to the work described in TESLA. Ericsson *et al.* [39] presented another formalization based on timed automata, in which they use the REX tool [40] to analyze the events and rules specified for CEP applications.

Some papers integrated CEP with Business Process Model (BPM) technologies and provide a mapping to PN as implementation semantics. Mandal *et al.* [41] proposed a model for event handling in business processes that is based on explicit subscriptions and event buffering. A formal execution semantics for this model is provided as a BPMN extension and a mapping to CPNs, justifying the use of CPNs as they are particularly suited to capturing the interactions of a process with its environment.

A large number of Petri net metamodels have been proposed. Gomez *et al.* [42] defined a metamodel for modeling Petri nets applied to biological data processing. Models conforming to Gomez *et al.*'s metamodel can be automatically transformed into the XML code executable by CPN Tools. However, Petri net modeling is close to CPN Tools concepts such as color and position of the graphical elements and is conducted through a tree model editor (not a graphical one with nodes and links). Moreover, a CPN model can contain only one page.

Westergaard *et al.* [43] developed Access/CPN, a framework that provides CPN Tools with two interfaces. One interface for analysis methods is implemented in Standard Markup Language, while the another Java one supports the object-oriented representation of CPN models. Since their metamodel is implemented with Eclipse Modeling Framework (EMF), Petri net modeling is conducted by making use of a tree model editor.

Kindler proposed ePNK (http://www.imm.dtu.dk/~ekki/projects/ePNK/index.shtml), a platform that allows us to define Petri nets by using both a tree editor and a GMF editor. However, the graphical editor does not support CEP domain and event pattern modeling nor does it support their automatic transformation into Petri nets.

Even though some works use CPNs to model CEP-based languages and other existing ones propose model-driven approaches for modeling Petri nets, to the best of our knowledge, none of them provides domain experts with a compositional approach for complex event pattern modeling and transformation to CPNs with black transitions. This MEdit4CEP-BPCPN model-driven approach proposed in the presented paper is supported by an all-in-one graphical tool with facilities for (1) modeling CEP domains and event patterns in a user-friendly way by dragging and dropping elements on a canvas, (2) validating the pattern syntax, (3) automatically transforming the graphical patterns into a compositional BPCPN model, (4) automatically transforming the BPCPN model to the XML code executable by CPN Tools and validating the pattern semantics, and (5) automatically generating the Esper EPL code and deploying it in a CEP engine.

8 Conclusion

This paper presented an extension of PCPNs, called BPCPNs, in which we included a new type of transitions, the so-called *black sequencing transition*, which allow us to fire these transitions following an order. An encoding of BPCPNs into PCPNs was also presented. We used BPCPNs to provide a compositional semantics of the most relevant operators of Esper EPL, namely the basic event detection, the every operator and the followed-by operator. These operators allow us to detect the main situations of interest, such as the first event fulfilling a certain condition, all events fulfilling a condition and streams of events. In addition, the use of separate event patterns enriches the model, as the complex events produced by some patterns can be used as inputs in subsequent event patterns. Thus, our contribution is to provide a rigorous semantics to EPL, since most information about the EPL operator behavior is only provided textually.

This transformation was then implemented in the MEdit4CEP-BPCPN tool by using model-to-model and model-to-text techniques based on the Eclipse Epsilon platform. This choice has some advantages that are further enhanced by our tool, since the BPCPN editor provided not only allows users to modify the obtained models but also to create their own models. Another important advantage is the possibility of extending the model and the transformation with new additions. This has been proven since our MEdit4CEP-BPCPN is an extension of our previous work MEdit4CEP-CPN.

Thus, with this tool a domain expert can easily create event patterns graphically, which can, in turn, be transformed either into EPL code or into a BPCPN model. This BPCPN model can also be transformed into a PCPN model executable in CPN Tools, which allows us to simulate and analyze the pattern behavior, and make predictions about scenarios of interest.

As future work, our plans include the modeling of more complex real-world scenarios and performing a usability analysis of the whole framework, so as to obtain the quantitative and qualitative feedback from the users.

We also plan to extend the event pattern operators supported in our framework, including other EPL operators, such as the every-distinct, repeat, repeat-until, conjunction and disjunction, as well as data windows. Thus, with these new event pattern operators, we will be able to apply the MEdit4CEP-BPCPN to other application domains such as e-health and distributed environments [44], [45]. For instance, the techniques presented in this work can be adapted to detect deviations between a process model and their running instances, using the event logs following the path established by Wang *et al*. in [46].

Finally, we plan to apply the techniques used in [47] to feed the input event flow in real time with the events produced by a set of sensors. This feature requires utilizing the CPN Tools capabilities to use HTTP sockets to feed input streams. With this additional feature we can conduct a real time analysis of the events produced from sensor stations.

Acknowledgments

This work was supported by the Spanish Ministry of Science, Innovation and Universities and the European Union FEDER funds under Grants RTI2018-093608-B-C32 and RTI2018-093608-B-C33, the JCCM project cofinanced with European Union FEDER funds, ref. SBPLY/17/180501/000276, and the UCLM group research grant with reference 2020-GRIN-28708. *Data availability*: The data obtained supporting the findings of this research can be found in the Mendeley repository (http://dx.doi.org/10.17632/scrjyndpv6.1). In this repository the reader can find all the files that are necessary to replicate the results obtained for the test cases and the use case.

References

[1] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley, 2001.

[2] N. Giatrakos, E. Alevizos, A. Artikis, A. Deligiannakis, and M. N. Garofalakis, "Complex event recognition in the big data era: A survey," *VLDB J.*, vol. 29, no. 1, pp. 313–352, 2020.

[3] D. Luckham, *Event Processing for Business: Organizing the Real-Time Enterprise*. Hoboken, New Jersey, USA: Wiley, 2012.

[4] K. Bok, D. Kim, and J. Yoo, "Complex event processing for sensor stream data," *Sensors*, vol. 18, no. 9, Sep. 2018, Art. no. 3084. [Online]. Available: http://dx.doi.org/10.3390/s18093084

[5] G. Díaz, H. Macià, V. Valero, J. Boubeta-Puig, and F. Cuartero, "An intelligent transportation system to control air pollution and road traffic in cities integrating CEP and colored Petri nets," *Neural Comput. Appl.*, vol. 32, no. 2, pp. 405–426, 2020.

[6] C. Czepa and U. Zdun, "On the understandability of temporal properties formalized in linear temporal logic, property specification patterns and event processing language," *IEEE Trans. Softw. Eng.*, vol. 46, no. 1, pp. 100–112, Jan. 2020.

[7] 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.

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

[9] CPN-Group, "CPN tools homepage," Accessed: Feb. 22, 2021. [Online]. Available: http://www.cpntools.org/

[10] J. Boubeta-Puig, G. Díaz, H. Macià, V. Valero, and G. Ortiz, "MEdit4CEP-CPN: An approach for complex event processing modeling by prioritized colored Petri nets," *Inf. Syst.*, vol. 81, pp. 267–289, 2019.

[11] G. Díaz, H. Macià, V. Valero, J. Boubeta-Puig, and G. Ortiz, "Facilitating the quantitative analysis of complex events through a computational intelligence model-driven tool," *Sci. Program.*, vol. 2019, pp. 1–17, 2019.

[12] J. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1981.

[13] K. Jensen, *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. London, U.K.: Springer, 1995.

[14] K. Jensen and L. M. Kristensen, *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*, 1st ed. Berlin, Germany: Springer, 2009.

[15] W. M. P. van der Aalst, C. Stahl, and M. Westergaard, "Strategies for modeling complex processes using colored Petri nets," in *Transactions on Petri Nets and Other Models of Concurrency VII*. Berlin, Germany: Springer, 2013, pp. 6–55.

[16] K. Jensen, L. M. Kristensen, and L. Wells, "Coloured Petri nets and CPN tools for modelling and validation of concurrent systems," *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 3–4, pp. 213–254, 2007.

[17] J. Boubeta-Puig, G. Ortiz, and I. Medina-Bulo, "A model-driven approach for facilitating user-friendly design of complex event patterns," *Expert Syst. Appl.*, vol. 41, no. 2, pp. 445–456, Feb. 2014.

[18] Eclipse Foundation, "Epsilon," 2021. [Online]. Available: https://www.eclipse.org/epsilon/

[19] 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.

[20] Y. Hafidi, L. Kahloul, M. Khalgui, Z. Li, K. Alnowibet, and T. Qu, "On methodology for the verification of reconfigurable timed net condition/event systems," *IEEE Trans. Syst., Man, Cybern. Syst.*, vol. 50, no. 10, pp. 3577–3591, Oct. 2020.

[21] R. Robidoux, H. Xu, L. Xing, and M. Zhou, "Automated modeling of dynamic reliability block diagrams using colored Petri nets," *IEEE Trans. Syst., Man, Cybern. A: Syst. Hum.*, vol. 40, no. 2, pp. 337–351, Mar. 2010.

[22] V. Valero, M. Emilia Cambronero, G. Díaz, and H. Macià, "A Petri net approach for the design and analysis of web services choreographies," *J. Logic Algebraic Program.*, vol. 78, no. 5, pp. 359–380, May 2009.

[23] N. Tax, N. Sidorova, R. Haakma, and W. M. van der Aalst, "Mining local process models," *J. Innov. Digit. Ecosyst.*, vol. 3, no. 2, pp. 183–196, Dec. 2016.

[24] J. U. Lorbeer and J. Padberg, "Hierarchical, reconfigurable Petri nets," in *Proc. CEUR Workshop Proc.*, vol. 2060, pp. 167–186, 2018. [Online]. Available: http://ceur-ws.org/Vol-2060/pemod1.pdf

[25] L. Kahloul, S. Bourekkache, and K. Djouani, "Designing reconfigurable manufacturing systems using reconfigurable object Petri nets," *Int. J. Comput. Integr. Manuf.*, vol. 29, no. 8, pp. 889–906, 2016.

[26] M. Offel, H. van der Aa, and M. Weidlich, "Towards net-based formal methods for complex event processing," in *Proc. Conf. "Lernen, Wissen, Daten, Analysen"*. 2018, pp. 281–284.

[27] 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.

[28] P. Carle, C. Choppy, R. Kervarc, and A. Piel, "A formal coloured Petri net model for hazard detection in large event flows," in *Proc. 20th Asia-Pacific Softw. Eng. Conf.*, 2013, pp. 323–330.

[29] 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. IEEE 10th Int. Conf. Ind. Inform*, 2012, pp. 380–385.

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

[31] H. Hanisch, J. Thieme, A. Luder, and O. Wienhold, "Modeling of PLC behavior by means of timed net condition/event systems," in *Proc. IEEE 6th Int. Conf. Emerg. Technol. Factory Autom.*, 1997, pp. 391–396.

[32] H. M. Hanisch, "Analysis of place/transition nets with timed-arcs and its application to batch process control," *Appl. Theory Petri Nets*, vol. 691, pp. 282–299, 1993.

[33] V. Valero, D. de Frutos, and F. Cuartero, "On non-decidability of reachability for timed-arc Petri nets," in *Proc. 8th Int. Workshop Petri Nets Perform. Models*, 1999, pp. 188–197.

[34] A. A. Bassit, S. Skhiri, and H. Ammar, "LEAD: A formal specification for event processing," in *Proc. 13th ACM Int. Conf. Distrib. Event-Based Syst.*, 2019, pp. 91–102.

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

[36] G. Cugola and A. Margara, "TESLA: A formally defined event specification language," in *Proc. 4th ACM Int. Conf. Distrib. Event-Based Syst.*, 2010, pp. 50–61.
[37] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 147–160.
[38] D. Gyllstrom, J. Agrawal, Y. Diao, and N. Immerman, "On supporting kleene closure over event streams," in *Proc. 24th Int. Conf. Data Eng.*, 2008, pp. 1391–1393.
[39] A. Ericsson, P. Pettersson, M. Berndtsson, and M. Seiriö, "Seamless formal verification of complex event processing applications," in *Proc. Int. Conf. Distrib. Event-Based Syst.*, 2007, pp. 50–61.
[40] A. Ericsson and M. Berndtsson, "REX, the Rule and Event eXplorer," in *Proc. Int. Conf. Distrib. Event-Based Syst.*, 2007, pp. 71–74.
[41] S. Mandal, M. Weidlich, and M. Weske, "Events in business process implementation: Early subscription and event buffering," in *Proc. Int. Conf. Business Process Manage.*, 2017, pp. 141–159
[42] A. Gomez, A. Boronat, J. A. Carsi, I. Ramos, C. Taubner, and S. Eckstein, "Biological data processing using model driven engineering," *IEEE Latin Amer. Trans.*, vol. 6, no. 4, pp. 324–331, Aug. 2008.
[43] M. Westergaard and L. M. Kristensen, "The access/CPN framework: A tool for interacting with the CPN tools simulator," in *Proc. Int. Conf. Appl. Theory Petri Nets*, 2009, pp. 313–322.
[44] I. Calvo, M. G. Merayo, and M. Núñez, "A methodology to analyze heart data using fuzzy automata," *J. Intell. Fuzzy Syst.*, vol. 37, no. 6, pp. 7389–7399, 2019
[45] R. M. Hierons, M. G. Merayo, and M. Núñez, "Bounded reordering in the distributed test architecture," *IEEE Trans. Rel.*, vol. 67, no. 2, pp. 522–537, Jun. 2018.
[46] L. Wang, Y. Du, and L. Qi, "Efficient deviation detection between a process model and event logs," *IEEE/CAA J. Automatica Sinica*, vol. 6, no. 6, pp. 1352–1364, Nov. 2019.
[47] G. Díaz, E. Brazález, H. Macià, J. Boubeta-Puig, and V. Valero, "An intelligent system integrating CEP and colored Petri nets for helping in decision making about pollution scenarios," in *Proc. 15th Int. Work-Conf. Artif. Neural Netw.*, 2019, pp. 729–740.

Valentín Valero received the degree in mathematics from the Complutense University of Madrid, in 1987, and the PhD degree in mathematics from the Department of Computer Science, Complutense University of Madrid, in 1993. Since 1987, he has been a member of the Computer Science Department, University of Castilla-La Mancha, Spain, where he is a full professor of distributed systems and operating systems with the Computer Science School of Albacete. His current research interests include concurrency, specifically in formal models for analysis and design of concurrent systems, and real-time systems.

Gregorio Díaz is an associated professor with the University of Castilla-La Mancha within the ReTiCS research group with tenure distinction (2011), published more than 18 journal papers, from which 16 are indexed by the JCR index, participated in 38 international and national conferences, main researcher of three FEDER projects. His research goals are aimed to make software more reliable, secure, and easier to design. He has supervised more than 27 master theses, including four in research areas and two PhD thesis. He has taught in undergraduate and postgraduate studies awarded with the quality award Euro-Inf Bachelor by EQANIE.

Juan Boubeta-Puig received the PhD degree in computer science from the University of Cádiz, in 2014. He is a tenured associate professor with the Department of Computer Science and Engineering with the University of Cádiz (UCA), Spain. He was honored with the Extraordinary PhD Award from UCA and the Best PhD Thesis Award from the Spanish Society of Software Engineering and Software Development Technologies (SISTEDES). His research interests include real-time big data analytics through Complex event processing (CEP), Event-driven service-oriented architecture (SOA 2.0), Internet of Things (IoT) and Model-driven development (MDD) of advanced user interfaces, and their application to e-health, smart city, industry 4.0, and cybersecurity.

Hermenegilda Macià received the degree in mathematics from the University of Valencia, and the PhD degree in computer science from the University of Castilla-La Mancha, in 2003. She is an associated professor with the Department of Mathematics, University of Castilla-La Mancha within the ReTiCS research group. She has published research articles in reputed journals of Mathematics and Computer Science. In the last years, her main research interests are focused on applying formal method in different areas such as in SSME (Service science, management and engineering), specifically considering CEP (Complex event processing) or to modeling biological systems, including metabolic pathways.

Enrique Brazález received the master's degree in computer science, in 2018–2019. He is currently working toward the PhD degree in advanced information technologies at the University of Castilla-La Mancha within the Retics research group. He has participated in some national conferences and he works in a FEDER project called "Analysis and Development of Models for Air Quality Control, Effects on Vegetation and Cloud Environments". His field of research is Complex Event Processing and its application in Industry 4.0 environments such as renewable energies and the study of air quality.