Software & Systems Modeling
https://doi.org/10.1007/s10270-019-00716-1

REGULAR PAPER

Profiling the publish/subscribe paradigm for automated analysis using colored Petri nets

Abel Gómez[1] · Ricardo J. Rodríguez[2] · María-Emilia Cambronero[3] · Valentín Valero[3]

Received: 9 April 2018 / Accepted: 7 December 2018

Abstract
UML sequence diagrams are used to graphically describe the message interactions between the objects participating in a certain scenario. Combined fragments extend the basic functionality of UML sequence diagrams with control structures, such as sequences, alternatives, iterations, or parallels. In this paper, we present a UML profile to annotate sequence diagrams with combined fragments to model timed Web services with distributed resources under the publish/subscribe paradigm. This profile is exploited to automatically obtain a representation of the system based on Colored Petri nets using a novel model-to-model (M2M) transformation. This M2M transformation has been specified using QVT and has been integrated in a new add-on extending a state-of-the-art UML modeling tool. Generated Petri nets can be immediately used in well-known Petri net software, such as CPN Tools, to analyze the system behavior. Hence, our model-to-model transformation tool allows for simulating the system and finding design errors in early stages of system development, which enables us to fix them at these early phases and thus potentially saving development costs.

Keywords UML 2.5 · Distributed resources · Publish/Subscribe · Automated analysis · WSRF · WSN · Colored Petri nets · CPN tools

1 Introduction

Web services are usually stateless [1], which means that no state information from clients is stored as interactions with the server occur. RESTful web services, for instance, are built to work on resources (data and services) under a client/server architecture using a stateless communication protocol (such as HTTP). However, in many cases, we have stateful distributed services, in which the user has got the ability to access and manipulate states, that is, data values that persist and evolve as a result of Web service (WS) interactions. Consider, for instance, a user shopping through an online website, in which the user can add/remove items to/from the shopping cart during the interaction. In this case, simple object access protocol (SOAP) is usually the messaging protocol used for the cient/server interactions. SOAP is a platform- and language-independent standardized protocol, which has some WS-extensions to provide additional functionalities, such as WS-Addressing, WS-Security, and WS-AtomicTransaction.

It is therefore desirable to have Web service conventions to enable the discovery of, introspection on, and interaction with stateful distributed resources in standard and interoperable ways. In particular, we focus our attention on timed Web services managing a collection of distributed resources using the publish/subscribe (PS) paradigm and the OASIS WSRF standard. The publish/subscribe paradigm [2] provides a loosely coupled form of interaction between

Communicated by Antonio Vallecillo.

✉ Ricardo J. Rodríguez
rjrodriguez@unizar.es

Abel Gómez
agomezlla@uoc.edu

María-Emilia Cambronero
memilia.cambronero@uclm.es

Valentín Valero
valentin.valero@uclm.es

[1] Internet Interdisciplinary Institute (IN3), Universitat Oberta de Catalunya, Avda. Carl Friedrich Gauss, 5, Castelldefels, 08060 Barcelona, Spain

[2] Centro Universitario de la Defensa, Academia General Militar, Carr. de Huesca s/n, 50090 Zaragoza, Spain

[3] Dpto. de Sistemas Informáticos, Escuela Superior de Ingeniería Informática de Albacete, Universidad de Castilla-La Mancha, 02071 Albacete, Spain

Published online: 22 January 2019

the participants in a distributed system that want to be notified when some event of interest occurs. Publish/subscribe systems are usually divided into two categories: subject-based and content-based. In subject-based systems, a publisher makes visible a message (within a topic) and all subscribers to such a topic will then receive the notification message. In the content-based approach, however, subscribers indicate a predicate or condition related to a resource, so they are only notified when a change in the resource makes such a predicate to hold. The latter is the case we consider in this paper, where resources are published by some publisher, with an initial state (integer value) and subscribers can submit their subscription conditions in order to be notified when these conditions are true. On the other hand, WSRF is the de facto OASIS standard for modeling and accessing stateful resources using Web services, so it provides us with standardized operations for the resource management. These operations allow us to set/get the resource property values and lifetime. In addition, Web services notification (WSN) [3] complements WSRF by establishing a standardized way for Web services to interact by using subscriptions and event notifications.

In this paper, we use UML 2.5 sequence diagrams extended with the so called *Combined Fragments* [4] to define the interactions in a Web service composition managing a collection of distributed resources. However, UML sequence diagrams are not very suitable to execute or to analyze the modeled system in an automatic way. A common solution to overcome this issue in the scientific community is to transform these UML models to other formal models for which a well-defined mathematical framework exists, and hence these obtained models are suitable for analysis purposes. In this paper, we consider an extension of Petri nets, particularly Colored Petri nets (CPN[1]) [5], as formal model since Petri nets are a mathematical formalism that easily represent common characteristics of computer systems, such as concurrency, synchronization, conditional branching, and sequencing. A CPN model also allows for expressing the timing events occurring in the system.

UML can be also extended to customize UML models for a particular domain by means of *profiling* [6]. The particular problem domain is first mapped to a UML model, obtaining a *UML domain model*. Then, a *UML profile* is built from this UML domain model. A UML profile defines one or more *stereotypes* that are used to mark a model element as representing a particular kind of object in the corresponding domain. These stereotypes can also be extended with a list of properties (termed as *attribute values*), useful to enrich the type description. A UML model in which a profile is being used is usually termed a *UML annotated model*.

The contribution of this paper is threefold. First, we present a UML profile for the publish/subscribe paradigm. This UML profile, created following the guidelines given in Lagarde et al. [7] and Selic [6], enables us to easily represent the underlying concepts of the publish/subscribe paradigm in UML models. Second, we propose a pattern-based M2M transformation to translate UML annotated models into CPNs. We describe the model transformation using a running example. Third, we present a tool that implements the M2M transformation using the QVT [8] OMG standard. This tool is able to automatically transform UML models annotated with our novel profile into a CPN model. The obtained CPN model is compatible with the format used by CPN Tools, a widespread tool for editing, simulating, and analyzing CPN models [9]. Hence, the obtained models can be analyzed to find design errors, thus enabling us to detect and fix design errors in early stages of system development and hence, to save production (and other) costs [10]. We describe also the validation that can be carried out in the generated CPN model.

This paper stems from previous proposals. A first preliminary version of this work was published in Cambronero and Valero, [11], where a smaller set of UML 2.0 constructions was considered. However, the time aspects were excluded in such a model. Afterward, in Valero and Cambronero [12], an algebraic syntax for sequence diagrams with combined fragments and WSRF was defined. An operational semantics to define the behavior of the modeled systems in a rigorous way was also introduced. In this paper, we consider those previous works as basis, and introduce a (more abstract) system view of the publish/subscribe paradigm by providing a new UML profile. This UML profile is complemented by a novel M2M transformation implemented in its companion tool. This tool allows us to obtain the corresponding formal models and thus enables us to analyze the modeled system in an effective way.

This paper is organized as follows. Section 2 gives some background on the publish/subscribe paradigm and Colored Petri nets. The technical approach that we followed in this paper is introduced in Sect. 3. Section 4 describes the UML profile for the publish/subscribe paradigm that we constructed, as well as the CPN metamodel used. We illustrate how our transformation tool performs by means of a running example. Section 5 briefly describes the implementation of the tool, including the QVT specification. The verification and validation of the generated TCPN models are explained in Sect. 6. Related work is presented in Sect. 7. Finally, Sect. 8 concludes the paper and provides possible lines of future work.

[1] In this paper, we use CPN interchangeably as a singular and plural acronym.

2 Background

This section first describes the required background on the publish/subscribe paradigm. Then, colored Petri nets are introduced.

2.1 The Publish/subscribe paradigm

Web service systems consist of a collection of services, resources, and clients. Clients and services are interacting with each other, so they are the *participants* in the system. Resources are published and managed by their corresponding Web services, according to the WSRF standard. Hence, the services provide a set of operations for the other participants to access and manipulate the resources.

These participants have their own local variables, whose values or properties range over specific domains. Variable values can be assigned, read, or checked in guards, whereas resources are assumed to have a lifetime and a numeric property that can be read or assigned.[2] Resources must be published before they are available for the participants. Before using a resource, a participant must first discover the resource. Once the resource is discovered, the participant is ready to use it. Furthermore, the participants can subscribe to resources, so they will be notified and some actions will be performed when subscription conditions related to the resource property value hold.

The behavior of the system interaction is then defined by using sequence diagrams, extended with Combined Fragments (CF). Specifically, we use the following basic control structures: parallel (*par*), strict sequencing (*strict*), guarded choice (*alt*), and iteration (*loop*). These are the most relevant operators, capturing the main control structures usually considered in the scope of system workflows, and rich enough to describe the general workflow of interactions among the participants and the resources being used.

On the other hand, taking as reference the algebraic syntax defined in Valero and Cambronero [12], participants can also execute local actions, such as variable assignments or time delays.

In this paper, we assume that WS-Resources are published during the system deployment (*publish* operation). WS-Resources contain different properties. A *textual tag* serves to identify the specific WS-Resource type, so the clients or other services that need to use a WS-Resource of such a class can invoke a *discover* operation indicating this tag. An End Point Reference (EPR)[3] must also be indicated to identify the resource, as well as its initial property values and lifetime. There can be several distinct implementations of a WS-Resource (e.g., a printing service may be offered using different printers), so the discovery mechanism will only return the EPR of one of them.

We have also operations to get or set the resource property values and a *subscription* operation. As previously mentioned, this operation can be used by a participant to perform some actions when the resource property value fulfills a certain condition. As actions, a reference to a UML sequence diagram can be used. As conditions to be fulfilled, a numeric interval is indicated so the actions enclosed in the corresponding UML sequence diagram are performed when the resource property takes a value in this interval. Furthermore, subscriptions have also a lifetime, so they are removed once its lifetime expires.

2.2 Colored Petri nets

Colored Petri nets [5,13] are a well-known formalism for the design and analysis of concurrent systems. CPN are supported by CPN Tools [14], which is a tool that allows us to easily create, edit, simulate, and analyze CPN. In the following, the reader is assumed to be familiar with the basics of Petri nets. First, we give an informal introduction to Petri nets and Colored Petri nets. Next, we provide a formal definition of the CPN formalism. For a complete description of the CPN formalism, the reader is referred to Jensen and Kristensen [13].

Petri nets [15] are a mathematical and graphical formalism that easily represent common characteristics of computer systems such as branching, sequencing, or concurrency, to name a few. Roughly speaking, a Petri net is a bipartite graph of places and transitions joined by arcs, describing the flow of a system with concurrency and synchronization capabilities. Graphically, places are represented by circles, transitions by rectangles, and arcs are represented by directed arrows. An arc can have an integer inscription, indicating the *weight* of the arc. A place can hold tokens, graphically represented by black dots or by a number inside the place and denoted as the *marking* of the place. When all input places of a transition t are marked with a number of tokens equal or greater than their weights, the transition t is said to be *enabled*. An enabled transition can *fire*, yielding to a new marking obtained from removing tokens from input places and setting tokens in output places. The number of tokens removed/set from/in each place corresponds to the arc weight connecting every place with the transition.

A CPN is an extension of Petri nets, in which places have a *color set* (a data type) associated with them that specifies the set of allowed *token colors* at this place. That is, each token in a place of a CPN has an attached data value *(color)* to it that matches the corresponding color set of the place. For instance, a place can have as color set the set of integer

[2] Without loss of generality, in this paper we assume a single Real property.

[3] An integer value that unequivocally identifies each published resource.

numbers INT, the untimed color set of the Cartesian product INT2 = INT × INT, or a singleton color set (UNIT), which contains a single value (), denoted by *unit*. Other complex data types can also be defined by using data types constructors, such as *list, union*, and *record*.

Timed Colored Petri nets (TCPN) [13] are a timed extension of CPN, in which there is a global clock that represents the total time (either discrete or continuous) elapsed in the system model. In this paper, we consider a discrete time scale, since as mentioned in Baeten and Middelburg [16], measuring time on a discrete time scale means that time is divided into slices, and timing of actions is done with respect to the time slices in which they are performed. Actually, computers measure time by means of discrete clocks, and if they are used to control a physical system, the state of the physical system is sampled and adjusted at discrete points in time.

The inclusion of time and data makes that the classical properties on PNs (e.g., reachability, liveness, and deadlock freeness) become undecidable in the TCPN model. In addition, state space exploration usually leads to infinite state graphs, because now the nodes in these graphs are timed markings. Thus, the analysis of properties must usually be done by simulations, i.e., properties are checked by executing the model with different initial markings and then drawing conclusions from the experimental results.

Color sets can then be timed or untimed in a TCPN. Therefore, tokens from timed color sets have a timestamp, indicating the time at which they will be available for the firing of transitions. Tokens from untimed color sets do not carry any time information and they are always considered available.

Let us illustrate this by means of a running example. Figure 1 shows a TCPN as it is presented in CPN Tools. Specifically, it consists of four places (with timed and untimed color sets) and two transitions. Places $p1$ and $p2$ have INTT as timed color set (`INT timed`, as specified by the color set annotation in the left corner of the figure). Similarly, $p3$ has INT2 as untimed color set, whereas $p4$ has the untimed singleton color set, expressed by UNIT (this color set is not explicitly indicated in CPN Tools, since all places have UNIT as color set by default). In CPN Tools, the current number of tokens on every place is drawn at the top right-hand side of the graphical place representation, while the color set of the place is drawn at the bottom right-hand side. The specific color of the current tokens in a place are described by the notation $n`v@s$, meaning that there are n instances of color v with *timestamp* s. A symbol '+++' (respectively, '++') is used to represent the union of timed (resp. untimed) colors in CPN Tools. Thus, place $p1$ has one token with value 1 and timestamp 2, and two tokens with value 2 and timestamp 5.

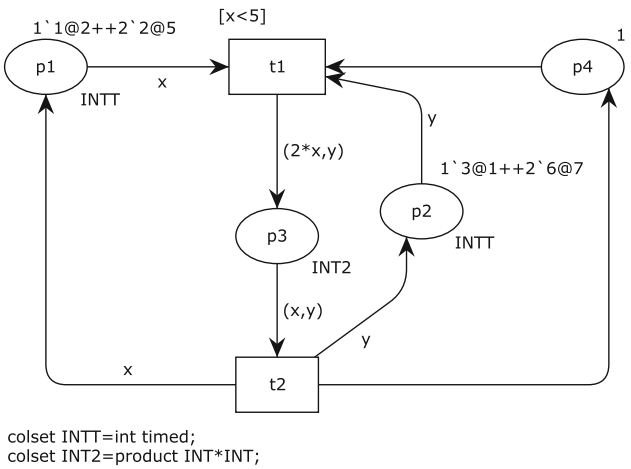

Fig. 1 Graphical view of a TCPN in CPN tools

In CPN, arc inscriptions are now *arc expressions*, constructed using variables, constants, operators, and functions. Arc expressions must evaluate to a color or multiset of colors in the *color set* of the attached place. For instance, the integer variable x in the arc connecting the place $p1$ with the transition $t1$ can evaluate either to the token with value 1 or one of the tokens with value 2, since x is an INT variable.

Enabling of transitions is then redefined in CPNs. For any transition t with variables $x1, x2, \ldots, xN$ in its input arc expressions, a *binding* of t is an assignment of concrete values to these variables, which are used to evaluate the input and output arc expressions of t. A binding of a transition t is *enabled* if there are tokens in its precondition places matching the obtained values of the corresponding arc expressions. Thus, arc expressions are evaluated by assigning values to the variables and those values are then used to select the tokens that must be removed or added when firing the corresponding transition.

Furthermore, transitions can have guards that can restrict their firing, as well as priorities. Guards are predicates constructed by using the variables, constants, operators, and functions of the model. A guard must evaluate to true with the selected binding for the transition to be *fireable*. Transitions can also have a priority. In the event of a conflict between two transitions that can be fired (executed) at a given time, the transition with the highest level of priority is fired first, where smaller values of priority correspond to higher levels of priority. Obviously, we can also have conflicts that cannot be resolved by priorities. In that case, the transition to be fired is non-deterministically chosen.

In TCPNs, we can also include delays associated either to output arcs or to transitions, which are used to age the time stamps of the tokens produced at the output places with respect to the current time. Hence, an enabled and fireable transition in a TCPN model means that: the transition is binding enabled, its guard evaluates to true with the selected binding, and the tokens selected for the firing from its precondition places are available, i.e., their timestamps are less

than or equal to the current model time. In addition, the transition can only fire if there is no other enabled transition with a higher level of priority.

The semantics of firing a TCPN is similar to firing a Petri net, but considering color sets: When an enabled transition is fired, new tokens are generated at the output places, with colors according to the corresponding output arc expressions, and the selected tokens for its firing (from the binding) are removed from its input places. As previously mentioned, tokens on timed places are only available at the time they have attached. This time will therefore determine the instant at which a transition will be able to use these tokens for its firing. When there are no enabled transitions at the current instant, the global clock is advanced to the earliest time at which a transition is enabled.

Let us now introduce the TCPN model in a more formal way.

Definition 1 *(Timed Colored Petri Nets)* We define a Timed Colored Petri Net (TCPN) as a tuple (P, T, A, V, G, E, π), where[4]:

- P is a finite set of *places*, with colors in a set Σ, which can be either timed or untimed. We denote the color set of place p by Σ_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*. PT-arcs are those connecting places with transitions ($P \times T$), while TP-arcs connect transitions with places ($T \times P$).
- V is a finite set of *typed variables* in Σ, i.e., $Type(v) \in \Sigma$, for all $v \in V$.
- $G : T \longrightarrow EXPR_V$ is the *guard function*, which assigns a Boolean expression to each transition, i.e., $Type(G(t)) = Bool$.
- $E : A \longrightarrow EXPR_V$ is the *arc expression function*, which assigns an expression to each arc. Arc expressions evaluate to multisets of the color set of the place connected to the arc. For any transition $t \in T$, the arc expressions of the PT-arcs connected to t are called *PT-arc expressions of* t (respectively, for TP-arcs). In the case of timed color sets, the arc expressions can indicate a delay for the time at which the tokens will be available, with the syntax $ms@+x$, where ms is the multiset of tokens and x the time delay.
- $\pi : T \longrightarrow \mathbb{N}$ is the *priority function*, which assigns a priority level to each transition. We use levels $P1$, $P2$, $P3$, and $P4$, where $P1$ is the greatest priority level.

In this definition, $EXPR_V$ denotes the expressions constructed using the variables in V, with the same syntax admitted by CPN Tools. □

Definition 2 *(Markings)* Given a TCPN $N = (P, T, A, V, G, E, \pi)$, a marking M is defined as a function $M : P \longrightarrow \mathcal{B}(\Sigma)$, such that $\forall p \in P,\ M(p) \in \mathcal{B}(\Sigma_p)$, i.e., the marking of p must be a multiset of colors in Σ_p (which can be empty).

A marked TCPN (MTCPN) is then defined as a pair (N, M), where N is a TCPN, and M a marking of it. □

We define the semantics for MTCPNs as in Jensen and Kristensen, [13], taking into account that transitions have associated priorities. We first introduce the notion of *binding*, then the *enabling condition* and finally the *firing rule* for MTCPNs.

Definition 3 *(Bindings)* Let $N = (P, T, A, V, G, E, \pi)$ be a TCPN. For any transition t, $Var(t)$ denotes the set of variables that appear in the PT-arc expressions of t. Then, a *binding* of a transition $t \in T$ is a function b that maps each variable $v \in Var(t)$ into a value $b(v) \in Type(v)$. $B(t)$ will denote the set of all possible bindings for $t \in T$. For any expression $e \in EXPR_V$, $e\langle b\rangle$ will denote the evaluation of e for the binding b. A *binding element* is then defined as a pair (t, b), where $t \in T$ and $b \in B(t)$. The set of all binding elements is denoted by BE. □

Definition 4 *(Enabling condition)* Let $N = (P, T, A, V, G, E, \pi)$ be a TCPN and M a marking of it. We say that a binding element $(t, b) \in BE$ is *enabled* at the current time at marking M when the following conditions are fulfilled:

1. The guard of t is evaluated to true for binding b: $G(t)\langle b\rangle = true$.
2. For all $p \in {}^\bullet t$, $E(p, t)\langle b\rangle$ is included in $M(p)$, and these tokens on $M(p)$ have a timestamp less than or equal to the current time, i.e., we have in $M(p)$ enough available tokens to fire t with the binding b.
3. There is no other binding element $(t', b') \in BE$ fulfilling the previous conditions such that $\pi(t') < \pi(t)$.

Time can only elapse when there is no enabled binding element for the current time. In this case, time elapses to the earliest time at which some transition can be fired. □

Definition 5 *(Firing rule)* Let $N = (P, T, A, V, G, E, \pi)$ be a TCPN, M a marking of N, and $(t, b) \in BE$ an enabled binding element at marking M.

The firing of (t, b) has the following effects on M:

- For any $p \in {}^\bullet t$, the tokens in $E(p, t)\langle b\rangle$ are removed from $M(p)$.

[4] We use the classical notation on Petri nets to denote the precondition ${}^\bullet x$ and postcondition $x^\bullet$ of both places and transitions: $\forall x \in P \cup T$: ${}^\bullet x = \{y \mid (y, x) \in A\}$; $x^\bullet = \{y \mid (x, y) \in A\}$

– For any $p \in t^{\bullet}$, the tokens in $E(t, p)\langle b \rangle$ are produced in $M(p)$.

□

Example 1 Let us illustrate how the execution of a TCPN works. Consider the TCPN depicted in Fig. 1. In this TCPN places $p1$ and $p2$ have the *INTT* color set (timed integer), $p3$ has INT2 as color set ($INT \times INT$), and $p4$ has the *UNIT* color set (untimed with no information). Place $p1$ has three tokens at the initial marking, one with value 1 and available at time 2, and two tokens with value 2 available at time 5. In the same way, place $p2$ has also three tokens, one with value 3 available at time 1, and two tokens with value 6 available at time 7. Place $p3$ is initially empty and $p4$ has one token. This is usually written using vector notation as follows:

$$M_0 = (1`1@2 + + + 2`2@5, 1`3@1 + + + 2`6@7, \emptyset, 1)$$

According to the token timestamps in $p1$ and $p2$, transition $t1$ can be fired at time 2, by using the token on $p1$ with value 1, the token on $p2$ with value 3, and the token on $p4$. Hence, the corresponding binding for this firing is $x = 1$, $y = 3$. Once $t1$ is fired, these tokens are removed from their places and a new token is obtained in $p3$, with color $(2x, y) = (2, 3)$. Then, the marking at time 2, after the firing of $t1$ is:

$$M_1 = (2`2@5, 2`6@7, 1`(2, 3), \emptyset)$$

Now, the only transition that is fireable is $t2$, at time 2, too, because $p3$ has an untimed color set. The binding now is $(x = 2, y = 3)$, so the firing of $t2$ produces the following marking:

$$M_2 = (1`2@2 + + + 2`2@5, 1`3@2 + + + 2`6@7, \emptyset, 1)$$

Thus, we can intuitively see that transitions $t1$ and $t2$ fire alternatively in this TCPN until the following marking is reached:

$$M = (1`8@2 + + + 2`8@3, 1`3@3 + + + 2`6@5, \emptyset, 1)$$

Given this marking, no further transition can be fired. Hence, we have reached a *dead marking*. □

3 Methodology

In this section, we describe our proposed methodology to build timed colored petri nets from UML systems annotated with the publish/subscribe profile. This methodology consists of three phases, as sketched in Fig. 2 (the scope of this paper is enclosed with a dotted line).

The first phase devotes to the analysis and design phase using UML. In this phase, we consider three different UML diagrams that cover the static and dynamic behavior of the system, as well as all actors involved in the publish/subscribe system under consideration. In particular, a UML deployment diagram (UML-DD) is used to describe the resources and services, a UML class diagram (UML-CD) is used to represent the clients, and a UML sequence diagram (UML-SD)

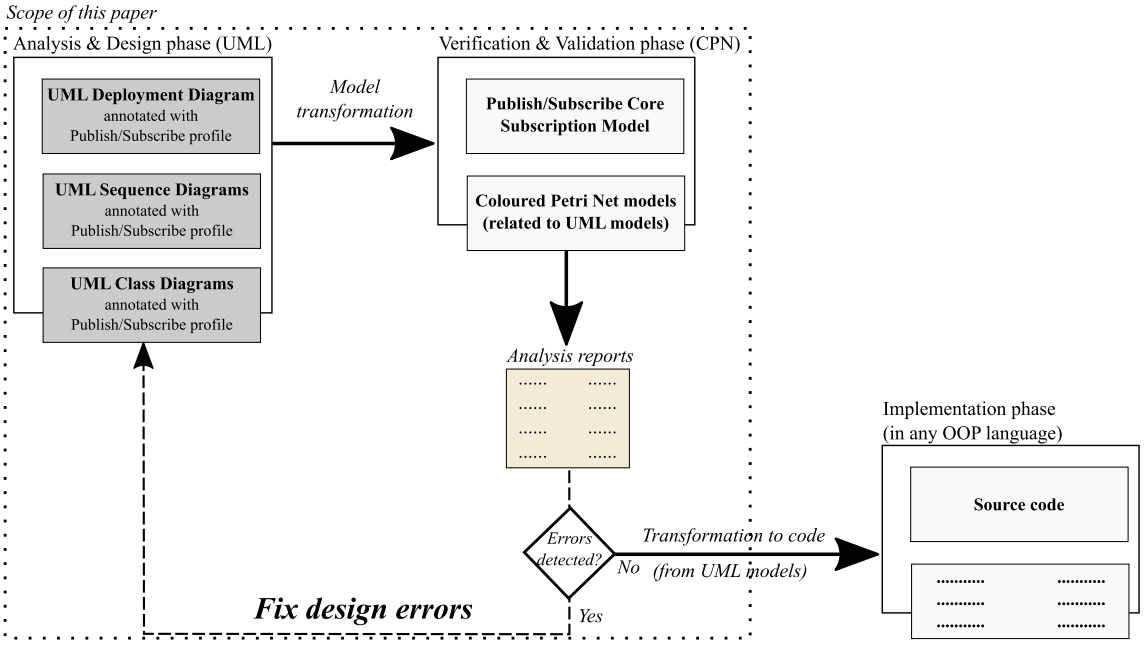

Fig. 2 UML-WSRF to CPNS methodology

is used to represent the interactions between clients, services, and resources. As a final step, these UML models are automatically transformed into colored petri nets using the model transformation that we introduce in Sect. 4.3, which are similar to the approaches given in Bernardi et al. [17] and Distefano et al. [18].

The second phase refers to the verification and validation phase. Here, the TCPN models obtained after transformation are first syntactically checked by using some OCL rules in order to guarantee their correctness, after which they are validated by using CPN Tools, and specifically the monitor features, which allow us to validate some properties of TPCN models in an easy way. General properties of the publish/subscribe paradigm can thus be checked at this point by simulations, using monitors defined on the obtained models. For instance, we can check that "a resource whose lifetime expires is not available any more, and all subscriptions to this resource have also been removed". We can also check that "*an active subscription whose condition holds is immediately notified*", which means that the associated actions are performed as soon as the subscription condition is met.

Let us remark that in this paper we focus on the analysis, design, verification, and validation phases. As future work, we aim at extending our work with the possibility of coming back to the UML model from errors detected at the validation phase by using the CPN Tools monitors features. This feedback to the user at the UML models would allow the user to edit the UML specification to fix these errors and generate again the corresponding TCPN model. This point requires further research in order to link the properties with the UML models in an automatic way, specially in the case of properties that are specifically defined for a particular scenario. Moreover, currently our tool is integrated with the CPN Tools, which provides simulation and analysis capabilities, but always through the use of its graphical interface.

The last phase in the figure concerns implementation, which is also out of the scope of this paper. Once we are aware that the UML models are correct, several techniques can be used to transform UML models directly into code [19]. This code can later be extended to implement other system aspects not considered by the PS paradigm.

4 UML profile for the publish/subscribe paradigm

In this section, we first introduce the UML domain model and the profile that leverages the Publish/Subscribe paradigm (described in Sect. 2.1) we propose to annotate UML models. Then, we describe the metamodel used for CPN and the model transformation patterns from UML models annotated with our profile to timed colored petri nets.

4.1 Description of the profile

The Unified Modeling Language (UML) [4]—a standard modeling language in the software development industry—is a powerful language that allows representing from architectural to behavioral aspects of systems.

UML can be tailored for specific purposes through *profiling* [6]. A UML profile provides a set of stereotypes and tagged values that are added into UML models to extend its semantics. To build a UML profile, a domain model shall be modeled in first place. This model captures all intrinsic characteristics of the domain under consideration. In our case, we defined a domain model for the PS paradigm. Then, following the rules given in Lagarde et al. [7] and Selic [6], a UML profile conformed by a set of stereotypes and its tags is obtained. Stereotypes define concepts in the domain under study, while tags are the attributes of a stereotype.

For instance, the modeling and analyzing of real-time embedded systems (MARTE) UML profile—actually, a standard promoted by the Object Management Group [20]—enables schedulability and performance analysis for real-time and other application domains. A specialization of MARTE, the non-standard dependability analysis and modeling (DAM) UML profile [21], enables to express dependability issues in UML models. Similarly, the non-standard security analysis and modeling (SecAM) profile [22] allows to express security characteristics into UML models.

Figure 3 depicts the UML domain model for the publish/subscribe paradigm. A *Service* can publish one or more *Resources*, while a *Resource* can only belong to a *Service*. A service can update both the lifetime and value of its resources (association class *TimedSetter*), in which both the operation to be applied over the resource value and a new expiring time are indicated. This operation can also be delayed as indicated by the optional argument *Delay*. A *Client* can perform different actions on a *Resource*, such as subscribing to the resource (specifying the minimum and maximum value of interest to the client and the subscription time, association class *Subscription*), getting the resource value and storing it into a variable (association class *Getter*), or updating the resource value (association class *Setter*). Finally, a *Client* can also assign values to local variables (association class *Assignments*).

UpdateOperation and *AssignmentOperation* are defined as complex data types. *UpdateOperation* has two attributes, *operator* and *value*. The *operator* values are specified as an enumeration type (*SignKind*), which consists of the arithmetic operators that allow us to update a resource value. *AssignmentOperation* has also two attributes, *property* and *value*.

The corresponding UML profile that maps the contents of the publish/subscribe domain model is depicted in Fig. 4. Since we need to identify the UML sequence

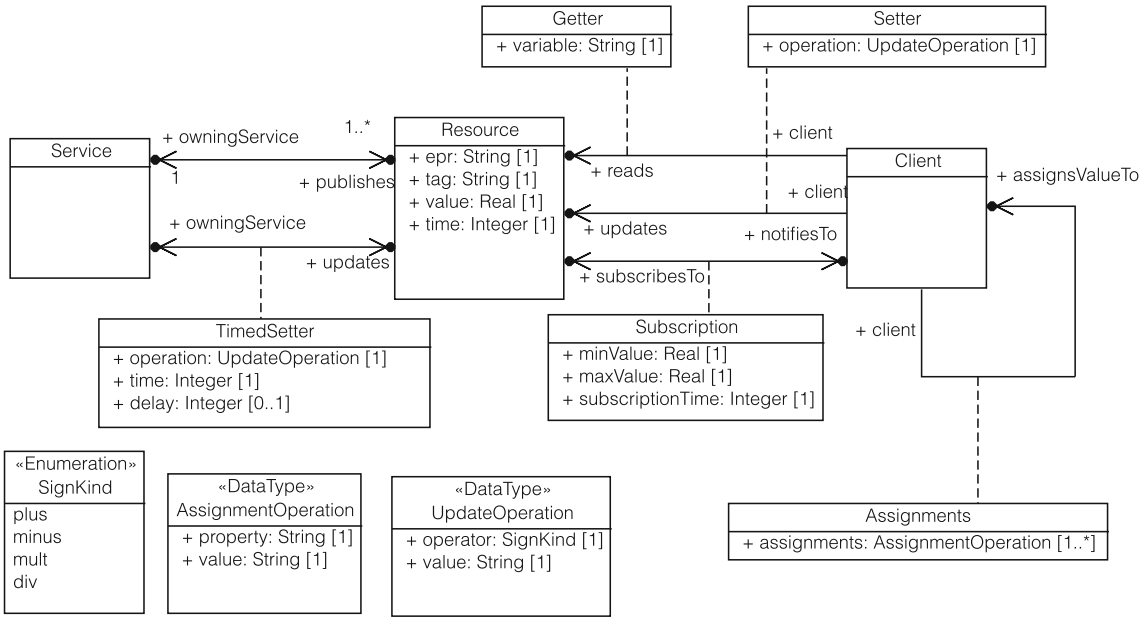

Fig. 3 UML domain model for publish/subscribe paradigm

diagram (UML-SD) scenarios in which the publish/subscribe paradigm is being used, we have incorporated the `PublishSubscribeScenario` stereotype that extends the *Interaction* metaclass. This stereotype indicates the UML-SD in which the subscription from a client to some resources are expressed.

Service, *Resource*, and *Client* classes have been modeled with stereotypes. The `Service` stereotype extends the *Node* UML metaclass, which belongs to the *Deployments* UML package. The `Resource` stereotype extends the *Artifact* UML metaclass (which also belongs to the *Deployments* UML package). Both stereotypes are related through an association, indicating that a `Service` may publish an arbitrary number of resources, while a resource only belongs to a single service. To verify the correctness of the annotated UML model, we added an OCL expression [23] into the `Resource` stereotype to check whether the container of the UML artifact is indeed stereotyped with `Service`. Finally, the `Client` stereotype extends the *Lifeline* metaclass, from the *Interactions* UML package.

The association classes related to the interactions between services and clients with resources have been modeled as stereotypes extending the *Message* metaclass. The tagged values of these stereotypes match with the attributes of the corresponding associated classes. The optional attribute *delay* of the class *TimedSetter* has also been transformed to a stereotype, also extending the *Message* metaclass. Furthermore, we have also included an `AbstractSetter` stereotype, which defines the *operation* attribute as an `UpdateOperation` complex type. This complex type is used to update the value of a *Resource* and thus it defines a tuple with the operation to apply (indicated by *operator*) and the value of interest (indicated by *variable*).

Finally, the association class *Assignments* has been modeled as the `AssignmentExecution` stereotype, which extends the `ExecutionSpecification` metaclass and enables us to indicate the initial assignment of `Clients`' attributes.

4.2 A metamodel for Colored Petri nets

Figure 5 shows the metamodel for Timed Colored Petri nets, which has been defined by extending a previous work [24]. The metamodel has been designed in such a way that it captures all the specifities of CPN Tools. This design poses two main advantages (at the cost of being tied to this specific tool): first, it allows us to deal with all the interesting concepts of this simulation and analysis tool from a single M2M transformation, without the need of any kind of preprocessing; and second, it eases the serialization of the final XML file that needs to be fed into the analysis and simulation tool, because the metamodel is a close representation of the schema that native CPN Tools net files must validate.

`Cpnet` is the main class of the model (see Fig. 5), and elements in the model can be differentiated into two groups. On the right-hand side of the figure, and contained within the `Globbox` class, we find the elements to declare *color sets*, *variables*, etc., while on the left-hand side of the figure, and contained within a `Binder`, we find the elements that visually represent a Petri net.

More specifically, declarations can be grouped in nested `Blocks`, and can be formed by *color sets* (`ColorSet`)

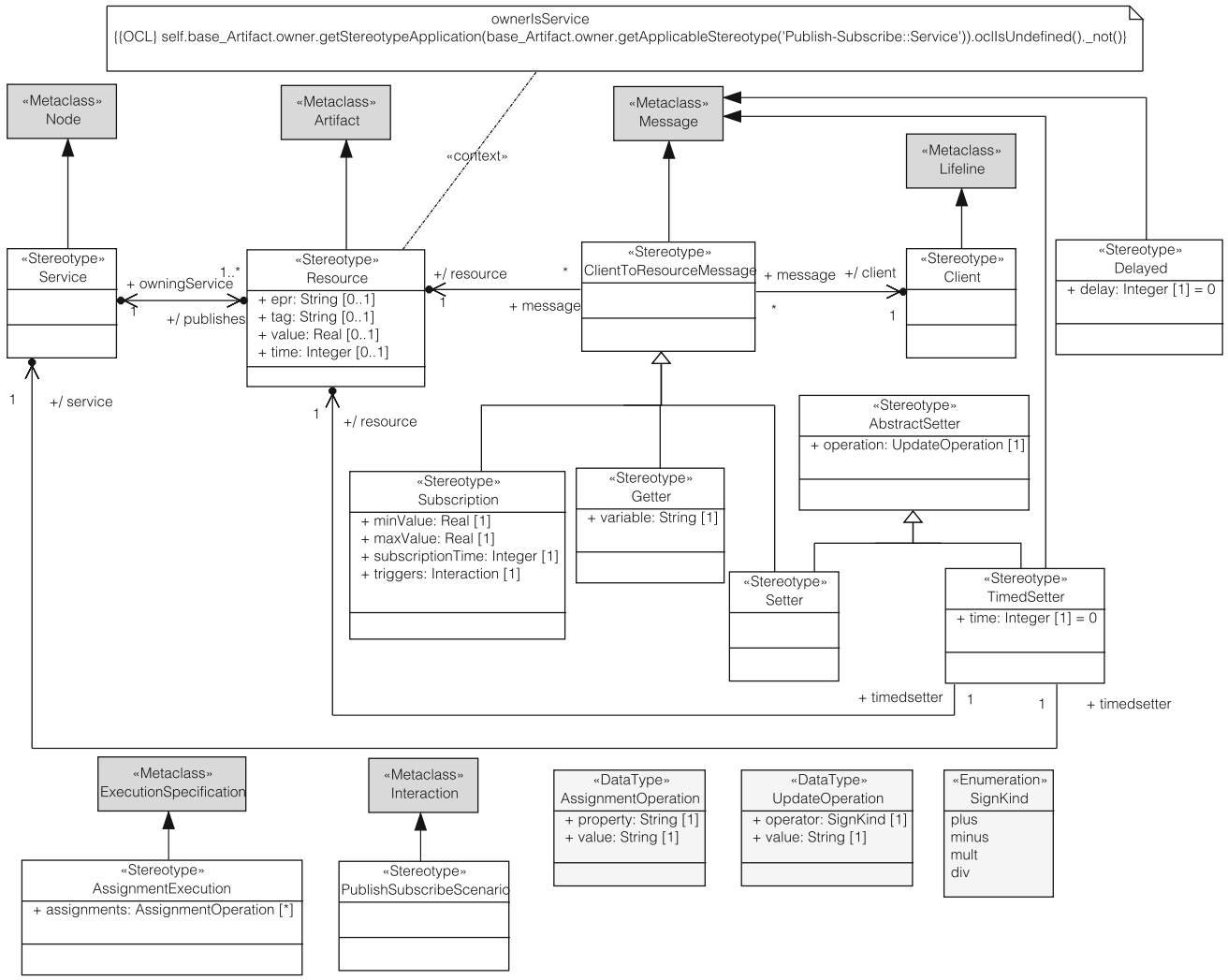

Fig. 4 UML profile for publish/subscribe paradigm

(either simple or compound—`SimpleColorSets` or `CompoundColorSets`, respectively), *variables* (`Var`), *reference variables* (`Globref`), or CPN ML expressions[5] (`Ml`). Examples of `SimpleColorSet` are basic datatypes, such as `Unit`, `Integer`, `Real`, `String`, etc., while for `CompountColorSet` are complex types, such as Cartesian `Product`, disjoint `Union`, etc.

On the other hand, graphical elements are placed inside `Pages`, which may be grouped in `Binders`. All graphical elements inherit from the `DiagramElement` class, and can be organized in different `Groups`. Thus, a `Page` can hold *places* (`Place`), *transitions* (`Trans`), *arcs* (`Arc`), *annotations* (`Annot`), etc. Places must have an associated color set, which must be defined in the declarations part. The relationship between the place and its color set is represented by means of the `type` role from the class `Place` to the class `ColorSet`. The `InitMark` determines the initial marking of a given place, i.e., the tokens owned by the place before starting the simulation. Transitions may have different inscriptions attached, such as *firing conditions* (`TransCond`), priorities (`TransPriority`), and transition delays (`TransTime`). Arcs link a place to a transition and have an `orientation` (from place to transition, or vice versa).

Finally, places can be *fused* via the `Fusion` class. Fused places act as a single place, thus allowing reuse, and making easier the partitioning of a single CPN in different pages (every page contains a CPN model).

4.3 Model transformation: from UML models to CPN

In the following, we detail the transformation of the UML models annotated with the profile into TCPN. We first

[5] CPN Tools uses the CPN ML language to specify declarations and net inscriptions. This language is an extension of the functional programming language Standard ML [25].

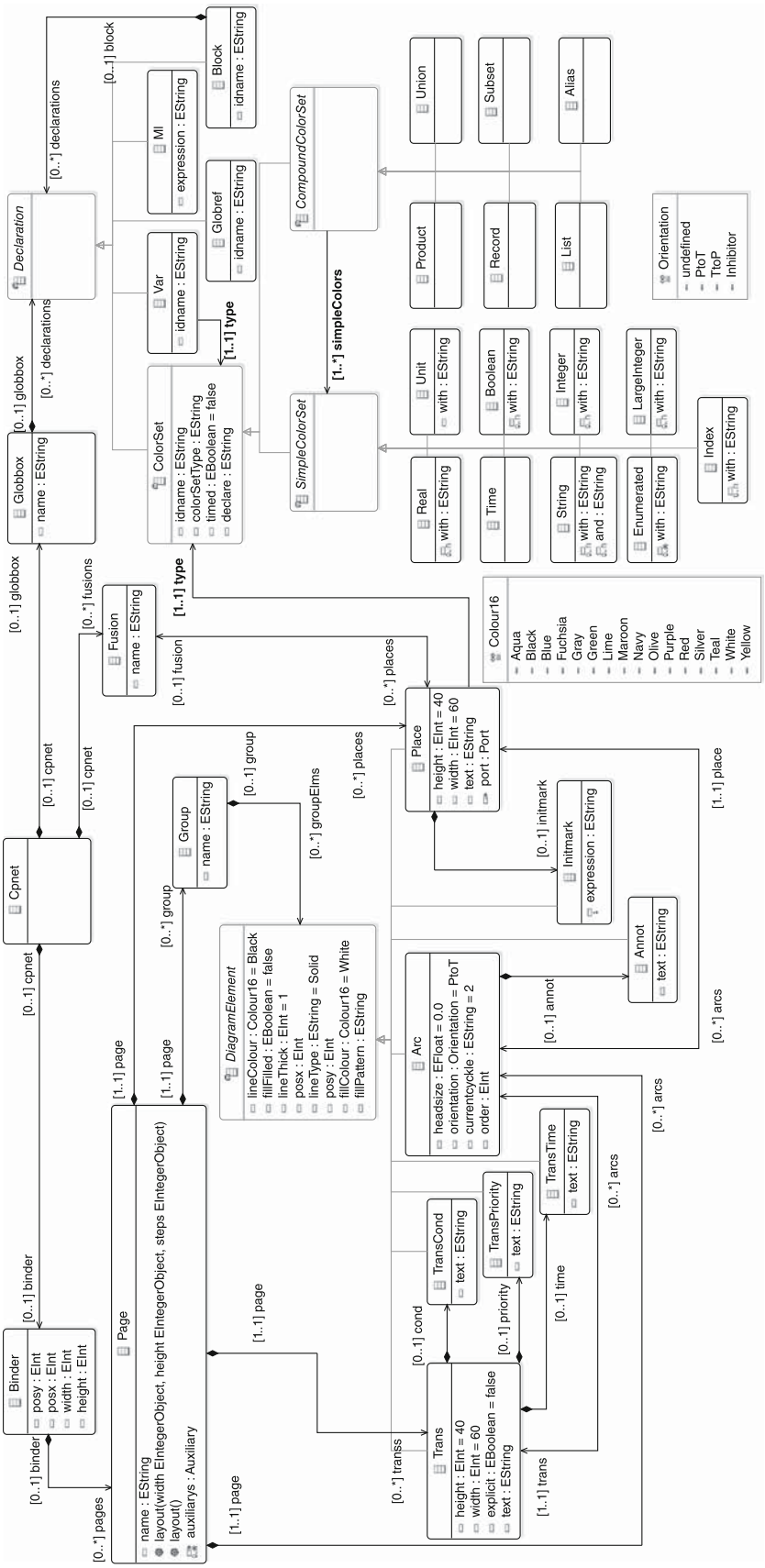

Fig. 5 Colored Petri nets metamodel

describe the UML parts, annotated with the profile, and then how the transformation is carried out for each part. To illustrate the transformation process, we have adapted the online purchase process example already introduced in Valero et al. [26] and used it as a running example. A detailed explanation regarding the patterns of the model transformation is given in https://github.com/abelgomez/publish-subscribe/blob/master/plugins/io.github.abelgomez.ps.transformer/doc/README.md.

4.3.1 PS core net

Figure 6 illustrates the TCPN of the PS core net model. This core net rules the resource publication, resource subscription, notifications, and resource expiration. The PS core net model generated by our model transformation is (slightly) simplified with regard to the previous model introduced in Valero et al. [26]. This modification was needed to cope with all automatic transformation contributed in this paper an later explained for the following stereotypes. A single core net is generated for each UML sequence diagram stereotyped as `PublishSubscribeScenario`. `PublishSubscribeScenarios` should include the initialization of resources, the subscription messages of the different clients, and the additional interactions which are triggered when notifications are sent. In the core net, the initial marking of the *Roles* and *Resources* places—which are explained below—are the only variable parts.

In this TCPN, the resources to be published are represented by tokens on the *Resources* place, which contain their *EPR*, *tag*, *value* and *lifetime*. Resources are then published by firing the *Publish_ok* transition, but if we try to publish a resource with an existing *EPR*, this operation fails (by firing *publish_fail*). Published resources are written to the *ResourceRegistry* place. Clients' behaviors are represented by tokens on place *Roles*, where we indicate a client's identifier, a resource tag and the subscription conditions for the indicated resource. The *Discover* transition is then fired to find published resources and write the corresponding subscription conditions into the *SubscriptionRequest* place. The *Subscribe* transition is then fired to submit the subscriptions, which are written on the *SubscriptionRegistry* place.

When the lifetime of a resource expires, the *ResourceExpire* transition is fired, which removes the resource token from *ResourceRegistry*, as well as its current subscriptions (transition *RemoveSubscription*). Furthermore, subscriptions can also expire. In that case, the *SubscriptionTime-Out* transition will be fired, thus removing the corresponding token from *SubscriptionRegistry*. Finally, notifications occur as soon as the associated conditions hold, which is captured by transition *Notify*, which has the greatest priority (P1).

4.3.2 Publish/Subscribe scenarios

Figure 7 (left side) illustrates the UML-DD (left side) and the UML-CD (right side, annotated with the `PublishSubscribeScenario` stereotype) of the running example. There exists a client who is willing to buy a laptop using her credit card. She is waiting for a good offer due to her limited amount of money, though. The artifact *CR* in the UML-DD diagram represents her current budget, as maintained by her personal bank. She also disposes of some cash, as indicated by the attribute *m* of *Client* class. Surfing on the Internet, the client finds two offers from two different online shops. However, both laptop prices still overrun her budget. Services *Shop1*, *Shop2*, and resources *L1*, *L2*, represent the online shops and the laptops, respectively.

Figure 8 depicts the actions of the client with the resources in a UML-SD annotated with `PublishSubscribeScenario` stereotype. Consider that the client has some cash (for instance, 2000€). Since she wants to pay by credit card, she first decides to subscribe to her bank deposit during 1 year to let her know when her credit is lower than 1000€ and then make a deposit to increase it (triggering UML-SD *Deposit*). After that, she decides to subscribe to both resources during 1 year to keep posted as soon as some offer in the laptop price comes up (triggering UML-SD *Purchase L1* and *Purchase L2*, respectively).

A UML-SD stereotyped with `PublishSubscribeScenario` creates a place *Start Subscription* and two transitions (*Acquire Locks* and *End Subscription*) into the generated TCPN. Additionally, every `Client` and `Resource` lifelines have an associated place which is used to avoid race conditions issues when handling client/resources attributes.

These places are in fact *fusioned* places, i.e., places that act as a single one although they are drawn multiple times in different parts of the CPN. The *Acquire Locks* and *End Subscription* transitions represent respectively the beginning and the ending of the UML-SD. In particular, the first transition also has as input places the client/resource lock places that represent the client/resources involved into the UML-SD. Similarly, the ending transition has as output places the same client/resource lock places, ensuring the conservativeness of the tokens (i.e., the acquired locks are eventually released). The subnet resulting from all the interactions described in the UML-SD lifelines is then enclosed between these two transitions. This internal subnet is built in a compositional way, by applying the rest of the patterns explained in this section. The places that serve to connect these patterns have UNIT as color set.

A `PublishSubscribeScenario` SD should be accompanied by a DD describing the allocation of `Services` and `Resources` (a node *Service1* and an artifact *Resource1* in this case). Note that the association between resources and services is directly taken from the node-artifact relationship.

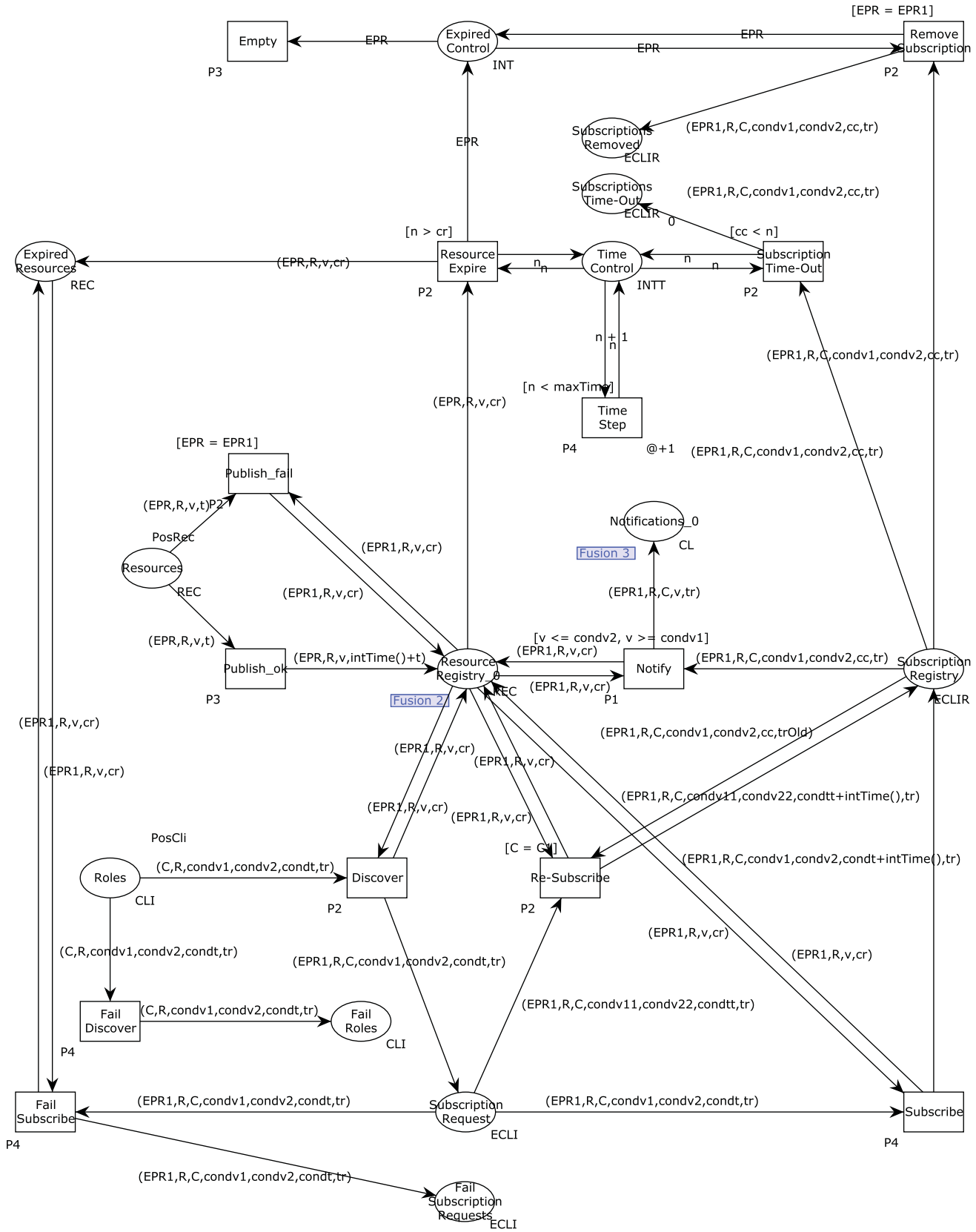

Fig. 6 Timed Colored Petri net of the PS core model (simplified with regard to Valero et al. [26]

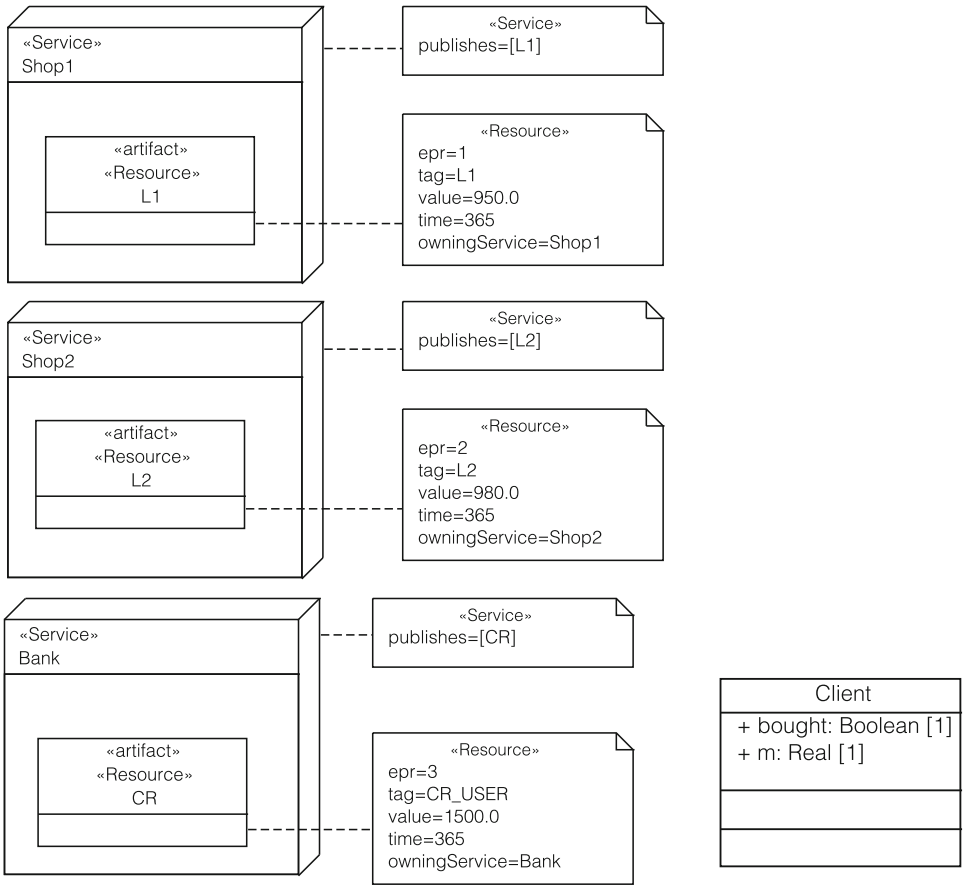

Fig. 7 Running example: UML deployment diagram (left side) and UML Class diagram (right side) of the online purchase process annotated with the PS profile

Finally, let us also note that both *Service1* and *Resource1* are also included in the UML-SD.

As introduced in Sect. 3, in addition to UML-DD, UML-CD are also used to provide the static view of the system under analysis. Recall that Fig. 7 (right side) shows the *Client* class of the running example. In this case, it has two attributes, a boolean-type *brought* and a real-type *m*. The transformation of this part is straightforward. The *Client* class is directly transformed into a timed product color set. The first component is a single string used to unequivocally identify every instance of the class (i.e., to identify the class objects), while the rest of the components are each of the class attributes). For the sake of simplicity, our transformation algorithm also defines a set of variables following the attributes defined within the class (e.g., "var att1: DATATYPE1", "var att2: DATATYPE2", etc.). Therefore, the existence of a lifeline in a UML-SD stereotyped as `Client` is straightforwardly transformed to a colored place having the previously mentioned timed product color set as color set and an initial token with the values of object *Client1*.

The set of variables and color sets in the CPN model generated by the annotations in the UML-SD PublishSubscribe scenario, the UML-CD, and the UML-DD are collected in Listing 1. Note that it also sets the initial marking of the PS core CPN model introduced previously.

Listing 1 Variables and colorsets generated by transformation of Fig. 7.

```
(* Clients declarations *)
    colset CLIENT = product STRING * BOOL * REAL timed;
    var client: STRING;
    var bought: BOOL;
    var m: REAL;
(* Resources declarations *)
    (* L2 *)
        var cr_R2: INT;
        var EPR_R2: INT;
    (* CR_USER *)
        var cr_R3: INT;
        var EPR_R3: INT;
    (* L1 *)
        var cr_R1: INT;
        var EPR_R1: INT;
(* Value declarations *)
    val m1 = 1
    val m2 = 1
    colset V0m1 = int with 0..m1;
    colset V0m2 = int with 0..m2;
    val PosCli=1`("Client","CR_USER",0.0,1000.0,365,"Deposit")@0++
    1`("Client","L2",0.0,850.0,365,"Purchase L2")@0++
    1`("Client","L1",0.0,850.0,365,"Purchase L1")@0;
    val PosRec=1`(3,"CR_USER",1500.0,365)@0++
    1`(1,"L1",950.0,365)@0++
    1`(2,"L2",980.0,365)@0;
    val maxTime=4
```

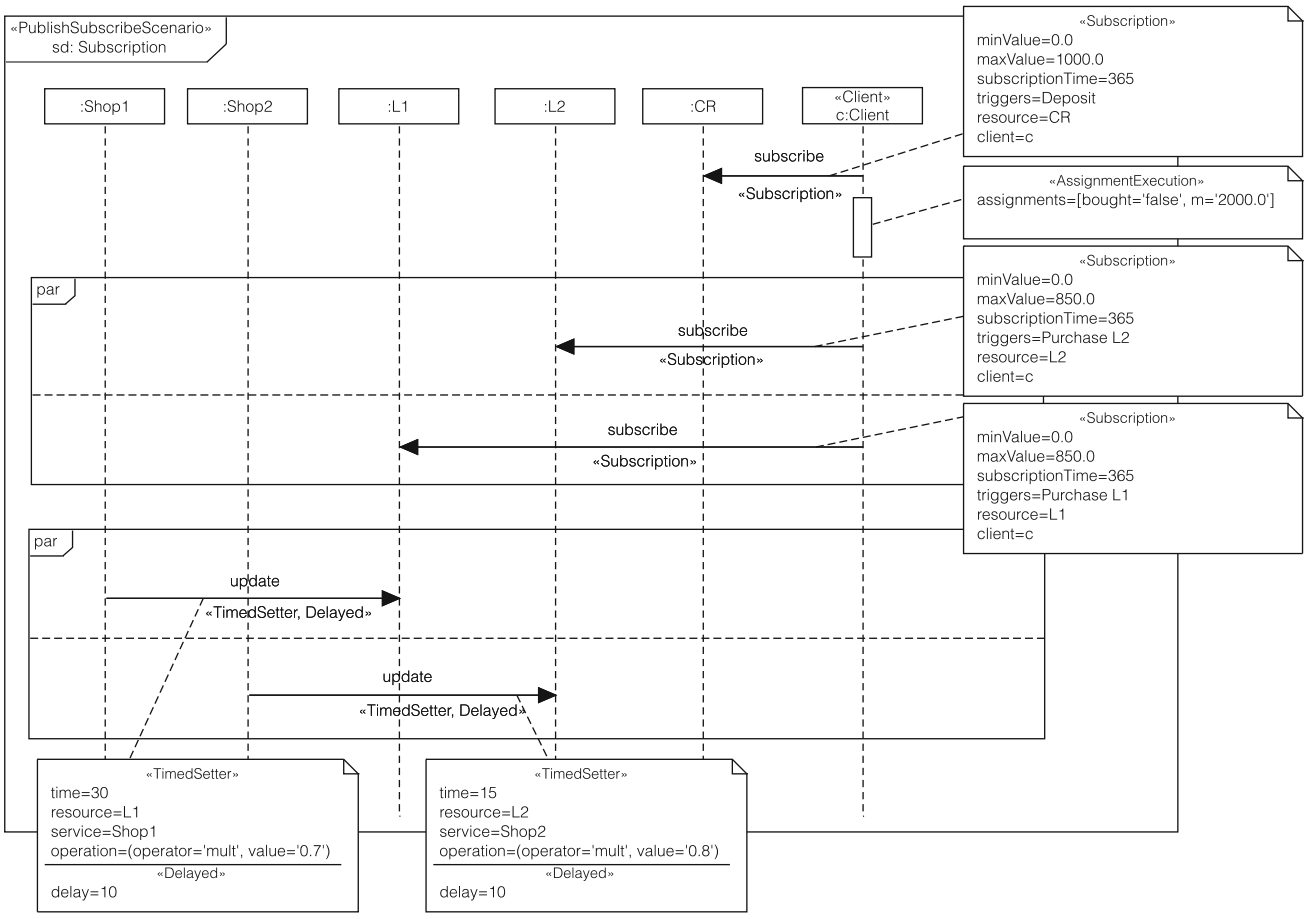

Fig. 8 Running example: UML-SD *Subscription* of the online purchase process annotated with the PS profile

The TCPN generated by our approach is depicted in Fig. 9. The "*Fusion n*" tag in some places of the net means that they are *Fusion* places, and thus they correspond to a same place that is used in several CPN pages. For instance, this is the case of the *Resource Registry* place, which also appears in the PS core net model shown in Fig. 6. In the following, we describe each part of the net considering the transformation of the stereotypes shown in Fig. 8. For the sake of readability, we have added dashed boxes in the figure to highlight from which stereotype comes each part of the net.

The `Subscription` *stereotype.* Once the basic elements have been transformed (core net, client definitions, and external structure of the *Subscription* PN), let us explain the transformation for subscription interactions (`Subscription` stereotype). Consider the resource *L1*, owned by *Shop1* and having *(1, 'RL',* 950.0*,* 365*)* as attribute values. Consider also the *Client* who subscribes to such a resource by means of a `Subscription` stereotyped message, as depicted in Fig. 8 (third subscription message). Note that *Client* subscribes for a time frame of 365 days, triggering the UML-SD named *Purchase L1* once the values of *L1* are in the interval of [0, 850.0]. As shown in Listing 1, both `Resource` and `Subscription` stereotypes are transformed into the corresponding initial colored tokens in the PS core CPN. Let us remark that the `Resource` stereotype serves us to populate the *Resources* place, whereas the `Subscription` stereotyped message serves us to populate the *Roles* place.

The `AssignmentExecution` *stereotype.* This stereotype is used in a UML-SD stereotyped with `PublishSubscribeScenario` to indicate when the attributes of a *Client* object are modified.

Our transformation tool automatically verifies the correctness of the model, i.e., it checks whether every property specified in each `AssignmentOperation` matches to some attribute of the *Client* class. Any datatype error in the specified values is indicated by the tool, indicating a mismatch data type error in the corresponding arc inscription.

Regarding the transformation, every execution specification in the annotated UML-SD generates a branch in the sequential Colored Petri net that represents the execution of the overall UML-SD. Then, each execution specification annotated with `AssignmentExecution` generates a transition for each of the `AssignmentOperation` attributes. This transition is connected to the place that stores the cur-

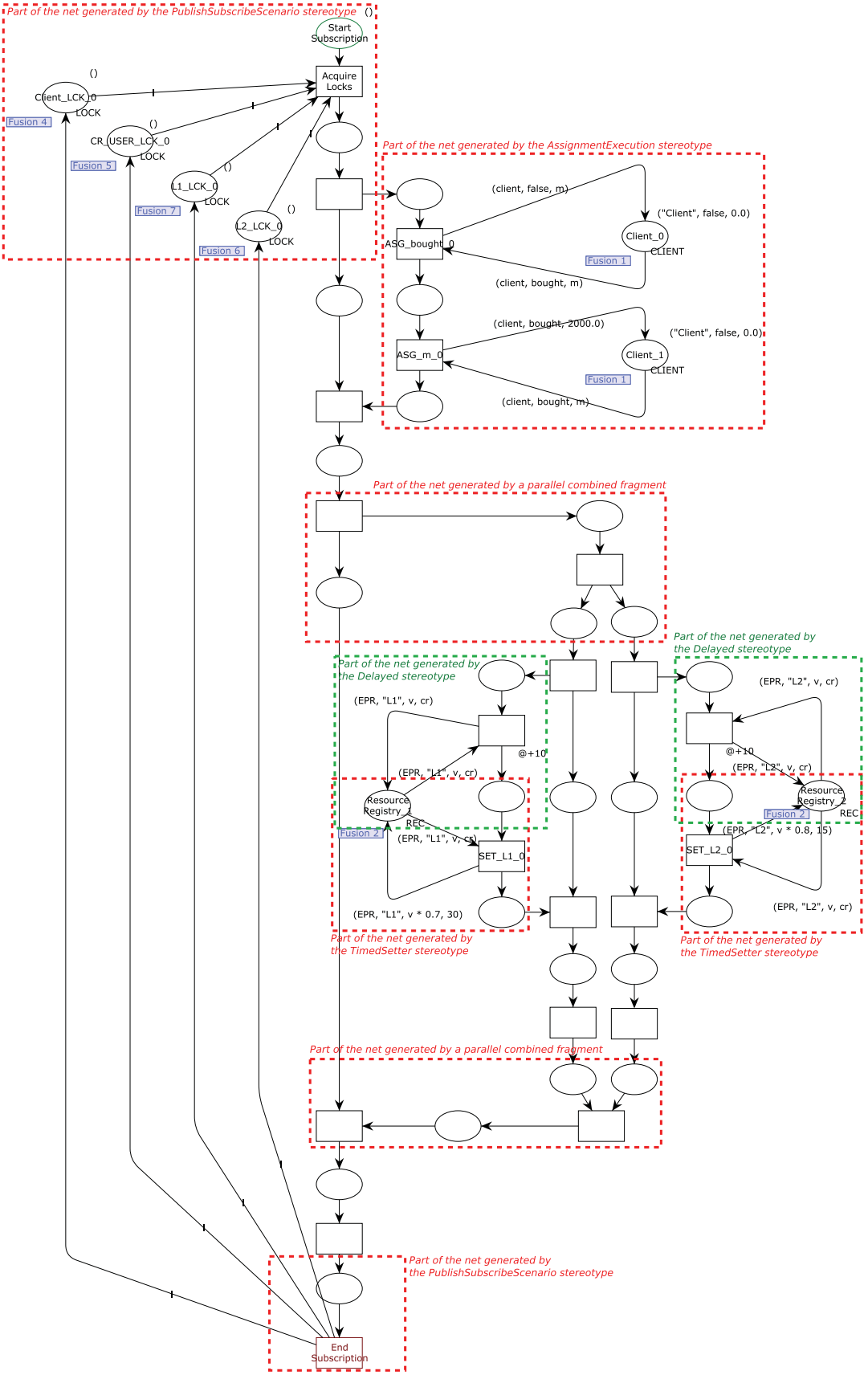

Fig. 9 Running example: TCPN model of UML-SD depicted in Fig. 8

rent status (values) of the object *Client* through an input and output arc. As input arc, it receives a tuple of the form $(client_1, att_1, att_2, \ldots, att_i, \ldots, att_n)$, which represents the particular instance of *Client* and the values of its set of attributes. As output arc, it returns a colored token conformed by $(client_1, att_1, att_2, \ldots, att_i, \ldots, att_n)$. The only value that is modified in the tuple of the output arc inscription is the one specified in the attributes of the stereotype.

For instance, in the UML-SD depicted in Fig. 8 the *Client* assigns the values of *false* and *2000.0* to its attributes *brought* and *m*, respectively. This execution specification, annotated with `AssignmentExecution`, generates the part of the net depicted in the second dashed box (from the top) in Fig. 9, following the aforementioned pattern. Note that the transformation model follows the assignment operations from left to right (that is, first the value of *brought* is set, and then the value of *m*).

Moreover, let us remark that since this action occurs inside a `PublishSubscribeScenario` UML-SD, the *Client* lock was acquired at the beginning (first dashed box) and hence no race conditions arise.

The `TimedSetter` *and* `Delayed` *stereotypes.* Let us recall the running example as depicted in Fig. 8. The client was subscribed to both resources to be aware if some changes were produced in their prices for a year. Let us consider now that a few days later after the subscription (consider for instance 10 days), the sales period begins so both shops simultaneously reduce their laptop prices. In particular, *Shop1* offers a 30% off for *Laptop1* during 30 days, while *Shop2* offers a 20% off for *Laptop2* during 15 days. Note that both resource subscriptions and the update of the laptop prices occur inside a parallel combined fragment. These actions have been annotated with `TimedSetter` and `Delayed` stereotypes in the UML-SD *Subscription*.

Recall that the `TimedSetter` stereotype allows us to update the value of a resource as indicated by the operator attribute. This stereotype also allows us to update the resource lifetime. In the running example, the current value of the resource *L1* is multiplied by 0.7, representing the 30% off of the sales offer. The generated subnet of `TimedSetter` stereotype (red dashed box in the central part of Fig. 9) contains two places and one transition (*SET_L1_0*), which is connected to the *Resource Registry* place. The arc inscription of the arc from the transition to the *Registry* place is used to update the value of the resource, taking into account the operation to perform and the value specified as stereotype attribute values. For instance, the incoming arc of *SET_L1_0* is *(EPR, "L1", v, cr)*, while the output arc is $(EPR, \text{“}L1'', v \times 0.7, 30)$. (See the attribute values of the `TimedSetter` stereotype in Fig. 9.)

The `Delayed` stereotype allows us to specify a delay on a given message. Let us remark that the transformation pattern for the `Delayed` stereotype will be inserted before any other pattern produced by other stereotypes also applied on the same message (see the green dashed boxes in 9). Its transformation follows a similar scheme to the `TimedSetter` stereotype. Recall that the update of the price of *L1* was done 10 days after the client subscription, as specified by the `Delayed` stereotype in Fig. 8. In this case, two places and a new transition connected to them are generated. These new places represent the beginning and ending of the delay operation, and thus they are connected to the sequential CPN representing the evolution of the whole UML-SD. The transition is connected to the *Resource Registry* place through input/output arcs that do not change the resource attributes, but serve us to check that it is already published. The value of the `Delayed::delay` attribute is the time that the new transition generated takes to fire (see the annotation *@+10* under the transition in Fig. 9).

Transformation of parallel combined fragments. Combined fragments are UML structural components whose transformation is not directly linked to any stereotype of the UML profile. Consider the parallel combined fragment of the UML-SD as depicted in Fig. 8 (bottom side). The transformation of this fragment is as follows (third dashed box in the figure). First, a fork transition creates a new branch of execution, separate from the sequential execution of the overall UML-SD, which terminates with a corresponding join transition. This branch corresponds in the figure with the part indicated in the right-hand side of the enclosing box. Notice that two places are also produced as the output of this fork transition. Then, for the new branch a new fork transition splits the execution into the parts indicated in the parallel fragment (two in this case), creating the initial and final places for each one. A join transition is also created in order to link the termination of the parallel branches with the original sequential flow (penultimate dashed box). Then, each part of the parallel fragment is transformed into its corresponding TCPN, following the patterns explained along this section.

4.3.3 Triggered sequence diagrams

Triggered sequence diagrams (*triggeredSDs*) are those UML-SD describing the interactions happening when the *value* of a *Resource* is between the minimum and maximum values specified by a subscribed *Client*. That is, they represent the UML-SD that takes place when the given conditions are fulfilled.

Let us illustrate how the transformation of a triggeredSD is carried out by means of the UML-SD *Deposit*. Recall that this UML-SD was triggered when the credit of the client is lower than 1000€ (see the first `Subscription` message in Fig. 8). The UML-SD *Deposit* of the running example is depicted in Fig. 10, while the TCPN generated is shown in

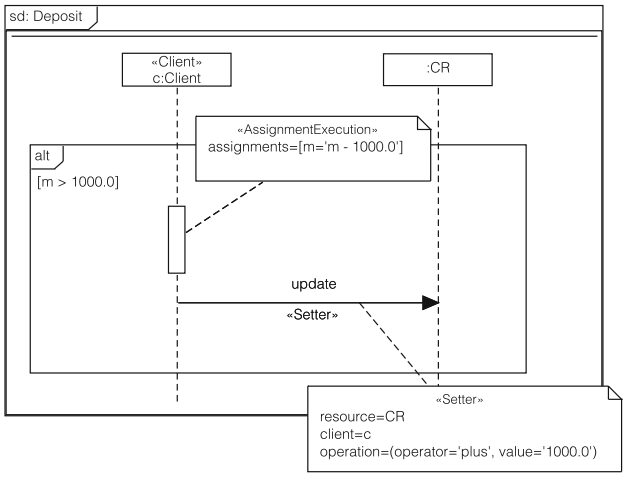

Fig. 10 Running example: UML-SD *Deposit*, annotated with PS profile

Fig. 11. As before, we have highlighted with dashed boxes the interesting parts of the generated net.

The transformation into TCPN is very similar to the one of a UML-SD stereotyped with the `PublishSubscribeScenario`. In particular, two places *Start Deposit* and *Finish Deposit*, and two transitions are created. The first transition works similar to the previous *Acquire Locks* transition: as input arc, it consumes tokens from the *Start Deposit* place, from all places that represent the client and resource locks of the clients and resources which interact in the UML-SD, and from the *Notifications* place. As initial marking, the *Start Deposit* place contains a string-typed token with the "Deposit" value.

The *Notifications* place is then used to activate this subnet when a notification occurs. As an illustration, in our running example the input arc of the initial transition and the *Notifications* place has as inscription the tuple *(EPR, "CR_USER", "Client", v, "Deposit")*, which indicates the participants of the UML-SD stereotyped ("Client" and "CR_USER") and the name of the triggered SD ("Deposit").

Transformation of alternative combined fragments. Notice now the use of an alternative combined fragment in Fig. 10 to specify a conditional flow in a sequence diagram. With this alternative combined fragment, we indicate an interaction between a *Client* and a *CR* that only occurs if the condition $m > 1000.0$ is fulfilled.

The transformation of this fragment is similar to the parallel combined fragment. First, a branch is created by the transition below the *Start Deposit* place, which separates the sequential Petri net that represents the execution of the overall UML-SD from the new branch for the combined fragment (on the right). The part produced for the alternative combined fragment is enclosed in the dashed boxes at the top and bottom of the figure. The combined fragment has its own sequential flow, so a new fork transition is used to separate this flow from the alternatives, which are started by transitions labeled with the corresponding guards. These transitions are connected with the *Client* place so as to obtain the attribute values (*m* and *bought*) required in their guards. Only one of these transitions can fire (see the conflict place in the right-hand side), and they must fire before the transition corresponding to the default condition, so they have a higher priority than the transition representing the default condition.

Finally, each part in the alternative combined fragment is individually transformed into its corresponding TCPN following the rules explained in this section. In our running example the default case is empty, so its transformation is straightforward. Thus, there are two branches in the generated TCPN (see Fig. 11), one branch without any activity (left branch) and the other branch having the condition $m > 1000.0$ (right branch). Furthermore, this branch contains the generated TCPN of the contents shown in the alternative fragment (the transformation of an `AssignmentExecution` stereotype and a `Setter` stereotype, explained later).

The `Setter` *stereotype.* The UML-SD depicted in Fig. 10 also shows a message stereotyped with `Setter`. The *Client* is modifying the value of resource *CR* by operating on its current value (in particular, its value is being incremented in 1000 units).

The transformation of this stereotyped message follows a similar approach to the `TimedSetter` stereotype (see the dashed box in the central part of Fig. 11). The generated subnet contains two places and one transition (*SET_CR_USER_0*), which is also connected to the *Resource Registry* place in order to modify the resource property value.

Let us now see the UML-SD *Purchase L1* depicted in Fig. 12, which corresponds to the triggeredSD for the third client's subscription in the UML-SD *Subscription* (see Fig. 8). The generated TCPN for this UML-SD *Purchase L1* is shown in Fig. 13. In this UML-SD, the *Getter* stereotype is used in order to get the values of the resource properties.

The `Getter` *stereotype.* The `Getter` stereotype is used in the UML-SD to indicate the variable in which the value of a resource is stored. Note that in the UML-SD depicted in Fig. 12 there are two messages stereotyped with `Getter`, getting the values of the resources *L1* and *CR* in the client's variables *PPL1* and *balance*. These variables are then used as part of the guard condition in the subsequent alternative combined fragment.

The transformation of the *Getter* stereotype is enclosed in a dashed box in Fig. 13. As in the previous cases, input and output arcs are produced to link the transition starting the corresponding branch with the *Resource Registry* place, so as to obtain the property values and assign the client's variables with the values obtained (*PPL1* and *balance*).

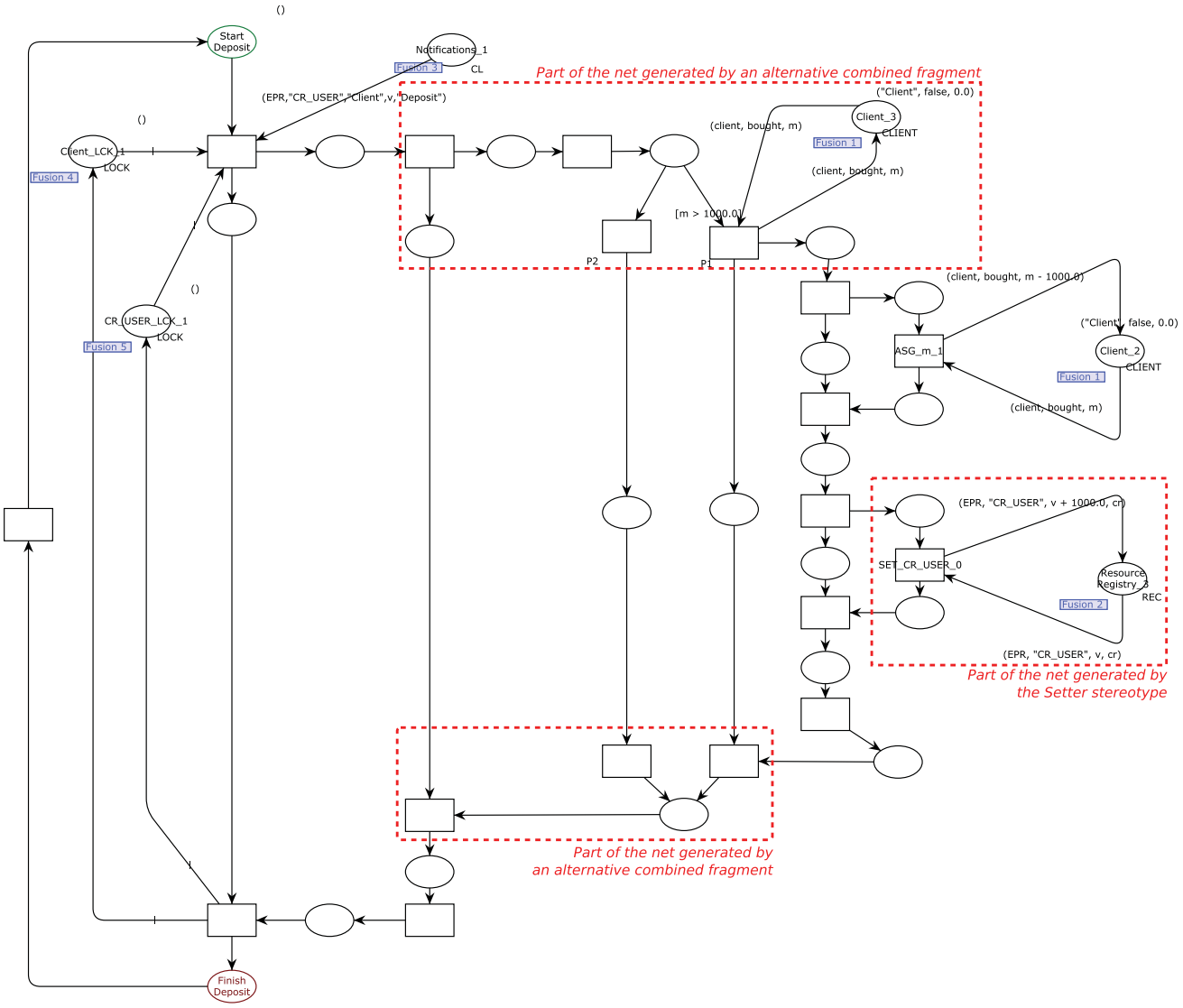

Fig. 11 Running example: TCPN model of UML-SD depicted in Fig. 10

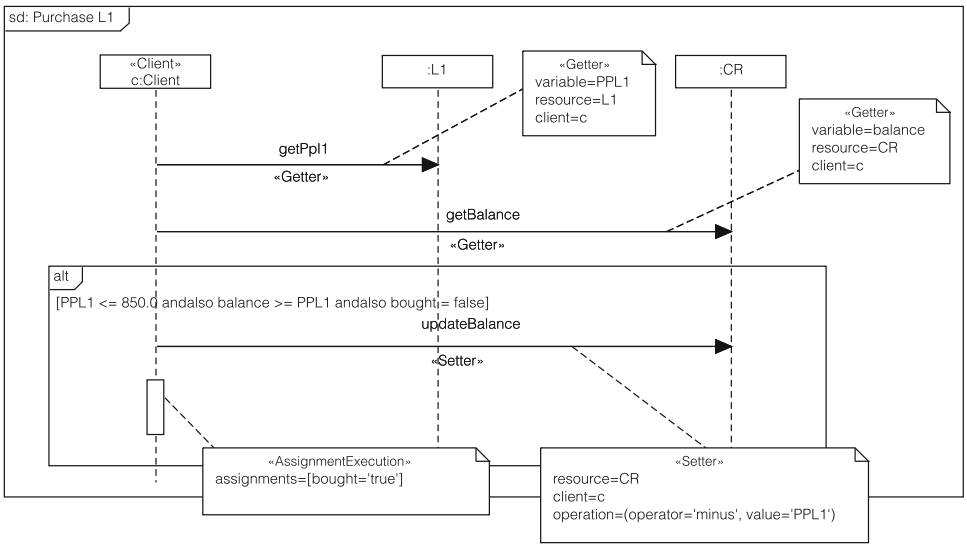

Fig. 12 Running example: UML-SD *Purchase L1*, annotated with PS profile

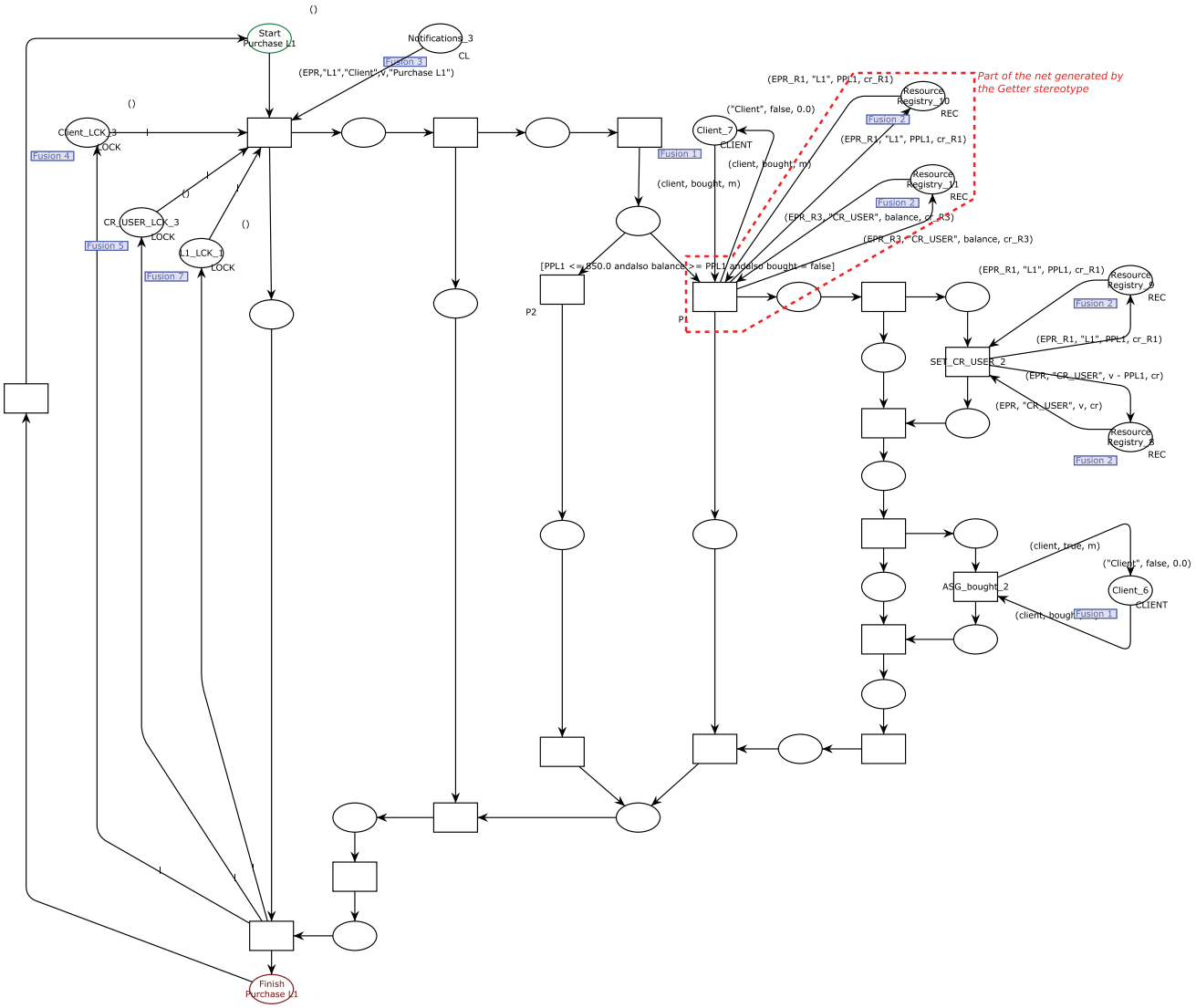

Fig. 13 Running example: TCPN model of UML-SD depicted in Fig. 12

5 A tool for modeling and simulating the publish/subscribe paradigm

To effectively automate the analysis of models expressing publish/subscribe interactions, we have implemented a complete toolset for their definition and automatic transformation. This toolset has been implemented using the Eclipse ecosystem due to its variety of tools for model-based development. In fact, Eclipse and its Eclipse Modeling Framework (EMF) [27] have become a *de facto* standard for building model-based tools, providing a common base for different purposes, e.g., model transformation [8,28], reverse engineering [29,30], code generation [31,32], or document generations [33], to name a few.

Eclipse is an open-source software development environment aimed at providing a platform for highly integrated tools. Indeed, it is usually described as "an open extensible IDE for anything and yet nothing in particular" [27] Specific tools can be plugged-in the base Eclipse platform to define a particular IDE configuration all together. Some pre-eminent projects within the Eclipse ecosystem that provide tools extending such core framework are the Eclipse Modeling Framework (EMF) [34], Eclipse Papyrus [35], or the QVT Operational SDK [36].

All these tools, among others that are presented next, are the baseline for our tool. In the following, we first introduce the description of the architecture of our tool and then illustrate how the UML models are transformed to CPN using QVT by means of a practical example. Finally, we show what our tool looks like from the final user's point of view.

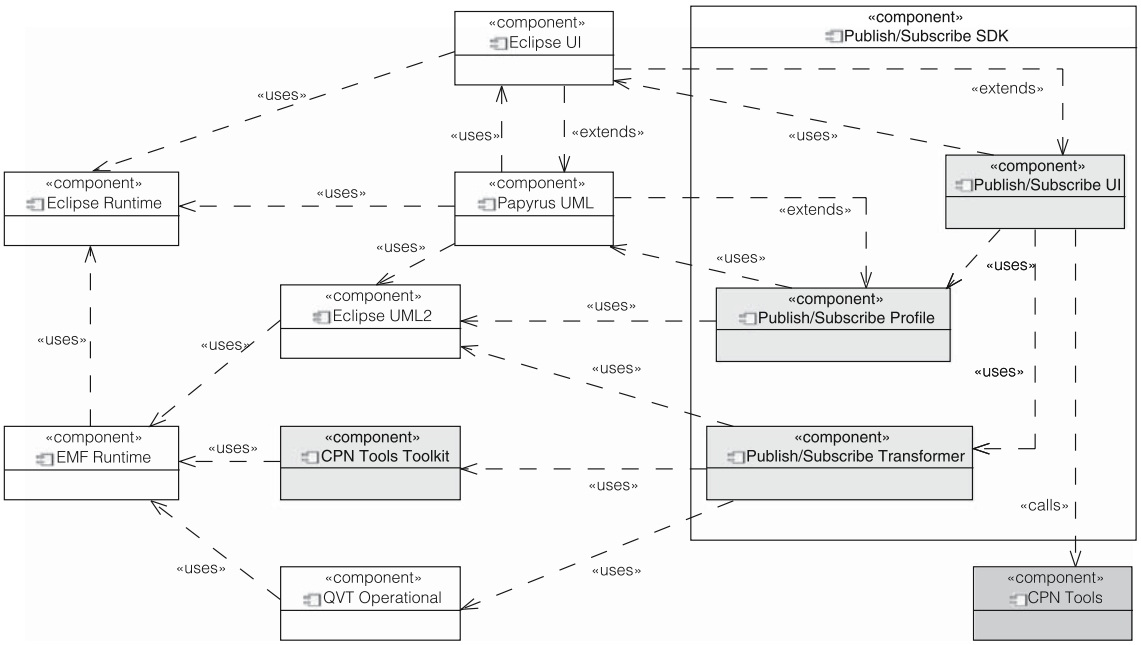

Fig. 14 Architecture of the publish/subscribe modeling and simulation tool

5.1 Architecture and components description

Figure 14 describes the architecture of our tool, showing the most remarkable components. The components with a white background represent coarse-grained Eclipse components, on which our tool relies, while the components with a light gray background represent the different component that were developed in the context of this work, and make our toolset for the modeling and transformation of the publish/-subscribe paradigm up. Finally, the dark gray component on the lower right-hand side of the figure represents the external CPN Tools tool. Next, a more detailed description follows:

Eclipse Runtime and *Eclipse UI* are the component-based runtime environment [37,38] and the user interface facilities provided by Eclipse, respectively.

EMF Runtime provides the modeling, meta-modeling, and code generation capabilities within the Eclipse platform. EMF uses *Ecore* [39] as the canonical language to describe models. An Ecore model is, essentially, a subset of the UML class diagram and thus can be considered as the reference implementation of the EMOF language proposed by the OMG [20].

Eclipse UML2 is an EMF-based implementation of the Unified Modeling Language 2.x OMG metamodel for the Eclipse platform. This component is considered to be a reference platform for UML that guarantees interoperability and provides the basis for the adoption and use of Model-Based Software Engineering (MBSE). This is the metamodel implementation used by our tool to support the definition of UML *Class Diagrams*, *Sequence Diagrams*, and *Deployment Diagrams* as presented in Sect. 3.

Papyrus UML is an industrial-grade open-source Model-Based Engineering (MBE) tool built on top of Eclipse. Papyrus has notably been used in industrial projects and is the base platform for several industrial modeling tools. Papyrus offers a very advanced support of UML profiles that enables users to define editors for DSLs based on the UML2 standard and its extension mechanisms. The main feature of Papyrus regarding this latter point is a set of very powerful customization mechanisms which can be leveraged to create user-defined Papyrus perspectives and give it the same look and feel as a native DSL editor. This component has been used to implement our publish/subscribe profile described in Sect. 4.1.

QVT Operational provides an implementation and an interpreter for the *Operational Mappings Language* defined in the query/view/transformation (QVT) [8] standard. The transformation patterns described in Sect. 4.3 are encoded in a *QVT Operational Mappings transformation* (as we briefly illustrate in the next subsection), which is executed by invoking this component.

CPN Tools Toolkit[6] provides the implementation of the metamodel presented in Sect. 4.2.[7] This metamodel is an extended and cleaned up version of a previous imple-

[6] https://github.com/abelgomez/cpntools.toolkit

[7] In fact, and following the model-driven development principles [19, 40], the metamodel shown in Fig. 5 is indeed the implementation itself.

mentation,[8] and is provided in a separate fashion from the *Publish/Subscribe SDK* tool for reusability purposes. This component also provides the ability to automatically post-process the generated Petri nets, applying a graph layout algorithm for a proper visualization.

Publish/Subscribe SDK is our Eclipse-based tool, which reuses, extends, and interacts with all the previous components. It is composed by:

- the *Publish/Subscribe Profile*, an implementation of the profile described in Sect. 4.1, which is based on *Eclipse UML2* and *Papyrus UML*;
- the *Publish/Subscribe Transformer*, an implementation of the transformation presented in Sect. 4.3 from the *Eclipse UML2* metamodel profiled with the *Publish/Subscribe Profile* to the *CPN Tools Toolkit* metamodel; and
- the *Publish/Subscribe UI*, the extensions plugged-in into the *Eclipse User Interface*, which allows invoking our transformation tool.

The *CPN Tools* component represents the external tool that is automatically invoked by the *Publish/Subscribe UI* once a UML model annotated with the *Publish/Subscribe Profile* is transformed into a Petri net by the *Publish/Subscribe Transformer* component.

5.2 From UML models to TCPN: a practical example using QVT

As previously introduced, the transformation patterns described in Sect. 4.3 have been encoded using the QVT *Operational Mappings Language* (QVTo).

A QVTo transformation represents the definition of a unidirectional transformation that is expressed imperatively. As Listing 2 shows, a *transformation* defines a signature indicating the models involved in the transformation and an entry operation for its execution (named *main*). The code excerpt shows the signature and the entry point of a transformation called `ps2cpntools`, which transform a *UML* model into a *CPN Tools* (TCPN) net. In the example, the *main* entry operation firstly calls the `interaction()` helper (which is not shown, but simplifying, retrieves the *Interaction* selected by the user) and then applies a mapping operation called `cpnet` on it. As shown in the listing, the `cpnet` mapping operation is the operation in charge of creating the corresponding instance of the *Cpnet* class of the metamodel shown in Fig. 5. Additionally, it initializes some of the *Cpnet* attributes by calling other mapping operations (line 14), or by directly instantiating new objects (lines 15–23). Objects that are created inline (such as the *Binder*) can also call subsequent mapping operation to initialize their attributes (lines 20–22).

Listing 2 Transformation declaration and main entry point, excerpt of the UML+Publish/Subscribe to CPN Tools QVT transformation (`ps2cpntools`)

```
1  transformation ps2cpntools(in uml : UML, out
       cpn : CPN);
2
3  main() {
4    var cpnet := uml.interaction().map cpnet()
       ;
5    -- Once the transformation has been
       executed, force the
6    -- automatic layout of the CPNet for all
       its pages
7    cpnet.binder.pages->forEach(p) {
8      var nNodes := p.places->size() + p.
         transs->size();
9      p.layout(nNodes * 40, nNodes * 40, 3000)
         ;
10   }
11 }
12
13 mapping UML::Interaction::cpnet() : CPN::
     Cpnet {
14   globbox := self.map invariantGlobbox();
15   binder := object CPN::Binder {
16     posx := 300;
17     posy := 30;
18     width := 500;
19     height := 500;
20     pages += self.map invariantPage();
21     pages += self.map scenarioPage();
22     pages += self.message[isSubscription()].
         subscription().map subscriptionPage
         ();
23   }
24 }
```

Listing 3 A transformation *mapping*, excerpt of the UML+Publish/Subscribe to CPN Tools QVT transformation (`ps2cpntools`)

```
1  mapping PS::Getter::getterSubnet
2      (inout _page : CPN::Page, in trans : CPN
         ::Trans) {
3    var pResourceRegistryGetter =
       pResourceRegistry(_page);
4    self.variable.map asRealVariable();
5    var inscription : String =
6      '(EPR_R{1}, "{2}", {3}, cr_R{4})'
7      ._format(
8        self.resource.epr, self.resource._tag,
           self.variable, self.resource.epr)
         ;
9    create_arc(_page, pResourceRegistryGetter,
       trans, inscription);
10   create_arc(_page, trans,
       pResourceRegistryGetter, inscription);
11 }
```

Listing 3 shows another example mapping that, thanks to its simplicity, serves as a clear demonstration of how the transformation patterns described in Sect. 4.3 can be described using QVTo. Specifically, it shows the basic pattern for the `Getter` stereotype. The *mapping* specifies that the `getterSubnet` mapping operation will be applied for a `Getter` stereotype application (line 1). Additionally, the *mapping* will receive a modifiable *page* and a read-only *tran-*

[8] https://issigit.dsic.upv.es/agomez/intergenomics.

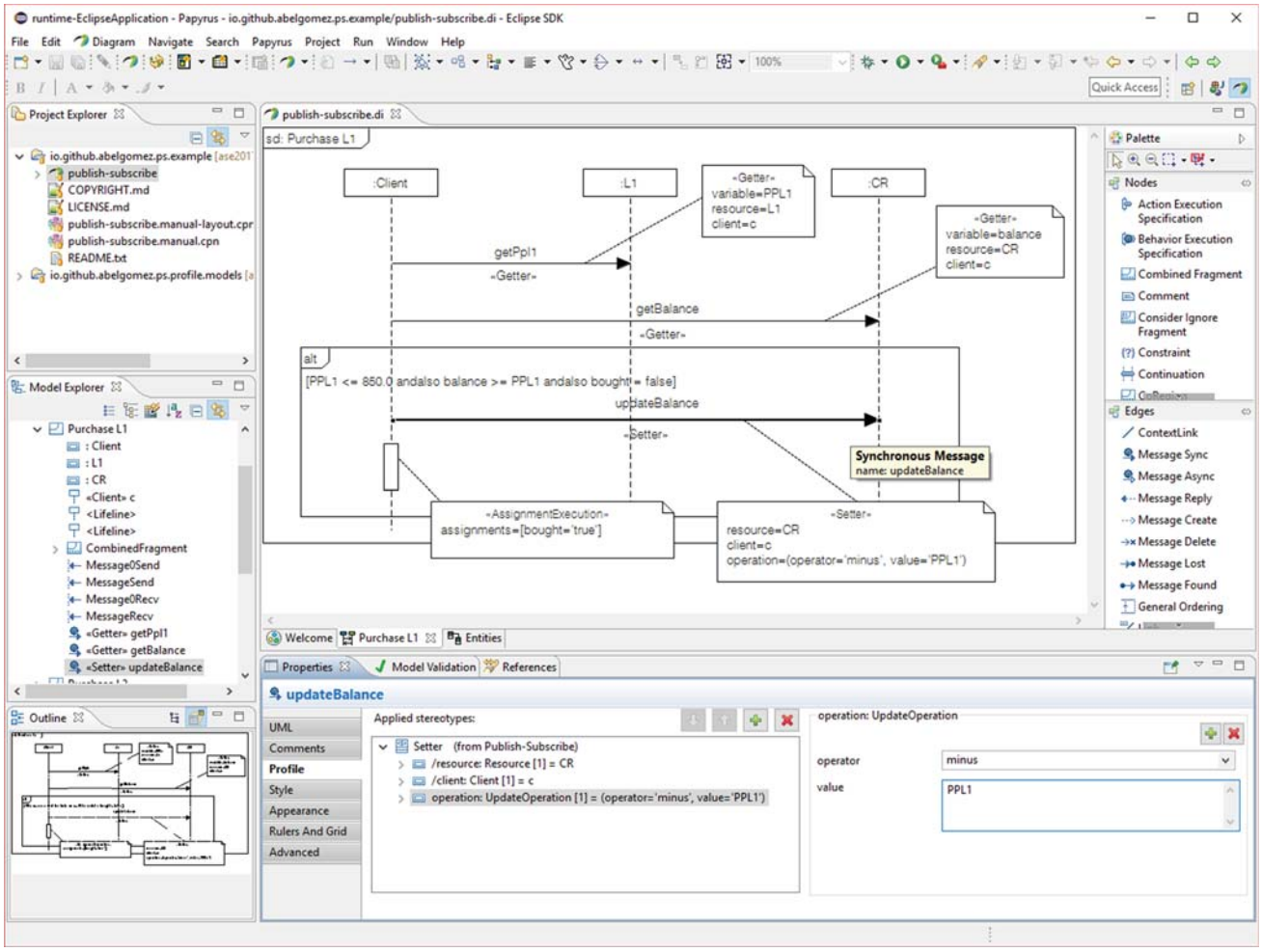

Fig. 15 Publish/Subscribe modeling and simulation tool

sition (line 2). The *page* is the element owning the *place* and the *arcs* created by this mapping, while the transition is the element to which the subnet created by this pattern will be attached. The `pResourceRegistry` operation will create and retrieve a *fusioned place* representing the *Resource Registry*. The instruction in line 4 creates a *Real* variable in the global *Globbox*, which will be later used in the *arc* expressions. Finally, lines 5–8 create the *arc* expression (the same for both arcs, making use of the *Resource* identifiers to avoid naming collisions), and lines 9–10 create the *arcs* between the place representing the *Resource Registry* and the *transition* passed as argument.

To inspect how other transformation patterns have been translated into QVTo, the full code of the `ps2cpntools` transformation can be checked online.[9]

5.3 Users' view

Although our tool for Modeling and Simulating the Publish/Subscribe Paradigm interacts with different tools and components, as well as relies on different models and file types, the orchestration among all these elements is performed in a transparent way from the final users' point of view. Thus, what a user sees is a light-weight integration of our tool within the Papyrus modeling environment.[10] Once the Publish/Subscribe profile is loaded into a UML model, users can apply the stereotypes defined in Sect. 4.1 to their models.

Figure 15 shows what the modeling environment looks like. In particular, this figure illustrates the *Purchase L1* UML-SD shown in Fig. 12a (all the UML diagrams in Sects. 4 and 5 have been directly included in this paper as they have

[9] The full code is available at https://github.com/abelgomez/publish-subscribe/tree/master/plugins/io.github.abelgomez.ps.transformer/transformation.

[10] Please refer to our online documentation for a detailed description of the tool user interface https://github.com/abelgomez/publish-subscribe/.

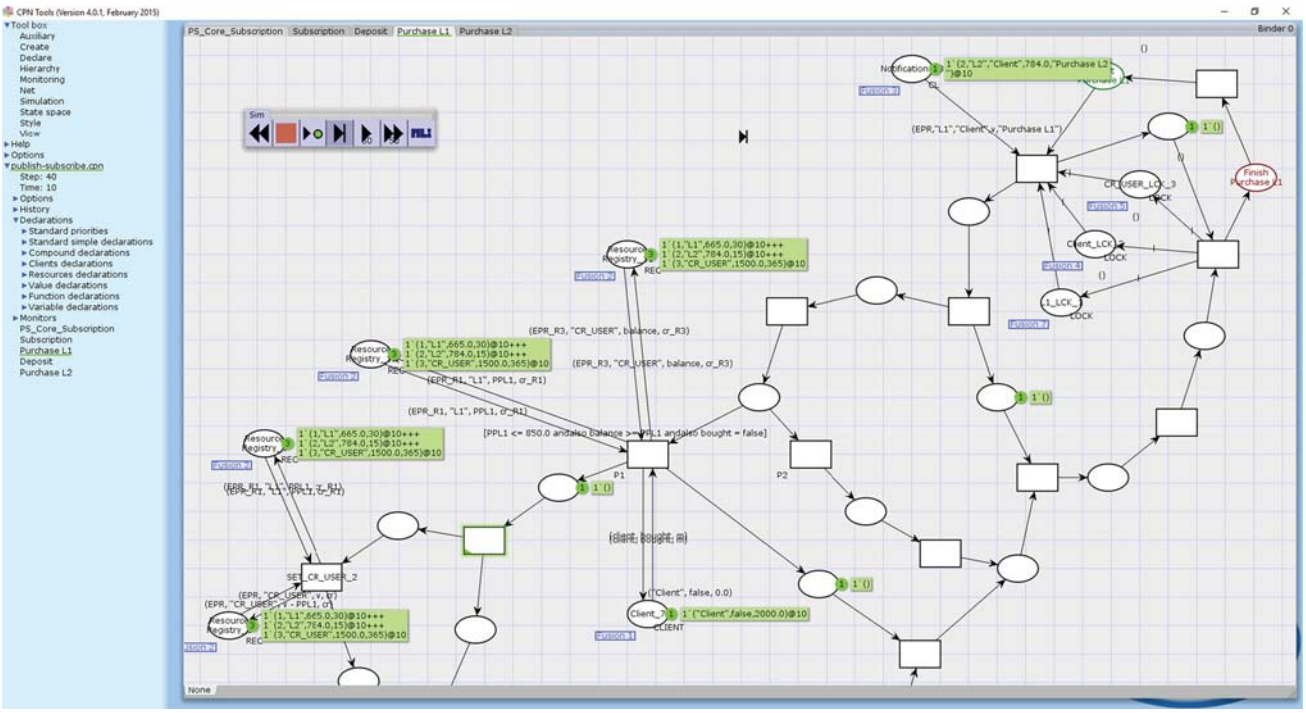

Fig. 16 Generated Petri net in CPN tools

been modeled with the only help of Papyrus and our profiling tool). The bottom part of the figure shows the *Properties* view, which allows setting the different tagged values. Specifically, the figure shows how the *operation* tagged value of the *updateBalance* message (stereotyped as *Setter*) is set to `(operator=minus, value='PPL1')`.

Finally, after a transformation process is launched from the Eclipse UI, a *CPN Tools* Petri net is generated and opened in the *CPN Tools* simulator, as show in Fig. 16. The figure shows in the active *page* the *Purchase L1* subnet (i.e., the same net that the one shown in Fig. 12b), with the automatically generated layout[11] during a simulation.

A limitation of our current approach is that the user must know how to use the CPN Tools for simulating the generated TCPN model. Unfortunately, CPN Tools does not provide a way to interact with it in a seamlessly mode (e.g., via command-line interface). Our next step is to solve this limitation by investigating how to interact with the graphical user interface of CPN Tools without any user intervention.

6 Verification and validation phases

In this section, we describe the verification and validation phases that are performed for the generated TCPN models, after the transformation of UML annotated models with the PS profile.

6.1 Verification phase

The verification phase takes place in two phases. Firstly, during the annotation of the UML model with the PS profile. This profile contains a set of OCL rules to verify the correctness of the annotated model with regard to the UML profile. For instance, the `ownerIsService` OCL rule shown at the top of Fig. 4 that verifies if the owner of a resource has been stereotyped with the `Service` stereotype. Secondly, during the transformation phase. The transformation patterns that have been defined for each stereotype are applied in a compositional way. Thus, as we have seen in the previous section, we define the transformation by pieces, where the behavior of each individual operator is first translated into a corresponding TCPN representation, whose behavior is the same as the operator it comes from. Thus, from this compositional approach, we guarantee that generated TCPN model reflect the behavior indicated in the UML annotated model. In addition, the QVT transformation also incorporates several OCL rules to verify that the UML annotated model is well-formed.

6.2 Validation phase

Once we have the generated CPN models, we can accomplish the validation phase. Table 1 contains a general set of prop-

[11] Please note, that Figs. 12b and 16 look different because the layout of the Petri net shown in Fig. 12b has been manually tweaked for readability purposes.

Table 1 Global properties of Publish/Subscribe paradigm to be fulfilled in the generated TCPN models

#	Property to check
1	If a resource is published, this resource becomes *published* with the information provided by the publisher
2	If a resource was already published, the publish operation fails
3	A published resource becomes expired when its lifetime runs out
4	All subscriptions to an expired resource are removed
5	A discovery operation must work for a published resource
6	A discovery operation fails when there is no published resource with the indicated tag
7	If *TimedSetter* is invoked with time argument 0 (new lifetime) and the resource is published, it becomes immediately expired or unpublished
8	Operations *TimedSetter*, *Setter*, *Getter* and *Subscription* fail when they are invoked for an expired or not published resource
9	If *TimedSetter* is invoked with a positive argument time, the resource becomes expired (unpublished) once the new indicated lifetime elapses
10	Operation *Getter* returns the actual property value of the resource, if it is currently published
11	Operation *Setter* changes the actual value of the resource, if it is published
12	A *Subscription* operation is immediately notified if the resource is published or a *Setter* operation or *TimedSetter* operation is performed such that the current value of the resource belongs to the indicated subscription interval
13	A subscription is removed when its associated lifetime runs out

erties that must hold for the TCPN model obtained by the translation. These properties have been checked by using the monitor features of the CPN Tools in the running example used in Sect. 4.3, and all of them are satisfied. A *monitor* is a mechanism to observe, inspect, control, or modify a simulation of a TCPN. Monitors allow to inspect the markings of places and the occurring binding elements during a simulation, and they can take appropriate actions based on the observations. Therefore, monitors are used for different purposes, such as stopping a simulation when a particular place is empty, counting the number of times a transition occurs, updating a file when a transition occurs with a variable bound to a specific value, or calculating the average number of tokens on a place.

Property 1 in Table 1 checks the basic functionality regarding the eventual publication of a resource. Property 2 allows us to check whether the publish operation fails when the resource is already published. Property 3 captures the resource expiration when its lifetime runs out. Property 4 checks whether all the subscriptions to an expired resource are removed. Properties 5 and 6 state the behavior of the discovery operation; specifically, property 5 checks if the discovery operation works with an already published resource, and property 6 if the discovery operation fails when the resource with the indicated tag does not exist. Properties 7, 8, 9, 10, and 11 capture the behavior of *TimedSetter, Getter* and *Setter* operations. Specifically, property 7 checks whether the resource becomes immediately unpublished when the *TimedSetter* operation is invoked with time argument 0 for a published resource. Property 8 captures whether *TimedSetter, Setter* and *Getter* fail when they are invoked for an expired resource. Property 9 tests the behavior of *TimedSetter* when it is invoked with a positive argument and the updated lifetime elapses. Properties 10 and 11 check whether *Getter* and *Setter* operations work properly when the resource is published. Properties 12 and 13 capture the basic functionality regarding the subscription and notification, respectively. Specifically, property 12 checks whether the *Subscription* operation works properly, that is, if the notification is sent when the resource value belongs to the subscription interval, once it has been published or a *Setter* operation has modified its value. Finally, property 13 checks whether the subscription is removed when its lifetime expires.

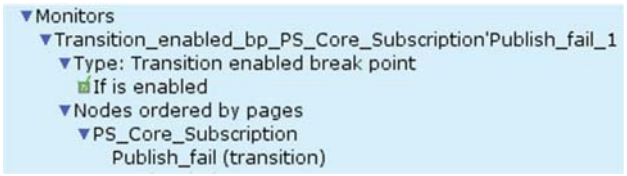

Fig. 17 Monitor feature of CPN tools defined for Property 2

As an example, Fig. 17 depicts the monitor used to check property 2 (we cannot publish twice a same resource). This monitor is defined as a *breakpoint monitor* (specifically, a *transition enabled monitor*), which checks whether a transition is enabled or not, stopping the simulation when enabled [14]. Thus, in this example, the monitor stops the simulation when the transition *Publish_fail* is enabled. To start the simulation, we consider the following initial marking in the *Resource Registry* place:

```
1`(1,"L1",950.0,365)@0+++
1`(2,"L2",980.0,365)@0+++
2`(3,"CR_USER",1500.0,365)@0
```

As shown, there are both laptop resources of types L1 and L2 and two identical user credit cards (CR_USER), with the same EPR (3). Thus, once the first instance of the credit card is published, the second instance cannot be published and transition *Publish_fail* becomes enabled, activating the monitor and stopping the simulation process. Figure 18 shows the state at which the simulation stopped, in which only one credit card has been published.

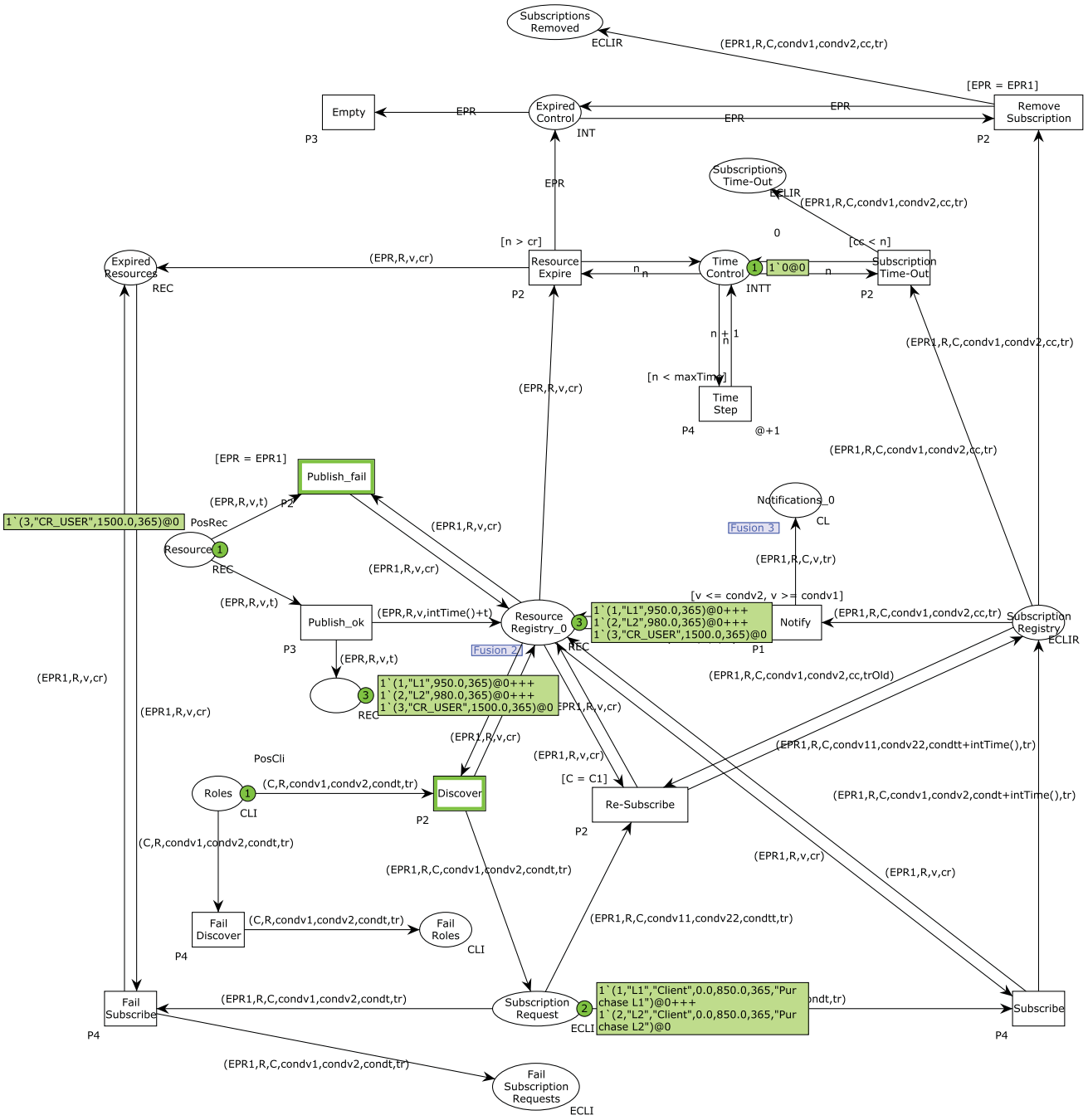

Fig. 18 TCPN final marking with monitor

Validation of the running example

Finally, the obtained TCPN model of the running example has been simulated, obtaining the following sequence of relevant events in the system:

1. Once the CPN Tools simulator tool opens, the initial TCPN marking for the *PS_Core_Subscription* page is that shown in Listing 1.
2. After 10 days, both laptop prices are reduced to 665€ and 784€, respectively.
3. Both subscription conditions are then fulfilled, so the *Notification* place becomes marked with the two corresponding tokens.
4. Only one purchase can proceed, because we only have one token in the *Lock* places. Thus, we have obtained the two possible purchases in different simulations. For

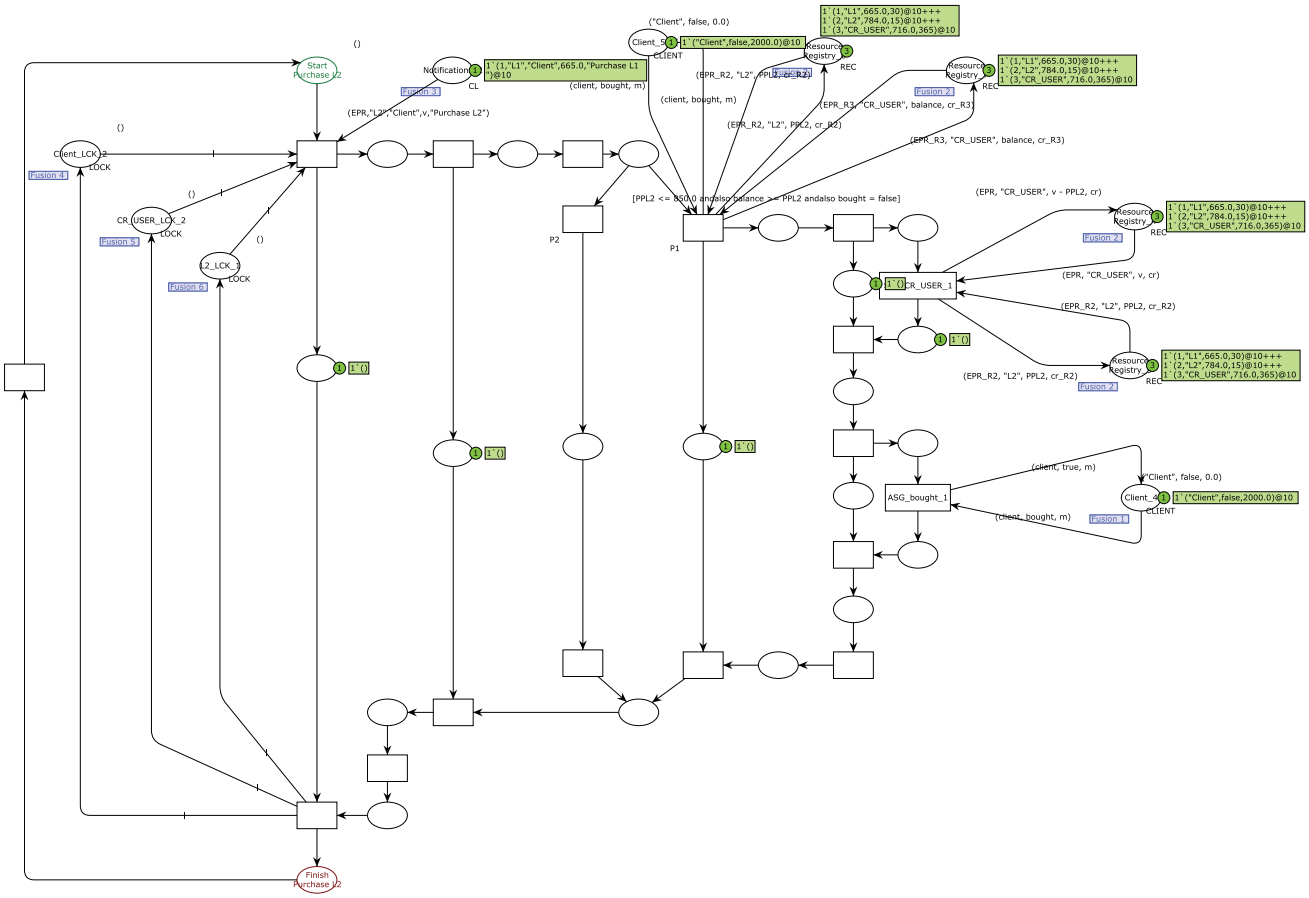

Fig. 19 Laptop 2 is bought and credit card balance low

instance, when laptop 2 is bought, the Credit Card balance is now 716€. This situation is shown in Fig. 19.

5. Since the credit card is lower than 1000€, the subscription condition for the *Deposit* UML-SD is fulfilled, so it is notified (the corresponding token is included in the *Notification* place. This situation is shown in Fig. 20.
6. The *Purchase* subnet is activated again with the other token (laptop 1), but as we now have that variable *bought* is true, its execution follows the default branch (no action) and terminates its execution.
7. Finally, the subnet corresponding to *Deposit* is therefore executed, and the client's credit card balance is increased by 1000€.

In this model, we have also checked other situations by changing the initial values of resources and client's information (initial credit card budget and cash available). For instance, we have checked the situation in which the initial credit card balance is 300€, with a cash of 300€. In this scenario, we have obtained that the *Deposit* subnet is immediately performed, so the credit card balance increases to 600€, and even when the laptop prices are reduced, she cannot afford to buy any of them, so the default branch is executed on both subnets with no action at all. Figure 21 shows that even though the laptop prices are reduced and the credit card balance is increased, she cannot buy any laptop. The enabled transition in the center of the figure corresponds to not buying the laptop, since the credit card balance is not enough. This transition has priority $P2$, so it fires because the transition on the right, which has a greater priority ($P1$) cannot be fired.

7 Related work

The Publish/Subscribe paradigm has received considerable attention in the last few years. A survey on this subject was carried out by Lin and Plade [2] and also by Eugster et al. [41], and formalizations of this paradigm can be found in Baldoni et al. [42] and Garlan et al. [43]. From these works, it becomes obvious that the way in which the Publish/Subscribe systems are modeled varies considerably depending on the specific model's goals. In our case, we have used a mechanism to publish distributed resources identified by a textual name,

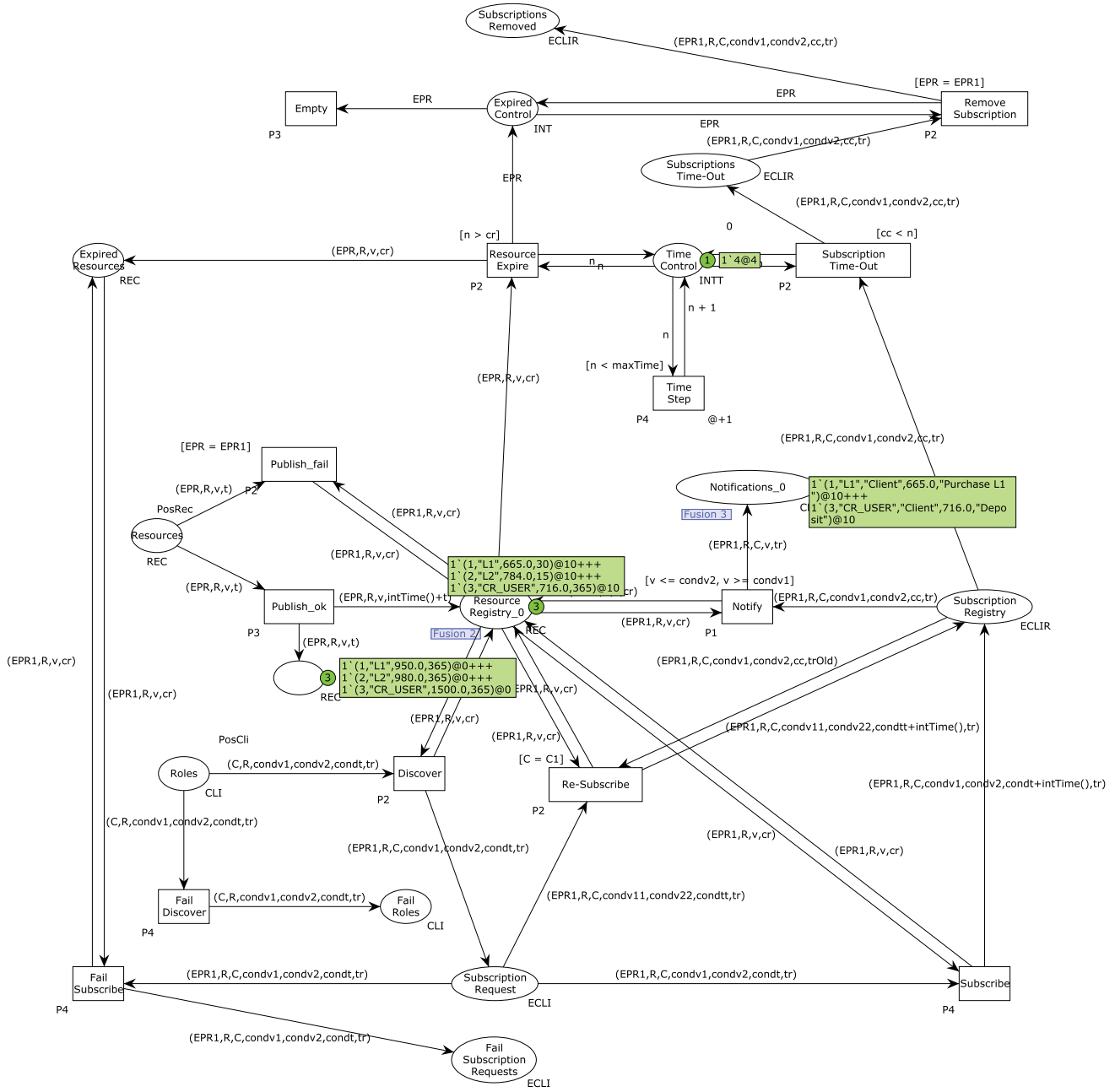

Fig. 20 Credit card low balance notified

and a mechanism to allow clients to discover these resources, by using these names.

To the best of our knowledge, there is no work combining the main features of UML 2.5 sequence diagrams with the WSRF standard for the description of distributed resources so as to automatically generate its corresponding representation in colored Petri nets to simulate and validate the system. In a preliminary work [12], we presented a UML formal framework based on a timed process algebra to model timed Web services with distributed resources and then, we provided a graphical model of timed Web services based on sequence diagrams that integrates the publish/subscribe paradigm in the context of distributed resources, with the goal that users have a formal framework to design these systems. In this work, we have used the formal framework presented in Valero and Cambronero, [12] as basis to define a UML profile and develop a tool that allows us to obtain automatically a corresponding TCPN.

Bran Selic [44] presented a generic framework for modeling resources with UML, focusing on the notion of *abstract resources*, which define the common characteristics of resources regardless of their specific manifestation. The

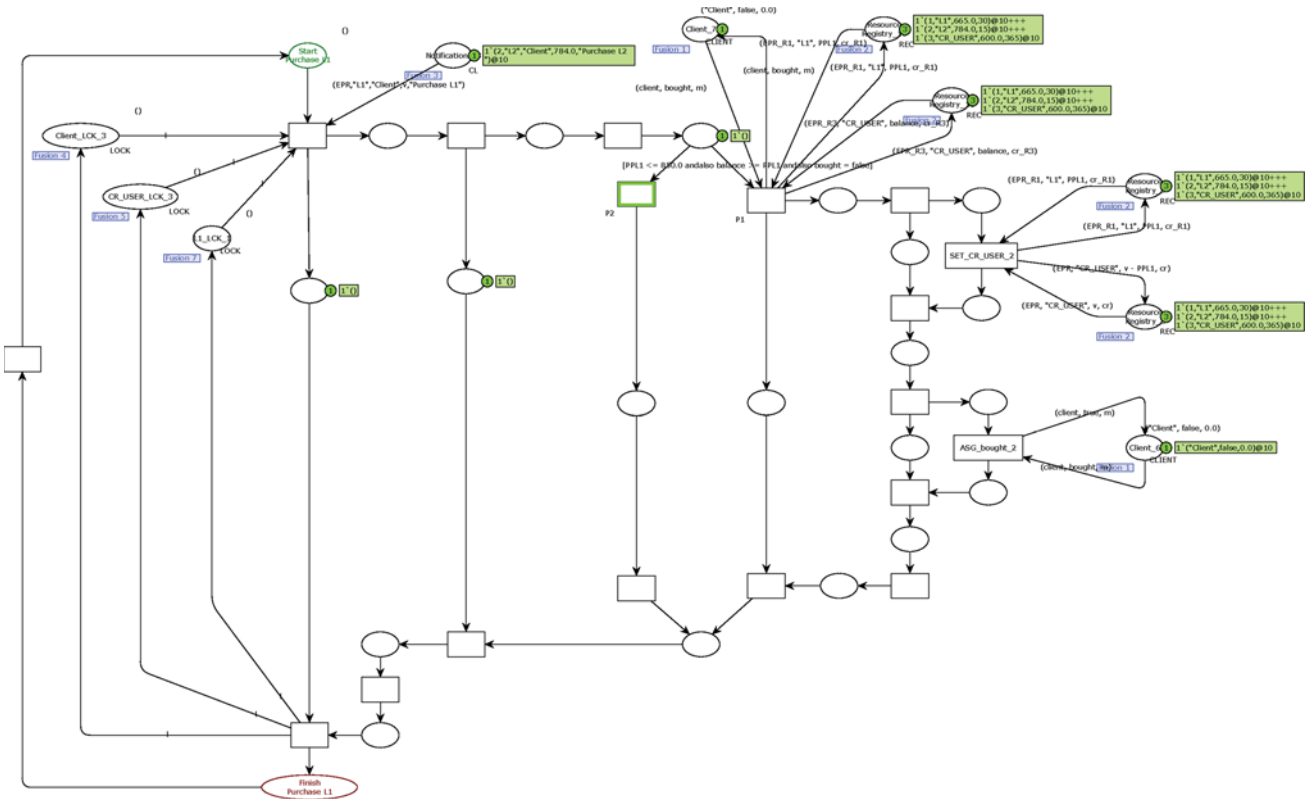

Fig. 21 Credit card balance low, no purchase

Publish/Subscribe paradigm was not considered in this paper and resources were modeled as servers with services, characterized by both their functional and non-functional aspects (such as response time and availability), thus focusing on Quality of Service (QoS) analysis.

There is considerable work on providing translations from UML sequence diagrams to colored Petri nets but, as previously mentioned, none of them integrate WSRF-resources in UML sequence diagrams. For instance, Mirandola and Cortellessa [45] used UML diagrams (in particular, use case diagrams, sequence diagrams, and deployment diagrams) to obtain a performance model of the system based on a queueing network, which can therefore be helpful as a support for early design decision making. Fernandes et al. [46] also presented a translation of use cases and UML 2.0 sequence diagrams to colored Petri nets supported by CPN Tools, but the transformation is only performed on a specific case study, an elevator controller. Bowles and Meedeniya [47] defined a formal strongly consistent transformation from UML sequence diagrams to colored Petri nets by using a set of transformation rules, showing that the obtained CPNs are equivalent in terms of trace semantics.

Translations of UML sequence diagrams to other formalisms have also been done. A true-concurrent semantics for UML sequence diagrams was defined by Juliana Bowles [48,49], using a two-level logic interpreted over labeled event structures. Starting from these two works, Bowles, Bordbar and Alwanain [50] checked the consistency of a composition of UML sequence diagrams, using a set of logical constraints to describe their behavior. Bernardi et al. [17] proposed patterns for model transformation from UML sequence diagrams and state-charts into a particular subclass of timed Petri nets. However, only plain UML models (not annotated) were considered. A transformation of UML sequence diagrams into State Machines by using graph transformation techniques was defined by Grønmo and Møller-Pedersen [51], by taking the parallel, choice, loop, and neg Combined Fragments, although time and resources were not considered in that approach. In Cambronero et al. [52], we used the RT-UML profile [20] for UML 2.0 and defined a corresponding process algebra, capturing the main aspects related to sequence diagrams extended with combined fragments to obtain a translation into a network of timed automata. Hence, the modeled system is suitable for simulation and analysis by means of the existing tools supporting timed automata, such as UPPAAL [53]. In that work resources were not considered, so the current work extends such a previous work by introducing a WSRF-compliant UML profile of distributed resource management, including the Publish/-Subscribe paradigm. Tribastone and Gilmore [54] defined

a translation of UML sequence diagrams annotated with stochastic information into the stochastic process algebra PEPA [55] to carry out quantitative evaluation. In Distefano et al. [18], proposed a methodology to validate the performance of a UML model representing a software architecture. They annotated the UML models and derived a Stochastic Petri net where performance measures are assessable.

Testing and analysis of model transformation has also been studied by Hilken et al. [56]. In that paper, the authors proposed the use of partitioning techniques based on classifying terms for testing models and model transformation. Anatasakis et al. [57] describe how to transform UML models into Alloy in order to apply the Alloy analyzer [58] and thus check and identify design faults within a specification. In a similar way, Gogolla et al. [59] study the testing and certification of UML and OCL models by using the validation tool USE [60].

8 Conclusions and future work

Many Web services behave as stateful distributed services, allowing a user to access and manipulate states when the user interacts with the service. In this paper, we focus on timed Web services that manage a collection of distributed resources using the Publish/Subscribe paradigm and the OASIS WSRF standard. These systems present the following characteristics: (i) the resources are published by some publisher and subscribers can submit their subscription conditions to be notified when these conditions become true; and (ii) resources have standardized operations for the management of resources.

In this paper, we have proposed a UML profile for the Publish/Subscribe paradigm that enables us to represent the underlying concepts of this paradigm in the UML models by means of annotations. We have also introduced a set of rules for transforming a UML annotated system into a formal model; particularly, to Colored Petri nets. Furthermore, we have developed a model-to-model transformation tool that follows these rules and provides us with a Colored Petri net model compatible with the format used by CPN Tools, a well-known tool for editing, simulating, and analyzing these nets. The obtained model becomes useful to detect and fix design errors in early stages of system development, thus saving production costs. To foster research in this area and for the sake of the reproducibility of our research in this paper, we publish our tool and the UML models of the case study that we used for the evaluation under an open-source license, namely, the Eclipse Public License [27]. All information is available at https://github.com/abelgomez/publish-subscribe/.

As future work, we aim at further extending the validation of properties of the Publish/Subscribe paradigm to include also properties more specific to the particular problem under consideration. Furthermore, we aim at improving the feedback to the user regarding the design errors detected and at providing a more seamlessly integration with CPN Tools to facilitate the adoption of our approach for non-expert users in Petri nets. Similarly, the automatic generation of code is also an important phase of our methodology that needs further research.

Acknowledgements The research of A. Gómez and R. J. Rodríguez was supported in part by the EU H2020 through the DICE Project under Grant 644869 and in part by the Spanish MINECO through CyCriSec Project under Grant TIN2014-58457-R. The research of M. E. Cambronero and V. Valero was supported by the Spanish Ministry of Science and Innovation and the European Union FEDER Funds through the DArDOS Project under Grant TIN2015-65845-C3, subproject 2-R, and also by the JCCM regional project SBPLY/17/180501/000276, which is also co-financed by the European Union FEDER Funds.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts. Architectures and Applications. Springer, Berlin (2004)
2. Lin, Y., Plade, B.: Survey of Publish-Subscribe Event Systems. Technical Report 16, Computer Science Department, Indiana University (2003)
3. Niblett, P., Graham, S.: Events and service-oriented architecture: the OASIS web services notification specifications. IBM Syst. J. **44**(4), 869–886 (2005)
4. OMG (2015) Unified Modeling Language (UML), v2.5. http://www.omg.org/spec/UML/2.5
5. Jensen, K.: Coloured Petri Nets. Monographs in Theoretical Computer Science. Analysis Methods and Practical Use. Springer, Berlin, Basic Concepts (1997)
6. Selic, B.: A systematic approach to domain-specific language design using UML. In: 10th IEEE International, Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), IEEE Computer Society, Santorini Island, Greece, pp. 2–9 (2007)
7. Lagarde, F., Espinoza, H., Terrier, F., Gérard, S: Improving UML profile design practices by leveraging conceptual domain models. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, ACM, New York, ASE'07, pp. 445–448 (2007)
8. OMG (2016) Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.3. http://www.omg.org/spec/QVT/1.3/
9. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf. **9**(3), 213–254 (2007)
10. Randimbivololona, F.: Orientations in Verification Engineering of Avionics Software. In: Wilhelm, R. (ed.) Informatics, Lecture Notes in Computer Science, vol. 2000, pp. 131–137. Springer, Berlin (2001)
11. Cambronero, M.E., Valero, V.: Modelling distributed service systems with resources using UML. In: Proceedings International

Conference on Computational Science (ICCS'13), Procedia Computer Science, pp. 140–148 (2013)
12. Valero, V., Cambronero, M.E.: Using unified modelling language to model the Publish/Subscribe paradigm in the context of timed web services with distributed resources. Math. Comput. Model. Dyn. Syst. **23**(6), 570–594 (2017)
13. Jensen, K., Kristensen, L.: Coloured Petri Nets. Modelling and Validation of Concurrent Systems. Springer, Berlin (2009)
14. CPN Tools.: CPN Tools Homepage. http://www.cpntools.org/ (2017). Accessed 12 Oct 2018
15. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989)
16. Baeten, J., Middelburg, C.: Process Algebra with Timing. EATCS Monographs Series. Springer, Berlin (2002)
17. Bernardi, S., Donatelli, S., Merseguer, J.: From UML sequence diagrams and statecharts to analysable Petri Net models. In: Proceedings of the Third International Workshop on Software and Performance (WOSP2002), ACM, Rome, Italy, pp. 35–45 (2002)
18. Distefano, S., Scarpa, M., Puliafito, A.: From UML to Petri nets: the PCM-based methodology. IEEE Trans. Softw. Eng. **37**(1), 65–79 (2011)
19. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co., New York (2000)
20. OMG.: Object Management Group. http://www.omg.org/ (1989) . Accessed 12 Oct 2018
21. Bernardi, S., Merseguer, J., Petriu, D.: A dependability profile within MARTE. J. Softw. Syst. Model. **10**(3), 313–336 (2011)
22. Rodríguez, R.J., Merseguer, J., Bernardi, S.: Modelling security of critical infrastructures: a survivability assessment. Comput. J. **58**(10), 2313–2327 (2015)
23. OMG.: Object Constraint Language (OCL), Version 2.3.1. http://www.omg.org/spec/OCL/2.3.1/ (2012). Accessed 12 Oct 2018
24. Gómez, A.: Intergenomics—Transpat2CPN. https://issigit.dsic.upv.es/agomez/intergenomics (2006). Accessed 12 Oct 2018
25. Milner, R., Tofte, M., Macqueen, D.: The Definition of Standard ML. MIT Press, Cambridge (1997)
26. Valero, V., Macià, H., Díaz, G., Cambronero, M.E.: Colored Petri net modeling of web services resources. In: 20th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'15), Lecture Notes in Computer Science, vol. 9128, pp. 81–95 (2015)
27. Eclipse Foundation.: Eclipse: The Platform for Open Innovation and Collaboration. https://www.eclipse.org/ (2004). Accessed 12 Oct 2018
28. INRIA, LINA.: ATLAS transformation language. http://www.eclipse.org/atl/ (2014). Accessed 9 Apr 2018
29. Bruneliére, H., Cabot, J., Dupé, G., Madiot, F.: MoDisco: a model driven reverse engineering framework. Inf. Softw. Technol. **56**(8), 1012–1032 (2014)
30. The Eclipse Foundation.: MoDisco Eclipse Project. http://www.eclipse.org/MoDisco/, http://www.eclipse.org/MoDisco/ (2014). Accessed 9 Apr 2018
31. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend, 2nd edn. Packt Publishing, Birmingham (2016)
32. Eclipse Foundation: ATL - a model transformation technology. http://www.eclipse.org/atl/. Accessed 3 Jan 2019
33. Gómez, A., Penadés, M.C., Canós, J.H., Borges, M.R., Llavador, M.: A framework for variable content document generation with multiple actors. Inf. Softw. Technol. **56**(9), 1101–1121 (2014)
34. Eclipse Foundation.: Eclipse Modeling Project. http://www.eclipse.org/emf/ (2017). Accessed 12 Oct 2018
35. Eclipse Foundation.: Papyrus. https://eclipse.org/papyrus/ (2017). Accessed 12 Oct 2018
36. Eclipse Foundation.: Eclipse QVT Operational. https://projects.eclipse.org/projects/modeling.mmt.qvt-oml (2017). Accessed 12 Oct 2018
37. McAffer, J., VanderLei, P., Archer, S.: OSGi and Equinox: Creating Highly Modular Java Systems. Eclipse Series. Addison-Wesley, Boston (2009)
38. OSGi Alliance.: OSGi Service Platform Core Specification. Tech. rep., OSGi Alliance. http://www.osgi.org/Specifications/ (2008). Accessed 9 Apr 2018
39. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional, Boston (2009)
40. Selic, B.: The pragmatics of model-driven development. IEEE Soft. **20**(5), 19–25 (2003)
41. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of Publish/Subscribe. ACM Comput. Surv. **35**(2), 114–131 (2003)
42. Baldoni, R., Contenti, M., Tucci, S., Virgilio, A.: Modelling Publish/Subscribe communication systems: towards a formal approach. In: Proceedings 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, pp. 304–311 (2003)
43. Garlan, D., Khersonsky, S., Kim, J.: Model-checking publish-subscribe systems. In: Proceedings 10th International SPIN Workshop on Model Checking Software (SPIN'03), pp. 166–180 (2003)
44. Selic, B.: A generic framework for modeling resources with UML. Comput. J. **6**, 64–69 (2000)
45. Mirandola, R., Cortellessa, V.: UML based performance modelling of distributed systems. In: Proceedings 3rd International Conference on the Unified Modelling Language: Advancing on the Standard, UML'00, Lecture Notes in Computer Science, vol. 1939, pp. 178–193 (2000)
46. Fernandes, J.M., Tjell, S., Jorgensen, J.B., Ribeiro, O.: Designing tool support for translating use cases and UML 2.0 sequence diagrams into a Coloured Petri net. In: Sixth International Workshop on Scenarios and State Machines, 2007. SCESM '07: ICSE Workshops 2007, pp. 2–2 (2007)
47. Bowles, J., Meedeniya, D.: Formal transformation from sequence diagrams to Coloured Petri nets. In: 2010 Asia Pacific Software Engineering Conference, pp. 216–225 (2010)
48. Juliana, B.: Modelling Concurrent Interactions. Theor. Comput. Sci. **351**(2), 203–220 (2006b)
49. Juliana, B.: Decomposing interactions. In: Michael, J., Varmo, V. (eds.) Algebraic Methodology and Software Technology, pp. 189–203. Springer, Heidelberg (2006a)
50. Juliana, B., Behzad, B., Mohammed, A.: A logical approach for behavioural composition of scenario-based models. In: 17th International Conference on Formal Engineering Methods (ICFEM 2015), LNCS vol. 9407, pp. 252–269 (2015)
51. Grønmo, R., Møller-Pedersen, B.: From UML 2 sequence diagrams to state machines by graph transformation. J. Object Technol. **10**(8), 1–2 (2011)
52. Cambronero, M.E., Valero, V., Díaz, G.: Verification of real-time systems design. Softw. Test. Verif. Reliab. **20**(1), 3–37 (2010)
53. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a Nutshell. Int. J. Softw. Tools Technol. Transf. **1**(1–2), 134–152 (1997)
54. Tribastone, M., Gilmore, S.: Automatic translation of UML sequence diagrams into PEPA models. In: International Conference on Quantitative Evaluation of Systems (QEST'08), pp. 205–214 (2008)
55. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press, Cambridge (1996)
56. Hilken, F., Gogolla, M., Burgueño, L., Vallecillo, A.: Testing models and model transformations using classifying rerms. Softw. Syst. Model. **17**(3), 885–912 (2018)

57. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to alloy. Softw. Syst. Model. **9**, 69–86 (2010)
58. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, London (2006)
59. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL models in USE by automatic snapshot generation. Softw. Syst. Model. **4**(4), 386–398 (2005)
60. Richters, M., Gogolla, M.: Validating UML models and OCL constraints. In: Evans, A., Kent, S., Selic, B. (eds.) International Conference on the Unified Modeling Language 2000, pp. 265–277. Springer, Heidelberg (2000)

Abel Gómez is an Assistant Professor at the *Faculty of Computer Science, Multimedia and Telecommunications*, and a researcher of the *Internet Interdisciplinary Institute*, both belonging to the *Universitat Oberta de Catalunya*, Spain. Previously, he has hold different positions at the *Universidad de Zaragoza*, the *École des Mines de Nantes & Inria*, and the *Universitat Politècnica de València*, being this latter institution where he obtained his PhD degree in Computer Science. His research interests fall in the broad field of model-driven engineering (MDE), and his research lines have evolved in two complementary directions: On the one hand, the development of core technologies to support MDE activities; and on the other hand, the application of MDE techniques to solve Software Engineering problems. More information is available at https://abel.gomez.llana.me.

Ricardo J. Rodríguez is an Assistant Professor at *Centro Universitario de la Defensa, General Military Academy*, Zaragoza, Spain. He received the M.S. and Ph.D. degrees in computer science from the *University of Zaragoza*, Spain, in 2010 and 2013, respectively. He was a Visiting Professor at the *Dept. of Mathematics and Physics, University of Campania "Luigi Vanvitelli"*, Caserta, Italy, during a three-month period in 2016 and other 3-month period in 2018. He was also a Visiting Professor at the Technische Universität Ilmenau, Germany, during a three-month period in 2017. His current research interests include performability and dependability model-based analysis, program binary analysis, and contactless cards security.

Maria Emilia Cambronero is an Associate Professor in Computer Science at *University of Castilla-La Mancha*, Spain, obtaining the tenure distinction in 2012. She received her PhD in 2007 and was an Assistant Professor for several years in the same university. Her research goals are aimed to make software more reliable, more secure, and easier to design. Her primary technical interests include software engineering and related areas, including contract specification, program monitoring, testing, and verification. Her research combines strong theoretical foundations with realistic experimentation in the area of web services and cloud computing.

Valentin Valero is a full Professor of Distributed Systems and Operating Systems at the *University of Castilla-La Mancha*, in the Computer Science School of Albacete, Spain. He received his degree in Mathematics from the *Complutense University of Madrid* in 1987, and his PhD. in Mathematics in 1993 at the *Department of Computer Science* of the *Complutense University of Madrid*. Since October 1987 he is a member of the *Computer Science Department* at the *University of Castilla-La Mancha*. His current research areas are in the field of concurrency, specifically in formal models for analysis and design of concurrent systems and real-time systems.