Strategies for Modeling Complex Processes using Colored Petri Nets

Wil M. P. van der Aalst, Christian Stahl, and Michael Westergaard

Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, The Netherlands.
{W.M.P.v.d.Aalst,C.Stahl,M.Westergaard}@tue.nl

Abstract. Colored Petri Nets (CPNs) extend the classical Petri net formalism with data, time, and hierarchy. These extensions make it possible to model complex processes as CPNs without being forced to abstract from relevant aspects. Moreover, CPNs are supported by CPN Tools—a powerful toolset that supports the design and analysis of such processes. The expressiveness of the CPN language enables different modeling approaches. Typically, the same process can be modeled in numerous ways. As a result, inexperienced modelers may create CPNs that are unnecessarily convoluted and bulky. Using a running example and a set of design patterns, we show how to solve typical design problems in terms of CPNs. By following these guidelines, it is possible to create succinct, but also comprehensible, models. In addition, we present some new features supported by CPN Tools 3.0 (e.g., priorities and real time stamps) and show how the software can be used for performance analysis (i.e., comparing design alternatives using simulation).

Keywords: Colored Petri nets, Design patterns, CPN Tools

1 Introduction

Petri nets have been around for about half a century and have shown to be able to model concurrent processes adequately. The basic formalism is simple and enables powerful analysis techniques. However, it is not easy to model complex processes in terms of classical Petri nets. Therefore, many extensions of the basic formalism have been proposed in the literature [1,13,14,15,17,18,19,20,22,23,30]. In fact, hundreds of extensions have been proposed for classical Petri nets, and it is impossible to name them all here. Some of the extensions proposed are rather exotic and did not progress beyond a proposal on paper (i.e., no tool support and no practical applications), whereas other extensions are widely supported and frequently used. Despite the many proposals, there seems to be consensus on the need for three types of *extensions*:

– *The extension with data.* In the classical Petri net, two tokens *cannot* be distinguished. The only way to distinguish two tokens is to put them in separate places. This is not practical for realistic applications as the model

quickly becomes extremely complex and potentially infinitely large. In fact, for most practical applications of Petri nets, we would like the tokens to be distinguishable and have particular characteristics (e.g., age, weight, price, value, owner, or address). For a token modeling a car, we may want to describe its brand, model, color, or license number. Therefore, we need to add data to the basic model. Without this extension, Petri nets are like a programming language without variables and parameters.

- *The extension with hierarchy.* No matter how expressive a modeling language is, models tend to become large because in most applications there are many entities that interact in a nontrivial manner. Therefore, a hierarchy concept is needed to deal with this type of complexity. When designing a model one would like to use a divide-and-conquer approach. Moreover, structuring a model is essential when communicating design choices and analysis results with stakeholders. Without hierarchy, Petri nets are like a programming language lacking subroutines and subprocedures.
- *The extension with time.* Petri nets are often used to model processes where time plays an important role. For example, activities take time or exception handling is required after a timeout. These temporal aspects should be reflected in the model. Durations may be deterministic or stochastic. In the latter case, the model typically also incorporates routing probabilities such that performance analysis comes into reach. In many application domains it is important to use models to predict response times, utilization, flow times, and service levels.

Although there is consensus on the need to support data, hierarchy, and time for practical applications of Petri nets, different proposals have been made. Some of the differences between competing proposals are mainly syntactical; for example, CPN Tools is using ML as an inscription language [19,24] whereas ExSpect is using a dedicated functional language [3,15]. Other differences are more relevant; for example, the hierarchy concept used in CPN Tools (transition refinement) is very different from the nets-in-nets paradigm used by Renew [21,30].

Colored Petri Nets (CPNs) are the most widely used formalism incorporating data, hierarchy, and time [6,17,18,20,19]. Initially, CPNs were supported by *Design/CPN*. Later, Design/CPN was replaced by *CPN Tools*.[1] Currently, CPN Tools is by far the most widely used Petri net tool. CPN Tools supports the design of complex processes and the analysis of such processes using state-space analysis and simulation.

Modeling complex processes in terms of CPNs is a nontrivial task. The expressiveness of the language allows for different styles of modeling; that is, the same process can be modeled in different ways. Modeling is "an art rather than a science", but there are recurring modeling problems that can be solved by applying *design patterns*.

The most well-known patterns collection in the IT domain is the set of design patterns documented by Gamma, Helm, Johnson, and Vlissides [12]. This

[1] See `http://cpntools.org`.

collection describes a set of problems and solutions frequently encountered in object-oriented software design. The success of the patterns described in [12] triggered many patterns initiatives in the IT field, including the Workflow Patterns Initiative [4,32]. The idea to use a patterns-based approach originates from the work of the architect Christopher Alexander. In [7], he provided rules and diagrams describing methods for constructing buildings. The goal of the patterns documented by Alexander is to provide generic solutions for recurrent problems in architectural design. The idea to use patterns for design problems in the IT domain is appealing as is reflected by the different collections of patterns [4,5,10,12,16,29,31,32]. Many of these collections focus on behavioral aspects as these are most difficult to model and implement.

The idea to provide patterns for modeling in terms of CPNs was first proposed in [26]. Based on expert opinions and an analysis of large collections of CPNs (taken from papers and Web pages), 34 patterns were identified (see Appendix). These patterns help to tackle particular problems. Each pattern is described using a standard format including elements such as pattern name, intent, motivation, problem description, solution, implementation considerations, examples, and related patterns. In this paper, we explain the most important patterns using a running example. Unlike in [25,26], we do not explicitly enumerate the patterns nor will we use a strict format to present them. Instead, we use a tutorial-style presentation showing how to address frequently recurring modeling problems.

For a detailed description of the CPN language we refer to [18,19]. Our goal is not to describe the language but to focus on the way it can be used most efficiently. This paper is based on lectures given by the authors during the 5th Summer School/Advanced Course on Petri nets (Rostock, September 2010). Hence, the goal is not to present new scientific results, but to guide people using the CPN language and CPN Tools. In addition, the paper presents recent extensions of CPN Tools. As of CPN Tools 3.0, priorities and real time stamps are supported. We shall show that these extensions provide additional support when tackling some of the most important design patterns.

In the last part of the paper, we focus on one particular analysis technique: *simulation*. We shall show that the timing concept used by CPNs is compelling and gives the designer full control over temporal aspects of the model. Moreover, CPN Tools provides a powerful simulation environment. Using our running example, we shall show that it is easy to compare different alternative models.

The remainder of this paper is organized as follows. Section 2 introduces the basics of CPNs. The focus is on the extension with data, that is, colored tokens. As a running example, we use a gas station that serves two types of customers. The extension with hierarchy is described in Sect. 3. Subsequently, we use different variants of the gas station to explain four of the simple design patterns (Sect. 4) and two of the more advanced design patterns (Sect. 5). The hierarchy concept is used to structure these patterns while the patterns themselves focus on the interplay between control-flow and data. Here, we also show how the priority concept of CPN Tools 3.0 can be used to simplify the realization of some of

the patterns. In Sect. 6, we shift our attention to modeling of time. We explain the time concept and highlight the new timing functionality of CPN Tools 3.0 (i.e., real time). Subsequently, Sect. 7 shows how the addition of stochastic elements can be used to simulate complex processes and analyze their performance. Section 8 concludes the paper.

2 Colored Petri Nets: Basics

Colored Petri nets (CPNs) extend classical Petri nets with data, hierarchy, and time. In this section we focus on the *extension with data*.

In a classical Petri net, tokens are indistinguishable (black), whereas in CPNs tokens are distinguishable; tokens may have different colors such that they can be differentiated. A colored token carries data attributes that characterize the entity it represents. Note that we use the terms "color" and "data" interchangeably. This extension enables us to explicitly model concurrency using the power of Petri nets but also to model sequential or data-processing systems using a programming language, leading to much more compact and precise models, especially if several entities in a system behave in a similar manner. That way, CPNs provide a modeling technique that enables us to model complex systems in detail.

As a running example, we consider a gas station. We start with a simple version, and, throughout the paper, we add more features. This will illustrate how to construct a model by incrementally refining and extending it. Moreover, it allows us to show various design patterns for CPNs.

In our example, cars arrive at the gas station, wait to be served or rejected if the station is lacking capacity, and finally leave. We distinguish between regular cars and taxis. Figure 1 shows the first version of a CPN model. In the rest of this section, we explain this example in detail and use it to introduce the CPN formalism. We do not give a formal definition of CPNs, but mention that one such exists and can be found in, for instance, [18,19].

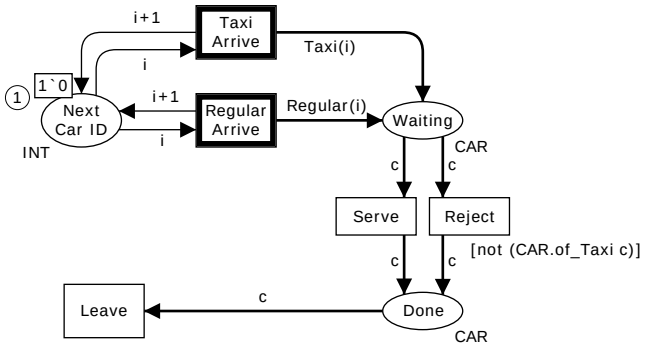

Fig. 1. Simple CPN model of a gas station.

Like for classical Petri nets, the basic components of CPNs are *place*s, *transition*s, and *arc*s. A place serves as a placeholder for the entities in the system and is represented by an ellipse. There are three places in Fig. 1: Next Car ID, Waiting, and Done. A transition represents an action of the system. Graphically, a rectangle represents a transition. The CPN in Fig. 1 has five transitions: Taxi Arrive, Regular Arrive, Serve, Reject, and Leave. Places and transitions are connected by directed arcs, which describe how data flows when transitions are executed, but we defer the exact description for a moment. An arc can only connect a place to a transition or a transition to a place. An arc between two places or between two transitions is not possible. So a CPN induces a bipartite directed graph with places and transitions as nodes.

Each place has a *type* (also known as a color set or sort) that determines which kind of tokens it may contain. This is comparable to how variables have a type in (explicitly) typed languages and is used both to make it easier to understand the model and to catch errors. In Fig. 1, place Next Car ID is of type INT and the places Waiting and Done are of type CAR. This indicates that we can only have integers in Next Car ID and only cars in the two remaining places. Types are declared explicitly in the model using the language CPN-ML, which extends Standard ML [24] with syntax for CPNs. In CPN-ML, the declarations of the types INT and CAR are:

```
colset INT = int;
colset CAR = union Regular: INT + Taxi: INT;
```

The first line indicates that the type INT corresponds to the simple type int (integer). Declarations allow us to give different names to simple types to make the model more readable (e.g., defining a type ID if we were using integers as identifiers). The second line specifies that CAR is a union type, which corresponds to a *datatype* in Standard ML or a disjoint union in mathematics. The idea is that we can have values that are either Regular cars or Taxis. Regular cars and taxis have an associated integer, which we use to be able to distinguish each individual car. Thus, the type CAR contains the values {Regular(0), Taxi(0), Regular(1), Taxi(1), ...}.

Aside from a type, each place also has a *marking*. A marking of a place is a multiset of values over the type of the place. A multiset is like an ordinary set (i.e., the order of elements does not matter), but the same element can occur multiple times. A token is an element of such a marking; that is, it has a value and resides in a place. In the example in Fig. 1, markings of places are shown in a circle and a rectangle near the places, such as the circle containing 1 and the rectangle containing 1‘0 on place Next Car ID. The number in the circle represents the total number of tokens in the place, and the text in the rectangle is a textual representation of the multiset of tokens. In the example, place Next Car ID contains exactly one token and the marking is written 1‘0. We use a backwards apostrophe (‘) to separate the value of a token and the count of how many tokens with that value is part of the marking. The marking in Fig. 1 consists of one token with value 0. If a marking consists of tokens with different values, we separate them with two pluses (++). This allows us to write a marking such as

2‘1++3‘5 to represent the multiset containing two tokens with value 1 and three tokens with value 5. Another example of a marking is 1‘"Hello"++1‘"World" for a marking containing two tokens, one with value "Hello" (i.e., the string Hello) and one with value "World". Places without a marking shown contain an empty multiset which is not shown explicitly. An assignment of markings to all places is a *marking* of the net (or model).

We think of arcs as belonging to transitions, and separate them into *input arc*s and *output arc*s. An input arc connects a place to a transition, and an output arc connects a transition to a place. An arc has an inscription, i.e. an *expression*, written in CPN-ML. Expressions are like expressions in programming languages and may contain constants, functions, and all common arithmetic operators. An expression may contain one or more free *variables* (i.e., it may be an open expression). In Fig. 1, transition Regular Arrive has an input arc from place Next Car ID with expression i and two output arcs—one to Next Car ID with expression i+1 and one to Waiting with expression Regular(i). A place connected to a transition using an input arc is an *input place*, and a place connected using an output arc is an *output place*. In the example, Next Car ID is an input place of transition Regular Arrive, and Next Car ID and Waiting are output places of the same transition. A place can thus be an input and an output place of the same transition. Variables must be declared to be of a certain type. In our example, we have declared two variables:

```
var i: INT;
var c: CAR;
```

Variable i is of type INT and variable c of type CAR. An expression on an arc must have the same type as the type of the place it is connected to or a multiset of the place type; that is, when a value (of correct type) is assigned to all free variables in an expression, it must evaluate to a multiset over or a single value of the type of the place the arc is connected to.

A transition has a natural set of variables, namely the ones occurring on all arcs belonging to it. Each of these variables can be assigned a value from the set represented of its type. For example, the variable i can be assigned the value 0, 1, or 37 as they are all integers. We refer to a transition along with an assignment to each of its variables as a *binding element* (or binding for short). We denote a binding element by the name of the transition and a list of assignments to all its variables in braces. In our example, there are binding elements Regular Arrive⟨i=0⟩, Regular Arrive⟨i=37⟩, Serve⟨c=Taxi(23)⟩, and many others. Note that such potential bindings exist independent of a particular marking; that is, when talking about binding elements we do not look at surrounding places, but only consider the free variables of arcs surrounding the transition. We note that even though variable i occurs on more than one arc connected with Regular Arrive, we only write it once in a binding element. If the transition is clear from the context, we may omit the name of the transition when talking about a binding element.

Given a binding element, we can evaluate the expressions on all arcs belonging to the transition. For example, given the binding element Regular Arrive⟨i=0⟩ in

Fig. 1, the expression i of the input arc from Next Car ID evaluates to 0, the expression i+1 on the output arc to Next Car ID evaluates to 0+1=1, and the expression Regular(i) on the output arc to Waiting evaluates to Regular(0). For the binding element Regular Arrive⟨i=37⟩, the same expressions evaluate to 37, 38, and Regular(37), respectively. As we only write each variable once, it has to have the same value in all expressions surrounding a transition, but it can have other values on other transitions. That means, the scope of a variable in a CPN model is a transition, and information cannot be exchanged between transitions directly. We can think of a transition as inducing a namespace for all variables surrounding it.

Given a model with a marking and a binding, we say that the binding is *enabled* if all input places contain at least the tokens specified by the evaluation of the expression on the corresponding input arc in the binding. In Fig. 1, the binding element Regular Arrive⟨i=0⟩ is enabled as the expression on the sole input arc to the transition evaluates to 0, and the marking of Next Car ID contains a token with value 0. The binding element Regular Arrive⟨i=37⟩ is not enabled in the marking in Fig. 1 as Next Car ID does not contain a token with value 37.

A transition is *enabled* in a marking if there exists at least one binding element which is enabled in the marking. In Fig. 1, we have two enabled transitions, Taxi Arrive and Regular Arrive, as evidenced by the enabled binding elements Regular Arrive⟨i=0⟩ and Taxi Arrive⟨i=0⟩. Enabled transitions are marked using a bold outline. Figure 1 shows that in the initial marking both Regular Arrive and Taxi Arrive are enabled. Each of them has one enabled binding: Regular Arrive⟨i=0⟩ and Taxi Arrive⟨i=0⟩.

If a binding element (or a transition) is enabled, it can *occur* or be *executed*. This has the effect of removing all tokens from input places corresponding to evaluations of expressions on input arcs and producing new tokens on output places corresponding to evaluations of expressions on output arcs. In the example, if binding element Regular Arrive⟨i=0⟩ occurs, we get a situation like the one in Fig. 2. Compared to the situation in Fig. 1, only the marking has changed; the net structure remains unchanged. When Regular Arrive⟨i=0⟩ occurs, it consumes the token with value 0 from Next Car ID and produces a token with value 1 according to the arc expression i+1 in Next Car ID. In addition, it produces a token with value Regular(0) in Waiting, and this place now contains exactly one token with this value. Transitions Taxi Arrive and Regular Arrive remain enabled in this marking, albeit the enabled bindings changed: Taxi Arrive⟨i=1⟩ and Regular Arrive⟨i=1⟩. Furthermore, transitions Reject and Serve are now enabled, both in a binding ⟨c=Regular(0)⟩.

We look at the net structure of a model and its marking as separate things; the marking of the model may change, but the net structure remains the same. The marking of a model before we start simulation is the *initial marking*; for example, in Fig. 1, only place Next Car ID contains a token; this token has value 0. The marking after executing Regular Arrive⟨i=0⟩ is depicted in Fig. 2. We refer to such a marking as the *current marking*. Executing binding Taxi Arrive⟨i=1⟩ yields a new current marking shown in Fig. 3. Execution of a binding element

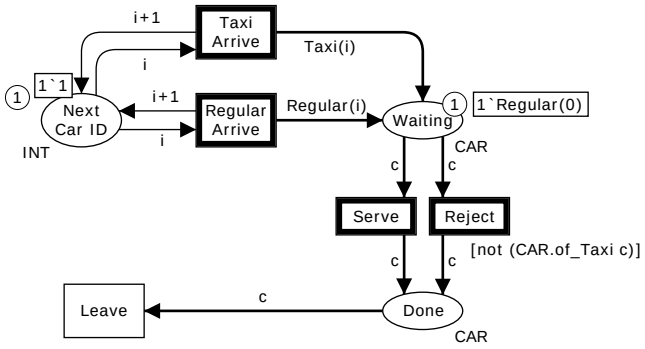

Fig. 2. Gas station model after executing binding element Regular Arrive⟨i=0⟩.

is a *step*. Transitions Taxi Arrive and Regular Arrive, thus, intuitively model that a car of the given type arrives and queues up in Waiting. We use the token in Next Car ID as a counter to number all cars arriving.

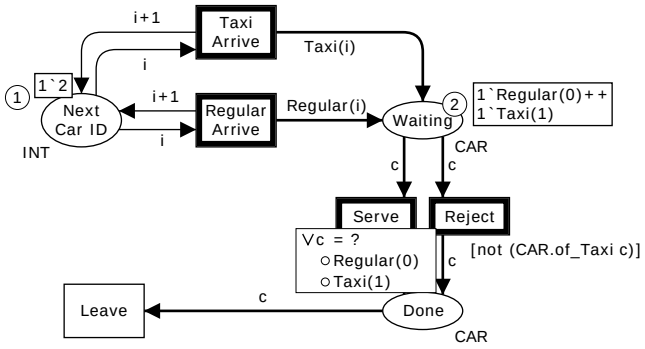

Fig. 3. Gas station model after Regular Arrive and Taxi Arrive have been executed. As shown, transition Serve is enabled in two bindings, ⟨c=Regular(0)⟩ and ⟨c=Taxi(1)⟩.

In Fig. 3, transition Serve is enabled in two bindings, ⟨c=Regular(0)⟩ and ⟨c=Taxi(1)⟩. These two bindings are shown in Fig. 3; the rectangle near the transition shows that c has two possible values enabling Serve.

If we execute the transition in the binding Serve⟨c=Taxi(1)⟩, we obtain the situation in Fig. 4, where the Taxi(1) is removed from Waiting and a token with the same value has been produced in Done. Intuitively, this models the serving of a taxi at the gas station and moving it to a place where it is no longer waiting to be served. Now, the taxi can leave (transition Leave is enabled), leading to a marking similar to Fig. 2 except the marking of Next Car ID is 1‘2 instead of 1‘1.

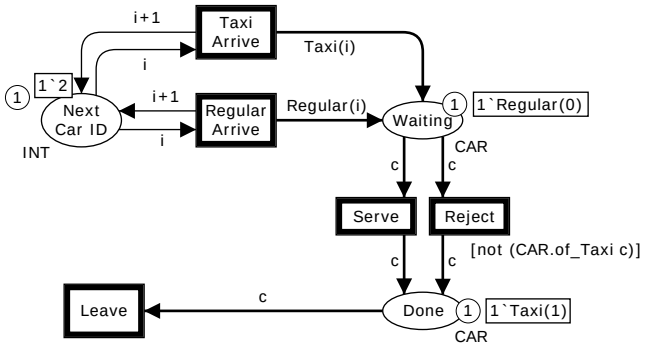

Fig. 4. Gas station model after serving the taxi.

Figures 1, 2, 3, and 4 show one possible sequence of steps. Executing enabled bindings and thus moving from one marking to another is also known as the *token game*. This is the mechanism used when simulating the CPN. Note that whenever a state has multiple enabled bindings, one needs to pick one of these bindings.

If we reconsider the marking in Fig. 3, we see that transition Reject has an inscription in squared brackets to the left of it, namely [not (CAR.of_Taxi c)]. This is a *guard*. A guard defines an additional constraint that must be fulfilled before a transition is enabled; that is, it is a Boolean expression that needs to evaluate to true in addition to the earlier requirements. When a guard is not shown, it implicitly always evaluates to true. A guard may also contain free variables, and they are considered in the same way as free variables on arcs surrounding a transition when considering bindings of the transition. In our example, the guard uses a function, CAR.of_Taxi, automatically defined for union types, which returns whether the parameter given is a Taxi (regardless of the integer value). Thus, the guard evaluates to true if the value of c is not Taxi. This models that we do not wish to reject taxis, for example, because they bring a lot of business. This semantics of a guard is also reflected in the enabled bindings of transitions as shown in Fig. 5, where the binding Reject⟨c=Regular(0)⟩ is enabled, but Reject⟨c=Taxi(1)⟩ is not, even though the token required is present.

3 Hierarchical Modeling

In this section, we present an approach to extend CPNs with *hierarchy*. This approach makes it possible to reflect the hierarchical structure of the system in the CPN model. Hierarchical CPNs simplify modeling, thus facilitating the modeling of large and complicated systems. The idea is to decompose a system into a set of *modules*. A module is a CPN with a set of interface places, and it can be used to describe the internal structure of a substitution transition. By showing the substitution transition at the higher level, we can abstract from the inner structure of a module at the lower level.

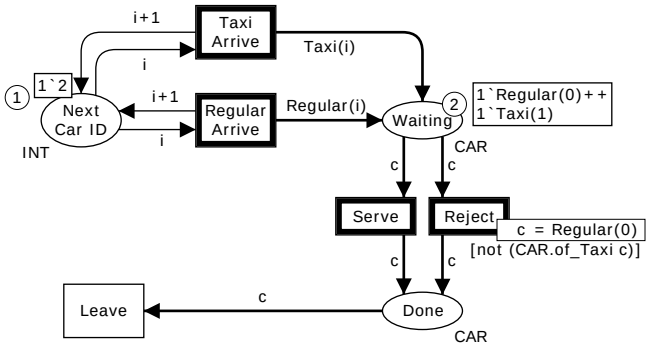

Fig. 5. Gas station model: Enabling of Reject is affected by its guard.

First, we present hierarchical CPNs, as supported by CPN Tools and show how hierarchical modeling can be used to refine our running example. Subsequent, we sketch different approaches of hierarchical modeling.

3.1 Hierarchical CPNs

CPNs, as introduced in the previous section, are suitable to model the behavior of complex systems. The concept of place types allows us to specify the flow of data objects of any data type, and by using guards and arc inscriptions we can manipulate these data objects. The weakness of "flat" CPNs is that they try to capture the system behavior in one comprehensive net and do not represent the hierarchical structure of a system. For example, the CPN in Fig. 1 models the serving and rejecting of cars at the gas station but also how cars arrive and leave the gas station. In other words, the CPN in Fig. 1 is unstructured.

For toy examples like the CPN modeling a gas station, this is not a problem. If a CPN has only few places and transitions, we can lay out the net structure in a way such that the individual parts of the system can be recognized. However, if we model more complex systems such as a more refined version of the gas station example, then the resulting CPN model could have hundreds of places and transitions. Such models cannot be overseen and are, therefore, not suitable for discussing design decisions or implementation details of a system.

In a CPN, we model the elements of a system as places, transitions, and tokens. These elements do not allow us to structure a model. As a consequence, modeling a system by using only places, transitions, and tokens is insufficient. Concepts to abstract from parts of the model are necessary. To this end, models are usually designed following a *hierarchical approach*. The idea is to have several levels of abstraction of the system and to refine elements at higher levels into more detailed elements at lower levels. That way, also the design of large and complex systems becomes manageable, because designers usually concentrate on a single aspect of a system and extend the model step by step. For example, if we only want to know when a car is rejected at the gas station, then the information

of arriving and leaving cars is not relevant and should be abstracted from. We illustrate the idea of hierarchical modeling by revisiting our running example introduced in Fig. 1.

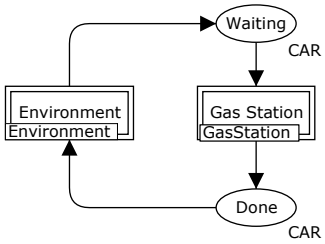

Fig. 6. CPN model of the gas station top module.

We structure the model in Fig. 1 by decomposing it into two modules: (1) the gas station and (2) its environment. The gas station module represents the functionality of the gas station—that is, the serving and rejecting of waiting cars, as modeled by transitions Service and Reject. The environment module models the arrival and leaving of the cars, as modeled by transitions Taxi Arrive, Regular Arrive, and Leave. The interface between these two modules can be specified by places Waiting and Done. We refer to such a place as a *socket*. Both socket places are of type Car. A token in Waiting models a car waiting to be refueled, and a token in Done models a car that has been refueled or rejected. Figure 6 shows the corresponding CPN model of this top-level module. The double-lined rectangles, which are labeled Environment and Gas Station, denote the two respective modules in an abstract manner.

As already mentioned, a module is a CPN consisting of places, transitions, arcs, and tokens. There are two kinds of transitions: *elementary transitions* and *substitution transitions*. An elementary transition is an ordinary transition, as introduced in the previous section. A substitution transition refers to a module. It abstracts from the internal behavior of a module; that is, it considers a module as a black box. Unlike a normal transition, a substitution transition may have internal states and does not need to consume and produce tokens in one atomic action. For example, depending on the underlying module, Gas Station may first consume ten cars from place Waiting, before it produces a token to Done.

A module may contain any number of substitution transitions. These substitution transitions refer to other modules that, in turn, may contain transitions referring to other modules. There can be an arbitrary many levels as long as no cycles are introduced in the inclusion graph; that is, a module may not (transitively) contain itself as this would correspond to an infinitely large model when we replace each substitution transition by the module it refers to.

The top-level module in Fig. 6 has two substitution transitions Gas Station and Environment referring to modules Environment and GasStation, respectively,

as can be seen from the tag on the bottom of a substitution transition. Figure 7(left) shows the CPN modeling module Environment and Fig. 7(right) the CPN modeling module GasStation.

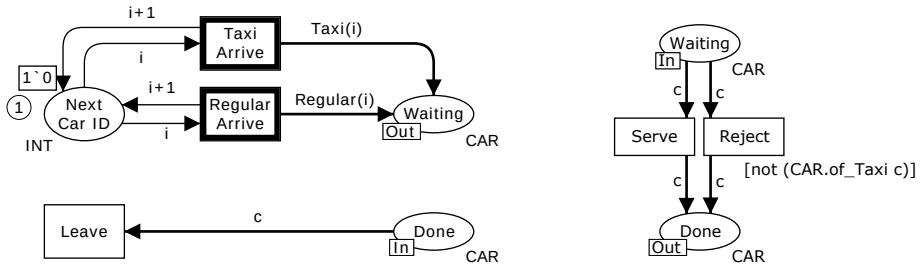

Fig. 7. CPN models of the environment (left) and gas station module (right).

Each of the two CPNs in Fig. 7 has two interface places: Waiting and Done. We refer to such a place as a *port*. To be able to replace a substitution transition by the CPN modeling the module it refers to, we need to relate each socket of the substitution transition to a port of the CPN modeling the module. That way, pairs of places—a port and a socket—are semantically merged into one place. As an example, sockets Waiting and Done in Fig. 6 are merged with the equally labeled ports in Fig. 7(right). On the level of a port, we specify the type of connection; that is, a port has a *type*. In Fig. 7(right), the tag In on place Waiting denotes that this port is of type input. Likewise, the tag Out on place Done shows that this port is of type output. A port can also be of type input and output. In this case, it is labeled I/O.

Replacing a substitution transition by the module it refers to is called the *flattening* of a CPN. The semantics of the CPN in Fig. 6 corresponds to the flat CPN in Fig. 1.

Another advantage of decomposing a system into modules is that it enables us to *reuse* existing functionality. That is, the same module can be used several times in a model if necessary. As a result, multiple substitution transitions may refer to the same module. To cope with this, we distinguish between a *module definition* and a *module instance*. Whereas a module definition serves as a specification of a CPN model, a module instance can be seen as an individual copy of the module definition. For example, we could extend our model in Fig. 6 and connect sockets Waiting and Done with an additional gas station, say Gas Station 1. In this case, we have two substitution transitions, Gas Station and Gas Station 1, both referring to the same module definition, GasStation. However, each substitution transition would be replaced by an individual module instance—that is, a separate copy of the CPN shown in Fig. 7(right).

Finally, we illustrate how hierarchical modeling simplifies the design and, in particular, the refinement of a system. Consider the module of the gas station,

as depicted in Fig. 7(right). Cars are modeled as tokens in place Waiting and are waiting to be served. In the current model, a car may be waiting to be served and after a while be rejected. This is not desirable. Therefore, we extend the model as follows: the gas station has some waiting space where cars are queueing. If the gas station has too little capacity, arriving cars will be rejected. However, once a car enters the queue at the gas station, it will be eventually served.

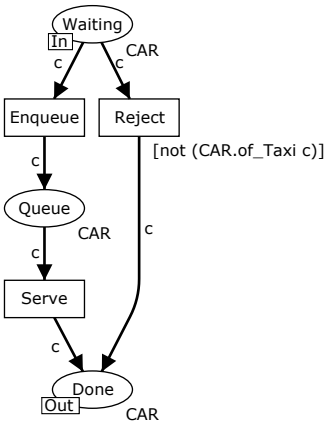

Fig. 8. Improved CPN model of the gas station module.

To modify the model, we only need to refine the CPN modeling the gas station module (see Fig. 7(right)). Figure 8 shows the resulting model. An arriving car is either rejected right away or put in the queue. As in the previous models, taxis are not rejected. Place Queue models the queue at the gas station.

The advantage of hierarchical modeling is that we need only to refine the module of the gas station. As the environment is not affected by the change, we do not have to touch the respective model. This example shows that by using hierarchical CPNs, designers can focus on single aspects of a model.

3.2 Approaches

There are two prominent approaches to obtain a hierarchical model for a system: the *top-down approach* and the *bottom-up approach*.

In the top-down approach, we start at the highest level of abstraction. and *decompose* the system into modules. Each module is considered as a black box, and only the relationship between the modules, which is modeled as an interface, is relevant. In subsequent steps, we can consider each module as a black box and refine it into a set of submodules. We can repeat this procedure until we have reached the desired degree of abstraction.

The bottom-up approach starts at the lowest level of abstraction and works in the opposite direction as the top-down approach. At this level of abstraction, we

describe the elementary modules in detail. These modules are then *composed* to form a compound module. This composition step is repeated until we reach the highest level of abstraction at which the system is modeled as a single module.

Hierarchical development of systems is widely used, and all modern programming languages offer facilities to support this approach. For example, functionality can be structured by developing a class hierarchy and by implementing procedures to decompose complex functionality.

Hierarchical CPNs, as presented in this section, have been formalized in [18,19] and implemented in CPN Tools. CPN Tools supports top-down and bottom-up design. Another approach for defining modules is to specify the module interface as a set of *transitions*. Flattening a hierarchical Petri net then corresponds to fusing equally labeled interface transitions, an established concept in the Petri net literature. Whereas place fusion models asynchronous communication—that is, sending a token by one module is not synchronized by receiving this token by another module—transition fusion models synchronous communication. Synchronous communication is similar to invoking methods in object-oriented programming and is the dominant composition approach in process algebraic approaches [8,11]. Transition fusion is not supported by CPN Tools.

A different approach to introduce hierarchy in Petri nets is the *nets-in-nets paradigm* proposed by Valk [30]. Whereas traditionally tokens are passive, tokens in this paradigm can be active. One can think about such tokens as agents rather than data containers. As an agent has behavior, the token modeling this agent can represent a Petri net again. That way, the nets-in-nets paradigm supports the modeling of hierarchy. The Renew tool [21] supports the modeling, execution, and analysis of a particular instance of the nets-in-nets paradigm.

4 Simple CPN Patterns

In this section, we introduce some simple modeling *patterns* that can be used frequently. The patterns make it possible to model constructs not natively possible using the CPN formalism. As indicated in the introduction, the idea to provide patterns for modeling in terms of CPNs was first proposed in [26] where 34 patterns were identified. The patterns are briefly described in the Appendix of the paper. Subsets of these patterns can also be found on the CPN Tools Web page[2] and in books such as [6,15,18,19]. However, these publications do not explicitly identify and name these patterns.[3]

In this section, we look at patterns for bounding the number of tokens that can be in a place at once, for imposing an order of how tokens are added to and removed from places, for checking the number of tokens in a place, and for folding equal or similar net structures into one generic copy. In the next section,

[2] See `http://cpntools.org`.

[3] Note that patterns refer to frequently recurring modeling problems and their solutions. Therefore, patterns are always based on earlier work and not intended to be original.

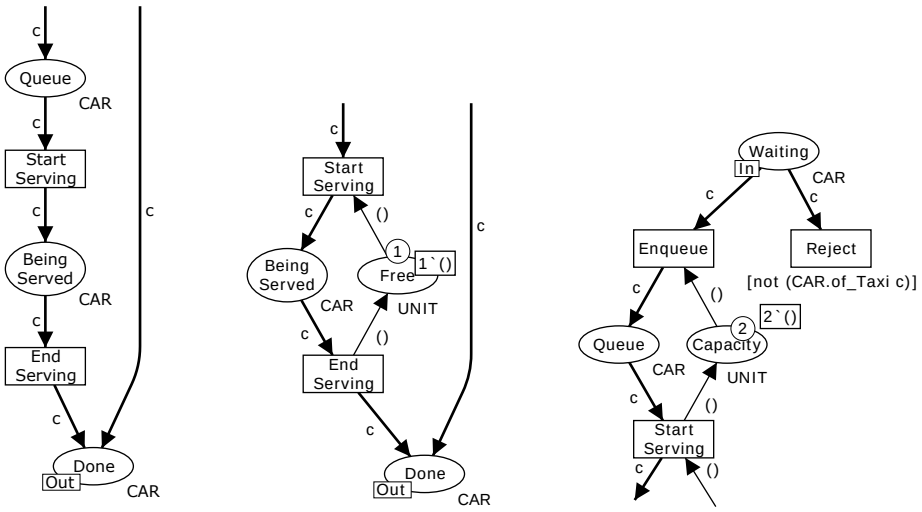

Fig. 9. CPN models of the gas station module with non-instantaneous serving of customers (left), with only one pump (middle), and with bounded queue size (right).

we present more advanced patterns to model more complex constructs that occur less often.

Unlike in [25,26], we will not be using a strict format for describing the patterns. Instead, we use examples based on the hierarchical gas station example from Fig. 6 with the environment module from Fig. 7(left) and the gas station module from Fig. 8. In our examples, we replace only the gas station module.

4.1 Bounded Places using Complement Places

Figure 8 models the serving of a car as a single transition, Serve, indicating that this action is instantaneous. As it actually takes some time to serve a customer, this is an oversimplification. For this reason, we split this action into two actions: start serving and end serving. We then obtain the situation in Fig. 9(left). Here, we have a transition Start Serving, moving a car from place Queue to Being Served, and End Serving, moving a car from place Being Served to Done. The model now reflects that serving a customer is not an instantaneous action, but it introduces a new problem. Consider what happens if we have three cars in Queue. Now, we can Start Serving all three of them. This does not really make sense if the gas station has only one pump; rather, the cars should remain in the queue until the pump is free, at which point we start serving the next one in line. We thus need to bound the number of cars (tokens) that can be in place Being Served at any point in time. Therefore, we look at modeling patterns to limit the number of tokens in place Being Served.

The simplest way to bound the number of tokens in a place is to introduce a *complement place* (or anti place). The idea is to ensure an invariant, namely

that the total number of tokens in this place and its complement place together is a constant. A place, which can at most contain a predetermined number of tokens, is *bounded*. A bounded place has a certain *capacity*. Bounded places are related to the concept of *safe places* in low-level Petri nets.

In Fig. 9(middle), we have added a new place Free with type UNIT. The type UNIT is declared to be:

```
colset UNIT = unit;
```

The unit type is a type which contains only one element, (), and simulates the behavior of (black) tokens of classical Petri nets: tokens of this type are indistinguishable, so we only care about the number of tokens. Whenever we produce a token in Being Served, we remove a token from Free, and vice versa. This ensures that the number of tokens in Being Served and Free remains unchanged and hence that Being Served contains at most one token. We can think of this as consuming a right to produce a token in Being Served when we need it and returning the right when we no longer need it (when we consume a token).

In general, we model a complement place for a place by mirroring all arcs connected to the original place. That is, whenever there is an arc from a transition to the original place, we add an arc from the complement place to the transition; and whenever there is an arc from the original place to a transition, we add an arc from the transition to the complement place. The type of the complement place can be UNIT, as in the example, or it can be another type if we want to model a complement place for bounding multiple places or for bounding tokens of a particular value. For example, assume that we need to adapt in Fig. 9(middle) to model the situation that there are five pumps: two diesel pumps, two petrol pumps, and one pump for electric cars. In this case, the complement place initially has five tokens making sure that each pump can only be used for one car at a time. If the pump for electric cars is busy while the other four pumps are free, then it is impossible to serve another electric car. This can be ensured by using a different type for place Free, for example,

```
colset Fuel = with diesel | petrol | electric;
```

Moreover, type CAR needs to be extended to indicate the type of fuel a car requires and the arc inscriptions need to be adapted accordingly. However, the principle is the same: We consume a token from the complement place whenever we produce one token in the original place. Likewise, we produce a token in the complement place whenever we consume one token from the original place. In Fig. 9(middle), the transitions unconditionally produce/consume tokens in/from the original place, so we do the same for the complement place. In more advanced examples, we may produce/consume a varying number of tokens in/from the original place depending on the binding of the transition; that is, we need to make the expressions on arcs from/to the complement place reflect this.

The pattern to add a complement place to bound the number of tokens in another place is used frequently. Consider for example, locking in databases, kanbans in production systems, and message buffers in middleware.

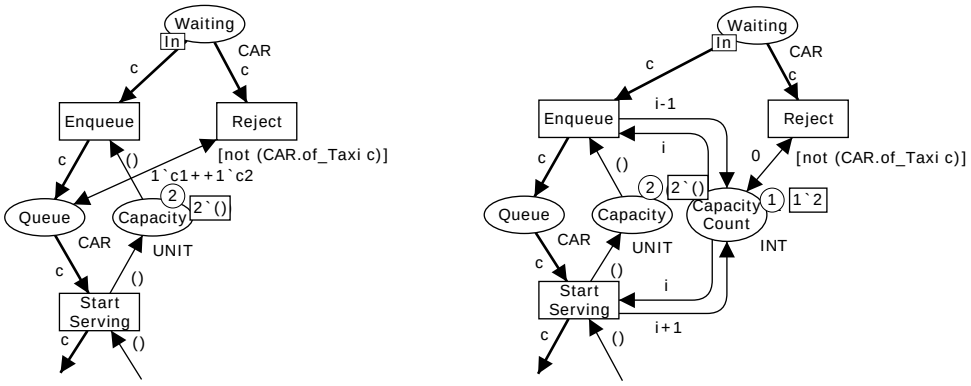

Fig. 10. Two gas station modules which reject customers only when there is no capacity for them.

4.2 Inhibitor Arcs using Counter Places

A gas station needs to reject customers if it does not have the capacity to serve them; otherwise, they are added to the queue. Our model does not reflect that, as customers are rejected nondeterministically. Transition Reject should be enabled only if the Queue is full. As a first attempt, we add a complement place, Capacity, for place Queue and obtain the situation in Fig. 9(right). Now, the queue size is limited to 2. At this point, we start unconditionally rejecting customers. The model is still not correct, though, as we may also reject customers before the limit is reached. We thus want to disable Reject if Capacity contains more than zero tokens (i.e., if the queue is not full).

We can model such a condition in various ways. An arc that inhibits enabling of a transition if a place contains more than a specified number of tokens is an *inhibitor arc*. An inhibitor arc can in particular be used to test whether a place contains zero tokens, which is also known as *zero-testing*. Some Petri net formalisms contain inhibitor arcs natively, but for CPNs, this is not necessary, as we can easily model them. The simplest way to model an inhibitor arc in our example is to add an arc between Queue and Reject checking that Queue contains two tokens, like in Fig. 10(left). This arc contains an arrowhead in both directions. This is a shorthand for an arc in both directions with the same expression. In CPN terminology, such an arc is called a *double arc* and in the literature, it is also known as a *test arc* or *read arc*.[4] The double arc between Queue and Reject allows us to test that two tokens are present in Queue without modifying them.

This solution works only for bounded places and is difficult to scale with the bound of the place (as we need a variable for each token). Instead, we introduce

[4] There are subtle differences between test and read arcs depending on the exact transition execution semantics; we will not elaborate on this in this paper.

a general way of modeling inhibitor arcs here and another solution in Sect. 4.3. The basic idea of the general solution is to introduce a *counter place*. This place counts the number of tokens in a place. It is similar to a complement place, but we store the number of tokens as an integer rather than indistinguishable tokens.

In Fig. 10(right), we have added a counter place Capacity Count counting the number of tokens in Capacity. The type of Capacity Count is INT. This place is similar to place Next Car ID used in Figs. 1 and 7(left). However, now we increase the value whenever we add a token to Capacity (Enqueue) and decrease it when we remove one (Start Serving). The double arc between Reject and Capacity Count tests that the remaining capacity is 0; that is, cars are only rejected if no capacity is left.

Place Capacity is redundant after adding Capacity Count. Recall that Capacity was introduced as a complement place to bound the number of tokens in place Queue. We can also bound the number of tokens in place Queue by inhibiting the enabling of Enqueue when it contains more than one token—that is, by using a guard.

Fig. 11(left) shows the situation where Capacity is a counter place for place Queue; that is, the value of the token in Capacity corresponds to the number of tokens in Queue. Both Enqueue and Reject use this information to block if needed; that is, the arcs between Capacity and Enqueue update Capacity and serve as inhibitor arc. The guard added to Enqueue ensures that Enqueue is only enabled if the number of tokens in Queue is less than MAX_CAPACITY. MAX_CAPACITY is a constant defined in the declarations of the model as:

```
val MAX_CAPACITY = 2;
```

This declaration allows us to use a symbolic constant instead of writing the same value in multiple places, which improves the readability of the model and makes it easier to change the value of the constant if necessary. We have also added a double arc between Capacity and Reject and an extra clause to the guard of Reject checking that the value of the token is greater than or equal to MAX_CAPACITY. We use a comma (,) to separate clauses in the guard. This acts as a shorthand for logical and (which in CPN-ML is written as andalso).

4.3 FIFO-Places using Complement Places or Lists

Any variant of the gas station we have looked at until now (see Figs. 7–11) serves cars in a random order. For most gas stations, this does not reflect reality; rather, most gas stations serve customers using a first-come, first-served policy, and we want to make our model reflect this. A place from which tokens are removed in the same order as they are added is a *FIFO-place* (first-in, first-out place).

Our first idea is to model each location of the queue explicitly using a complement place to ensure that each location has at most one car. This is shown in Fig. 11(right). The only new thing is that we have a double arc between each location of the queue and transition Reject. These arcs test that every location in the queue is filled before rejecting customers. We have used separate variables

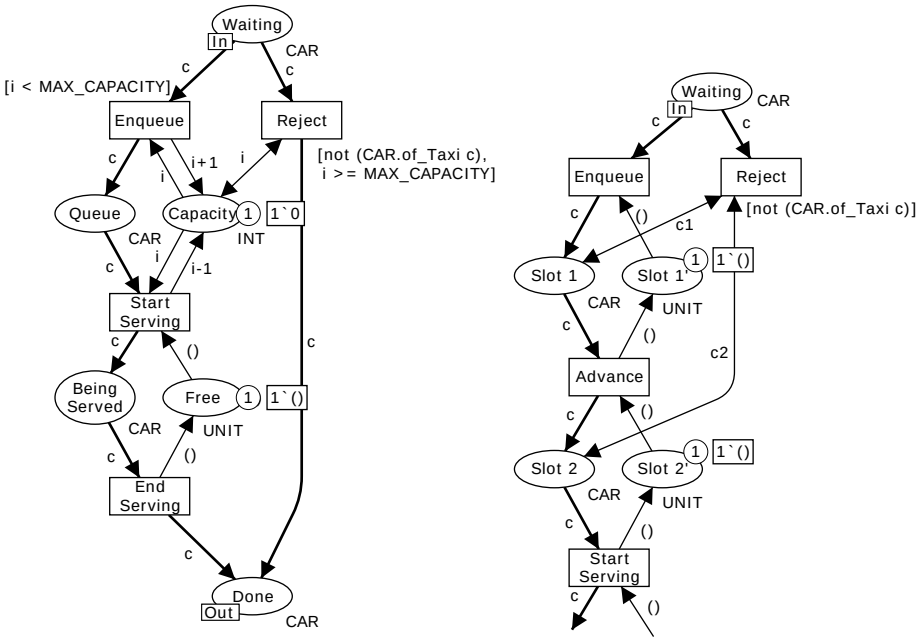

Fig. 11. Improved gas station module which only rejects customers when there is no capacity for them (left), and gas station module which serves customers in the order they entered the queue (right).

for each double arc, as we are not requiring that each location contains the same car (which in this model is impossible). Each car arriving at Waiting can only enter the queue if the first spot is vacant. Likewise, a car in the first spot can only progress if the second spot is vacant. So as soon as a car has entered the queue, it is guaranteed to be served in the order it appeared. This is in contrast to place Waiting where cars are not ordered.

The queue in Fig. 11(right) serves its purpose but has some problems. First, the construction is not scalable. Compared to Fig. 11(left), where we could change the capacity of the queue by just changing the value of a constant, we have to add two places, a transition, and four arcs for each slot we want to expand the queue with. Second, it seems a bit excessive that a car has to drive through each spot in the queue explicitly even if it is the only car. Instead of using the net structure to express what is essentially a data-structure, we—for the first time in this paper—use that CPNs support *list* types. The idea is to represent the cars in Queue as a queue rather than a multiset. In CPN-ML, a queue is easiest modeled using a list.

Figure 12(left) shows an implementation of a queue using lists. It is easier to compare this module with the one in Fig. 11(left) rather than the one in Fig. 11(right). We have changed the type of Queue from CAR to CARS and use

some more elaborate expressions on the arcs around the place. The type of Queue is a list of cars declared as:

```
colset CARS = list CAR;
```

Place Queue has an initial marking, 1‘[], indicating that the place contains a single token, an empty list (which is written as [] in CPN-ML). Whenever we produce a token in Queue in Fig. 11(left), we now, in Fig. 12(left), replace the token in Queue with a new list consisting of the previous list with the new element appended at the end. This is written as cs ^^ [c] in CPN-ML (i.e., append the singleton list [c] to the end of list cs). We have also added an arc opposite the original arc to get access to the previous value (cs). We need to introduce a variable cs that is declared as:

```
var cs: CARS;
```

Where we previously removed an arbitrary token from Queue, we now take the first element of the list and return the tail of the list to Queue. We do this by using pattern matching, which is a powerful mechanism CPN-ML [19] inherits from Standard ML [24]. The expression c::cs assigns the head of the list to c and the tail to cs. A transition using such an expression on an input arc is only enabled when the list on the corresponding place contains at least one element.

The list structure also enables an alternative realization of the inhibitor arc pattern. As all tokens are inside a single data-structure, we can test the number of tokens without maintaining a separate counter. We have eliminated Capacity and changed the guards of Enqueue and Reject to refer to length cs, which returns the number of elements in list cs. Thus, if we have imposed an ordering of elements on a place, we can count the elements directly.

Another advantage of using the pattern in Fig. 12(left) is that we can use any data structure to impose any ordering of elements. For instance, we can change the inscription on the arc from Enqueue to Queue to c::cs to add the new car to the head of list cs, thereby implementing a *stack place* from which tokens are removed in last-in, first-out order. We can also implement *priority queue place*s by sorting the list according to a priority upon insertion. See [26] for concrete examples.

Figure 12(left) (but also the other CPN models modeling a queue for the cars waiting to be served) has the problem that tokens may be queueing in place Waiting. When the queue has reached its maximal capacity and a taxi arrives via port Waiting, then Enqueue and Reject are unable to handle the taxi. One can handle this in different ways. However, in case of a stochastic arrival process, it is impossible to ensure that there is a free position in the queue for taxis.

4.4 Folding Identical Net Structures

Most of the gas stations considered in this section (Figs. 9–12) had only one pump. We can easily change that by increasing the number of tokens initially in place Free, but only if we do not care which pump a customer uses. Now,

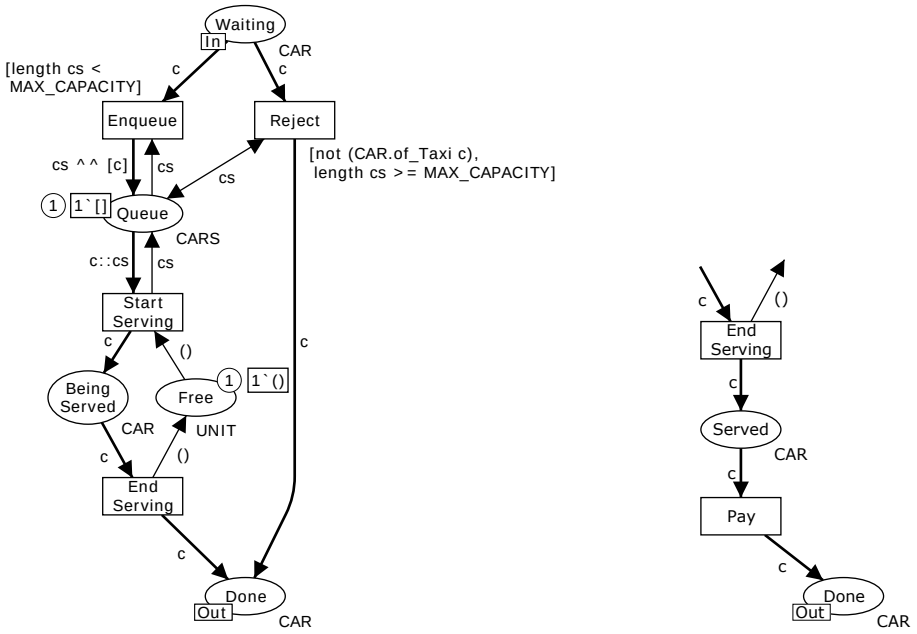

Fig. 12. Improved gas station module which serves customers in the order they entered the queue by using lists instead of the net structure (left) and gas station module which requires users to pay (right).

assume that we add an extra step after refueling for paying for the fuel, as seen in the fragment in Fig. 12(right). This model does not preserve the information for which pump the customer has to pay.

It is possible to retain the information about which pump was used by duplicating the structure representing the pump and the payment procedure; a model doing so is shown in Fig. 13(left). Now, depending on which pump we chose to use initially, we have to execute either transition Pay 1 or Pay 2, thereby ensuring that each car pays for the gas it actually refueled.

Naturally, duplicating net structure is rarely the best solution. If the focus of the model is the geographical distribution of cars during a day, it may be a good choice as we have a one-to-one correspondence between places of the model and physical locations. Here, we are not interested in that, so it may be better to *fold* the two paths in Fig. 13(left) into one, thereby avoiding copying and making it easier to subsequently add more pumps or to change the behavior of all pumps (e.g., we may want to keep the reservation of a pump until the customer has paid, to make it even easier to match the pump and customer to the amount of gas purchased). A folded version of the model in Fig. 13(left) is shown in Fig. 13(right). In the remainder of this section, we explain folding using this example.

Folding means that we add to all places and expressions another component representing the identifier of the folded value. In our example, we define a new type, PUMP, and a variable of that type as:

```
colset PUMP = index pump with 1..2;
var p: PUMP;
```

An index type consists of a name (here pump) index by a set of integers (here 1..2); that is, allowed values of tokens in PUMP are {pump(1),pump(2)}. Consider using index types when you mathematically would have used index values like $pump_1, pump_2$. We then have to define types for all places we wish to fold (here Being Served, Free, and Served). We define these types as Cartesian products of the identifier used to fold and the original types. If the original type was UNIT, there is no need to keep it in the Cartesian product and we can just use the identifier type. We have replaced CAR on Being Served and Served with CARxPUMP and UNIT on Free with PUMP. Type CARxPUMP is declared as:

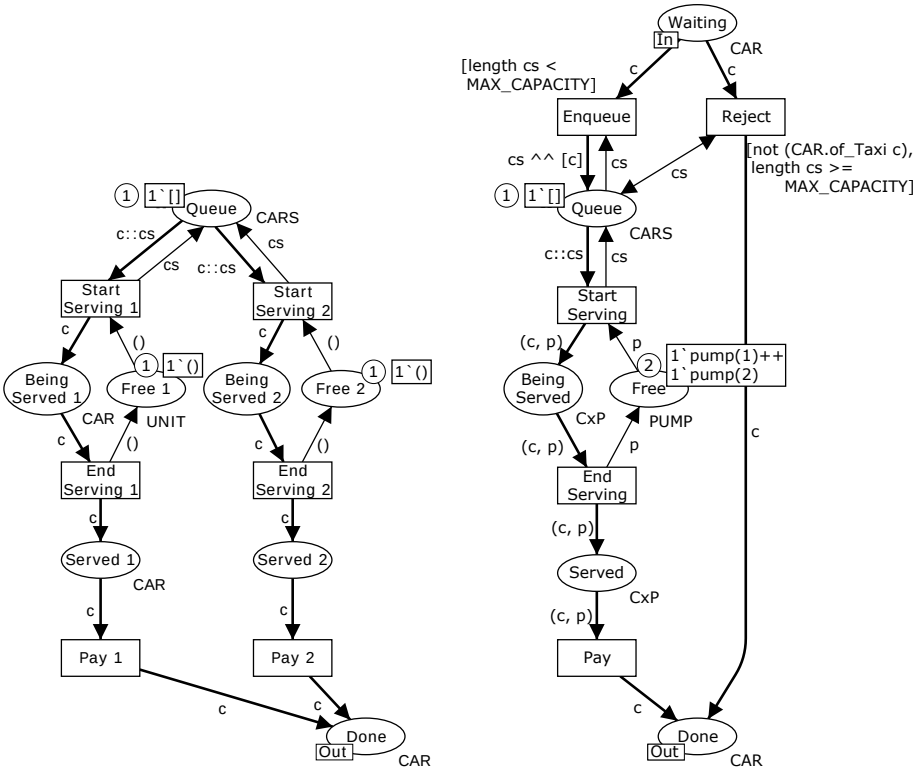

Fig. 13. Gas station modules which model the situation where payments are linked to the pump used. Two alternative modules are shown: with (left) and without (right) replicated net structure.

```
colset CxP = product CAR * PUMP;
```

This is the syntax for declaring a Cartesian product of preexisting types. We can also declare Cartesian products of more than two types by adding them at the end. We use the naming convention of using the original types (or the first letters) separated by a lowercase letter x, but sometimes it may be more useful to give it a more descriptive name if the product has a meaning in the domain of the model. Changing the types of places also requires us to change the initial marking; in our example, this is simple, as only Free has a nonempty initial marking. We add an initial marking of 1‘pump(1)++1‘pump(2), specifying that initially both pump(1) and pump(2) are free for use. We also have to update all arc expressions. In Fig. 13(right), we just carry around the pump id and the car id, changing all inscriptions consisting of c to (c, p)—a pair of a car and a pump—and all inscriptions consisting of () to p, the id of the pump.

Figure 14 shows a fragment of a slightly modified version of the gas station, illustrating that we now only have to change the behavior once to change it for all pumps. In this version, a pump is occupied until a customer has paid. We have executed some steps of the model; for example, car Regular(7) is currently being served at pump(1), whereas car Regular(3) is done and is about to pay for the fuel taken from pump(2). There is no available pump at this time (i.e., Free contains no tokens), and car Regular(5) has been served and paid (or refused service) and is now done.

In the examples we have seen here, we have treated each customer equally regardless of the pump they use, but we could also discriminate depending on the pump they use—for example, by inspecting the pump in the guard—introducing

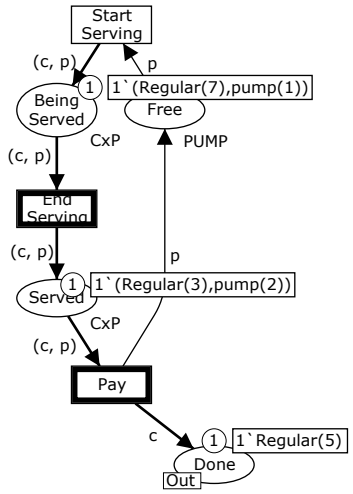

Fig. 14. Fragment of modified gas station module which requires users to pay for their own usage.

different paths depending on which pump a car uses. In fact, the entire model can be seen as a folded model in which the same procedure is shared for all cars. All cars are treated almost the same, except that taxis are never rejected, as seen by the guard of transition Reject. When we have a finite number of objects having the same behavior, such as the two pumps, it is convenient to share the net structure among them to avoid net replication. When we have an unbounded number of objects, such as all cars, it is not only convenient, but even necessary, as we would (theoretically) have to replicate the net structure an infinite number of times to be able to distinguish them.

5 Advanced Modeling Concepts

The patterns, we presented in the previous section, cover constructs that frequently occur in models of systems and processes. In this section, we present two additional patterns: message broadcast and region flush [25,26]. These patterns are more advanced than the previously presented patterns and cover constructs that occur less frequently in models. Before presenting these patterns, we introduce the concept of prioritized transitions, as supported by CPN Tools version 3.0, and illustrate how this concept can simplify the modeling of systems.

5.1 Extending Transitions with Priorities

A CPN may have a reachable marking in which several conflicting transitions are enabled. We refer to this situation as a *nondeterministic choice*. The marking in the gas station module shown in Fig. 15(left) is an example of a nondeterministic choice. The token in place Waiting models a waiting, regular car. This marking constitutes two enabled bindings, one enabling transition Enqueue and the second enabling transition Reject. Which of these two transitions fires is not predetermined. In reality, this would mean that an arriving car may be rejected even though the queue is not full yet. Because this situation is not desirable, we need to adjust the model such that in the situation shown in Fig. 15(left) always transition Enqueue fires. Earlier, in Sect. 4.3, we resolved this problem by adding an inhibitor arc between Queue and Reject (see Fig. 12(left)). In the following, we present another solution by prioritizing the firing of transition Enqueue over the firing of transition Reject.

Priority or *prioritized transitions* is supported in CPN Tools from version 3.0 and onwards. The modeler can assign a priority to each transition: *P_HIGH*, *P_NORMAL*, and *P_LOW*. The default value is *P_NORMAL*. Alternatively, it is also possible to specify the priority of a transition as an integer, where 0 represents the highest priority and larger numbers lower priorities. The built-in priorities, P_HIGH, P_NORMAL, and P_LOW correspond to the integer values 100, 1,000, and 10,000, respectively. The semantics of the model are defined by calculating first all enabled transitions ignoring priority and then considering only transitions with the highest priority. If there are multiple enabled binding

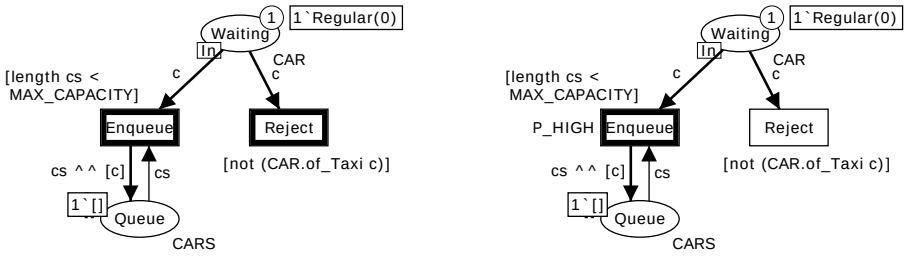

Fig. 15. CPN model without priority (left) and where Enqueue has a higher priority than Reject (right).

elements having the highest priority, then one of them is nondeterministically chosen.

Figure 15(right) results from Fig. 15(left) by assigning value P_HIGH to transition Enqueue—all other transitions have the default value P_NORMAL, which is not explicitly shown in Fig. 15(right). As a result, only transition Enqueue is enabled in the marking shown in Fig. 15(right), because transition Reject has a lower priority than Enqueue.

The example illustrates the advantage of extending CPNs with priorities, namely simplicity of the model. The interplay of priorities is a *global* property of a CPN; that is, assigning a priority to a single transitions may affect the enabling of all other transitions. Therefore, it is possible that the resulting CPN allows for undesired behavior. As an illustration, suppose we assign *P_LOW* to Reject and *P_NORMAL* to all other transitions. Having the lowest priority of all transitions, Reject can only fire if it is enabled (in the setting without priorities) and no other transition is enabled. However, this is never the case, because the environment module can continuously produce tokens in Waiting. As a result, Reject is dead.

Priorities simplify the modeling, but they do not increase the expressiveness of CPNs. Every CPN with priorities can also be modeled as a CPN without priorities. In Fig. 15(right), we can remove the priority from transition Enqueue if we use an inhibitor arc and extend the transition guard of Reject, as shown in Fig. 12(left). In general, expressing priorities between transitions without using the concept of transition priorities is nontrivial, in particular, if the transitions have disjoint presets.

5.2 Message Broadcast

Sometimes we need to model sending of a message to an unknown number of objects. Such a task is referred to as a *message broadcast*. It is particularly useful for modeling systems where many participants interact with each other (e.g., interorganizational business processes) and for message protocols. We illustrate this pattern with the following modification of the gas station. Suppose that there is a promotion at the gas station and every car driver who has refueled

her car gets a voucher before paying for the gas. The difficulty is that we do not know the number of car drivers. The promotor would go to each car driver and hand over the voucher. Figure 16, which is an extension of Fig. 13(right), shows how we can model such a broadcast as a CPN.

Firing transition Start starts the procedure. First, we identify all car drivers who should receive a voucher. To this end, transition Read stores a copy of all tokens of place Served in a list in place Receivers. Each list entry refers to one car driver who will receive a voucher. The guard of transition Read ensures that the same token cannot be added twice to the list. Observe that this construction is independent from the number of tokens in place Served. By assigning a high priority to transition Read, we can make sure that Begin Broadcast can only fire if there are still tokens in place Served that have not been added to the list in place Receivers. When the list is complete, Begin Broadcast fires and the broadcast starts. Note that transition Pay cannot fire, because Pending is unmarked. Firing Begin Broadcast produces the length of the car list cs in Pending. Furthermore, as variable cs is a list of cars and Vouchers is of type CAR, cs is unfolded and for each list entry a single token is produced in Vouchers; that is, CPN Tools automatically converts the list cs into a multiset of tokens for place Vouchers. Transition End Broadcast models the distribution of the vouchers to each car driver. After all car drivers have received a voucher, there is a token with value 0 in place Pending, thus enabling Pay and Start. If Start fires again, then every car driver receives a second voucher.

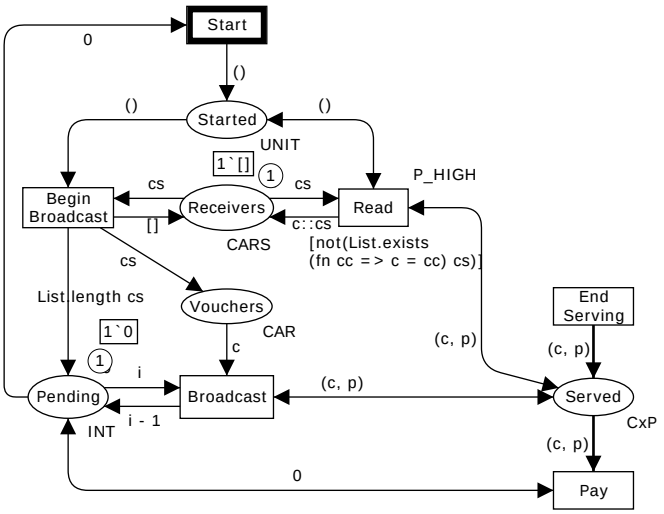

Fig. 16. Broadcasting vouchers to all car drivers who refueled their cars.

5.3 Region Flush

Suppose that the gas station attendant is eager to stop working in time. Every day at 6 p.m. he serves only the cars that are being refueled; all cars in the queue will not be served anymore and have to drive on. To model this as a CPN, we must move all tokens from place Queue to place Done and also reset the value of place Capacity to 0, thereby making sure that transition Start Serving does not fire while tokens from Queue are moved to Done. This is trivial if we have modeled the queue using lists as in Fig. 12(left). However, when modeling the waiting cars in the queue as individual tokens, things become more involved. Figure 17(top) extends the model in Fig. 11(left) with this functionality. Transition Close models the closing of the gas station. It removes the token from place Capacity. This token is needed to learn the number of cars in the queue and to prevent transitions Enqueue and Start Serving from firing. After the queue has been flushed—that is, all cars have been moved to Done—transition Open models the opening of the gas station by initializing place Capacity again.[5] Only then transitions Reject and Start Serving can become enabled. In case we need to flush more than one place, we must copy the pattern accordingly.

Removing tokens from a part of a CPN is referred to as *region flush*. The idea is to disable the transitions in the respective part (i.e., the region) of the CPN while removing all tokens from the places in the region. Afterward, the region is optionally reset.

Using the concept of prioritized transitions, as introduced in this section, we can generalize the model of a region flush. The respective CPN model is shown in Fig. 17(bottom). Transition Flush reads a Boolean false from place Open, moves a car token from place Queue to place Done, and decrements the value of Capacity. As only Flush has a high priority, it can fire until the queue is empty without being in conflict with any other transition. Transitions Close and Open model the closing and opening of the gas station. Close produces a Boolean false in place Open. If there is at least one car in the queue, then Flush is enabled (because of its priority). Only after the queue has been flushed, transition Open can change the Boolean in place Open to true, modeling that the gas station opens again.

The advantage of using the pattern with priority in Fig. 17(bottom) is that it is independent of the presence of complement place Capacity. In contrast, Fig. 17(top) relies on place Capacity as it requires knowledge about the number of cars to be removed; otherwise, we would not know when we have moved all waiting cars to place Done and, hence, when we can open the gas station. Place Flush is connected to place Capacity to update this complement place; it is not used to see how many cars need to be flushed.

[5] When adding explicit time, we can model that the gas station opens again at a particular time (e.g., 9 a.m. the next day). Transition Open is not supposed to fire immediately after closing the gas station; this needs to be governed by the time concept explained in the next section.

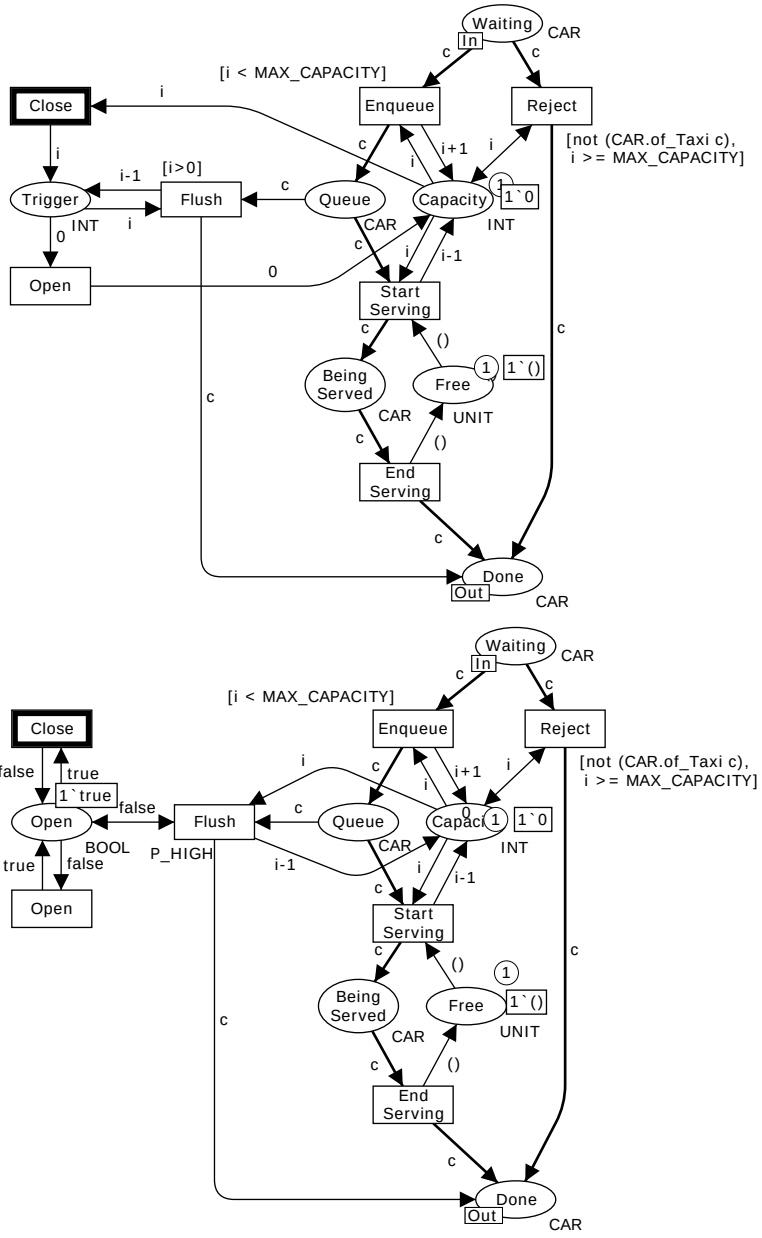

Fig. 17. Removing all cars from queue without (top) and with (bottom) priority.

In Fig. 17, only one place is flushed (place Queue). The construct needs to be copied per place. This may become quite involved, therefore, workflow languages such as YAWL support this natively (see the cancellation pattern in [4,32]).

6 Modeling Time

Time is an important aspect in many systems. Let us, for example, consider the gas station obtained by combining the modules in Figs. 6, 7(left), and 13(right) (shown together as Fig. 18). In this example, we have modeled the serving of customers as two transitions, Start Serving and End Serving, indicating that the action of serving is not instantaneous. However, serving customers is an atomic action in the sense that neither the car nor the pump can be used for anything else during the feat. It would be more elegant to model serving as an atomic action which takes time. The action of paying for the gas should also take time. In

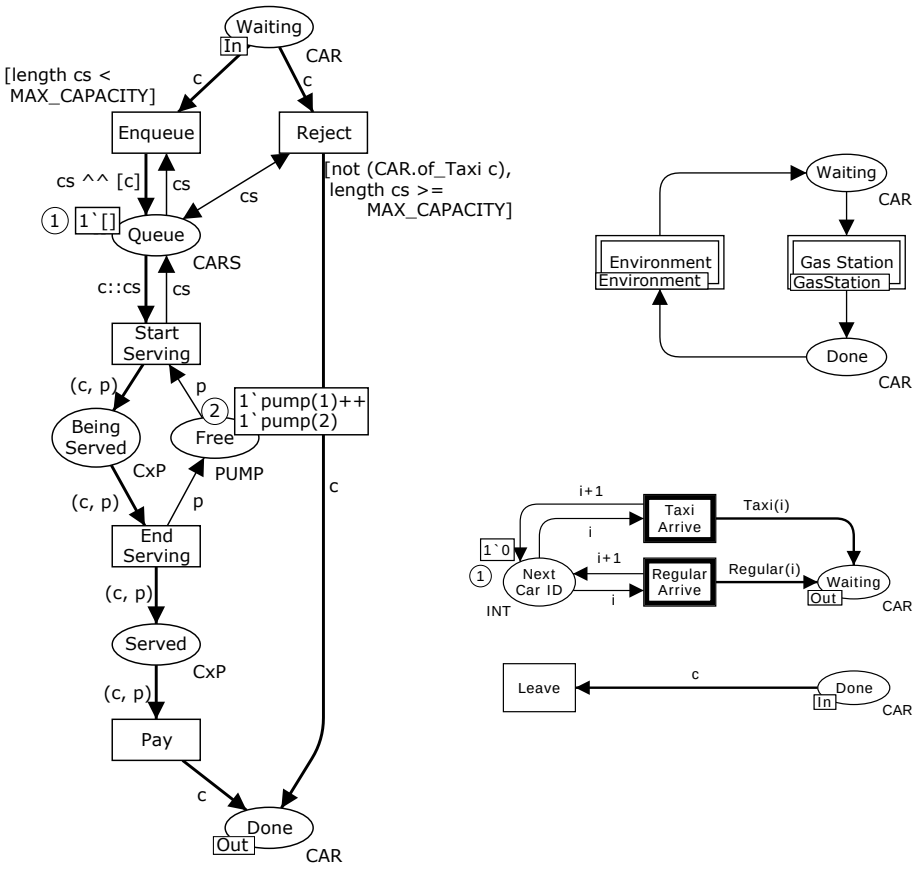

Fig. 18. Untimed gas station.

this section, we look at how CPNs enable us to model timed aspects of systems and use this to extract information about the performance of the system. As shown in the next section, one can do experiments with a timed system using the model, thereby extracting performance data that can be used as input for decisions regarding the modeled system. For example, simulation can be used to find out whether it is better for a gas station to acquire extra pumps or to reserve more capacity for the queue if the number of customers doubles. We also look at the difference between real and integer time stamps and the interaction between timed models and prioritized transitions.

6.1 Time Basics

In CPNs, time is introduced by assuming a *global clock* representing the current *model time*. We assign a time stamp to each token. A time stamp on a token indicates when the token can be consumed. A token can only be consumed if its time stamp is less than or equal to the current model time. In a sense, a token with a time stamp in the future (compared to the current model time) can be regarded as a promise that, at some point in the future, a token will be produced. Hence, one can think of time stamps as reversed expiry dates: tokens with a time stamp x can be consumed at time x or later.

Figure 19(left) shows a timed version of the gas station from Fig. 18. We have merged the two Serve transitions into a single one and added an annotation @+5 to transition Serve. This annotation specifies that executing the transition takes 5 units of time (i.e., firing is atomic, however, the tokens are produced with delay of 5 time units). In the same way, we have added an annotation to Pay that states that paying takes 2 time units. We also see that the current marking of Waiting, Pumps, and Paying reflects time stamps, so that Regular(1) in Waiting is available at time 0 (due to @0 after the token value). Furthermore, we now join tokens with different values using +++ instead of ++. This is merely a technicality due to typing. We see that Taxi(0) has been served by pump(2) (it is on Served) and that Taxi(2) is currently in line to be served.[6] As pump(0) is available at time 0, the current model time is 0, and Serve is enabled at this time. Pay is not enabled at time 0, however, as the token in Served is not available until time 5. In a sense, the tokens (i.e., (Taxi(0),pump(2)) in Served and pump(2) in Pumps) have not been produced yet; we have only a promise that in the future (in 5 time units) the tokens will be produced.

Each pump can serve at most one car every 5 time units; that is, the time-related annotations specify that serving a customer takes time and during that time the pump is not available.

We have to specify for each type whether it is timed or not (we allow tokens without a time stamp, which is a shorthand for a token that is always available). We specify that a type should include a time stamp by adding the keyword timed at the end of the declaration. In our example:

[6] Although there is a token referring to Taxi(0) in place Served, the service has not been completed yet (in fact, it just started). This can be seen by comparing the time stamp of the token in place Served (@5) with the current model time (0).

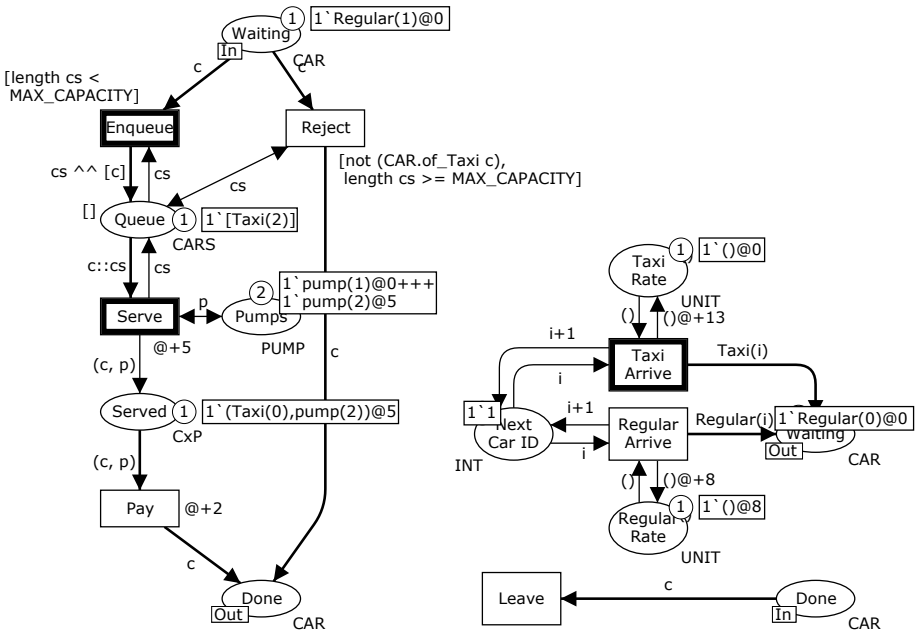

Fig. 19. Simple timed version of the gas station (left) and environment (right).

```
colset CAR = union Regular: INT + Taxi: INT timed;
colset PUMP = index pump with 1..2 timed;
colset CxP = product CAR * PUMP timed;
```

The model of the environment in Fig. 18 does not take time into account and, therefore, produces cars that all appear at time 0. This causes all cars to enqueue at this point. Thus, time will never progress, as time only progresses when there are no more enabled transitions at a given time stamp. We therefore need to make the environment time-aware. We would like that cars arrive at a constant rate, but that the rate is different for taxis and regular cars. We could add separate counters for each of the transitions and use a construction similar to the one used for Serve and Pump, but instead we choose to add separate places with a single token limiting the rate of the transitions. We use a timed UNIT as type and place a single token in each place, obtaining the environment in Fig. 19(right). We have not added time annotations to the transitions, but rather to the output arcs. This enables us to produce tokens that are available at different times. In Fig. 19(right), we have just executed Regular Arrive at time 0, yielding a car Regular(0) on Waiting which is available at time 0, and a token () on Regular Rate which is available at time 8. In this way, we have modeled that arriving takes no time, but the transitions can only occur at a specified rate. The rate for regular cars is once every 8 time units, and the rate for taxis is once

every 13 time units. The token in Next Car ID is untimed and does not influence the enabling of any transitions as it is always available.

6.2 Embedding Time Stamps in Tokens

Place Queue in Fig. 19(left) always contains one token (the queue) and this token is always available as the place is untimed. Hence, the @+5 annotation of transition Serve does not apply to this place. In this particular situation, this is just fine; time progresses only in-between subsequent arrivals and because it takes time to serve a car. Although the untimed queue works well in this situation, there are situations in which an untimed queue cannot express the desired behavior—for example, if the transition Enqueue takes time or to model more complex priority and resource allocation rules, where the next time stamp needs to be computed based on the entire queue. Therefore, we would like to make the tokens of the queue timed as well, but this is not possible, because whereas we think of the queue as a list of tokens, the list is a single token and can therefore have only one time stamp. To fix this, we need to embed the time stamps of the tokens in the queue. We do not have access to the time stamp of a token (it is either available or not), but we do have access to the current model time, and can use that to embed time stamps in the elements of the queue, obtaining the model in Fig. 20. The idea is to not just store cars in the list representing the queue, but a pair of the car and the model time the car arrived; that is, we define the types:

```
colset CxT = product CAR * STRING;
colset CARS = list CxT timed;
var t : STRING;
```

Now, CARS is timed. We want the time stamp of the list token modeling the queue to be the minimum of all time stamps *in* the queue. If the queue contains no cars, we assign the current time to the queue. We need to convert the model time to and from strings, as there is no type in CPN Tools that directly corresponds to the type of the model time. For this purpose we define the following function:

```
fun modelTime() = ModelTime.toString(ModelTime.time())
```

Moreover, we need to compute the time stamp of the queue. Therefore, we define function unwrap_time, which takes as list of cars and returns the same list, but with a time stamp equal to the time stamp of the first car in the list.

```
fun unwrap_time([]) = [] @ (ModelTime.time())
  | unwrap_time((c, t)::rest) =
                ((c, t)::rest) @ (ModelTime.maketime t)
fun append(cs, c) = unwrap_time(cs ^^ [(c, modelTime())])
```

Function append adds a new car to the end of the list together with its arrival time converted to string format. Moreover, using unwrap_time the time stamp of the whole list is computed.

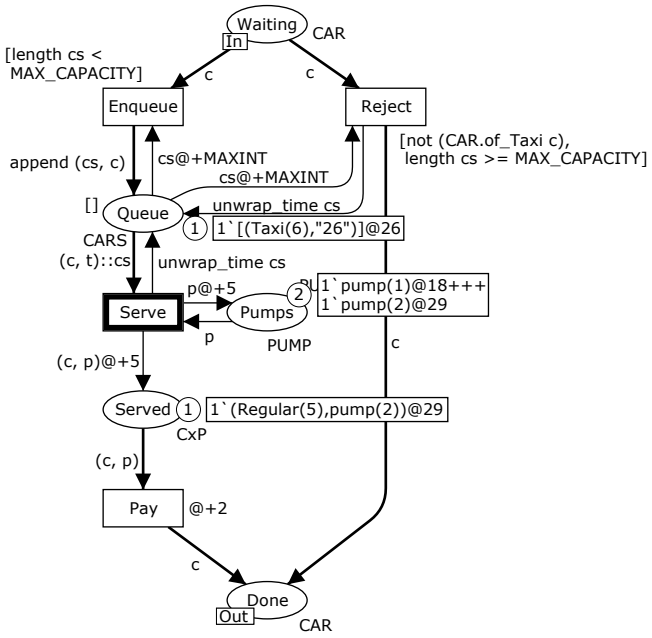

Fig. 20. Timed version which embeds time stamps in the list.

6.3 Ignoring Time Stamps of Tokens

After adding a time stamp to Queue, we need to make some minor changes to other transitions surrounding it so they are not wrongly delayed. First, we have changed Serve to no longer have a time inscription, but to instead only delay the pump and car moved to Served, as unwrap_time takes care of delaying the queue the correct amount. If we kept the time inscription, the time stamp of the queue might also be delayed 5 time units, resulting in a car not being served immediately even though a pump is available. Second, we have added time inscriptions to the arcs from Queue to Enqueue and Reject. The inscription uses MAXINT, which is a constant, declared as:

```
val MAXINT = Option.valOf (Int.maxInt)
```

An inscription on an input arc means that the token can be consumed the indicated time units before it is really available, so consuming it MAXINT time units before basically says to ignore the time stamp. This allows us to reject or enqueue a car even though the queue is not currently available. Although transition Serve needs to wait until the first car is available, the queue can be updated at any time. In this situation, this is not required, but the example shows that we can have full control over time even when objects are put in a list.

6.4 Real Time and Random Distributions

Until now we have used constant delays and rates on all transitions. This is not realistic. Often we would rather know that something has an average delay or rate and perhaps even a guess (or assumption) about how the values are distributed. We thus wish to replace the delays and rates by values resulting from drawing a random number using a given distribution. However, most random distributions occurring in practice, especially randomly distributed times, are not integers. In previous versions of CPN Tools, we would need to scale the random values to a desired precision and convert the randomly drawn values to integers. But from version 3.0 and onwards, CPN Tools supports using real time stamps. Changing time stamps to be real values, we can create a new environment as in Fig. 21. We have changed only the inscriptions on arcs indicating the rates. Rather than using a constant value, we draw values from the exponential function, which randomly samples values from a negative exponential distribution, which is appropriate for modeling arrival rates. The exponential function is given a parameter λ, and it draws values with a mean value of λ^{-1}, which is why we write 1.0/13.0 rather than 13.0. We use inscription @++ rather than the inscription @+. This is needed when we want to make increments that are not integers. Figure 21 shows that the tokens now have time stamps that are not integers.

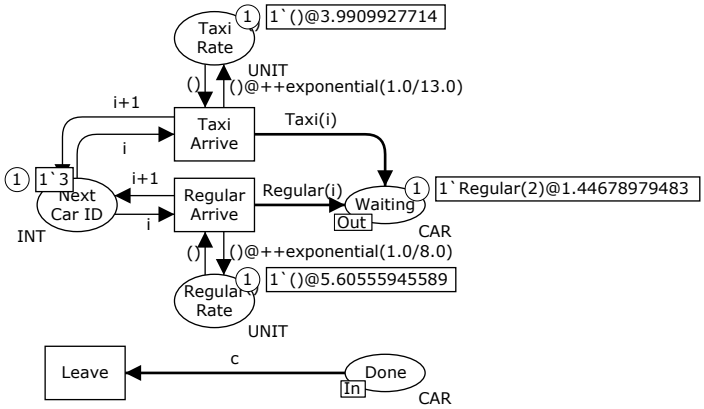

Fig. 21. Real timed environment drawing rates from the negative exponential distribution.

We would naturally like to make an implementation of the gas station also using randomly generated delays. Figure 22 shows an example. We have changed the time inscription of Pay to randomly draw a value from the normal distribution with a mean value of 2 and a variance of 1.[7] Transition Serve has become

[7] Note that the normal distribution is not a very suitable delay distribution as it may generate negative values that are effectively treated as zero's, thus shifting the mean.

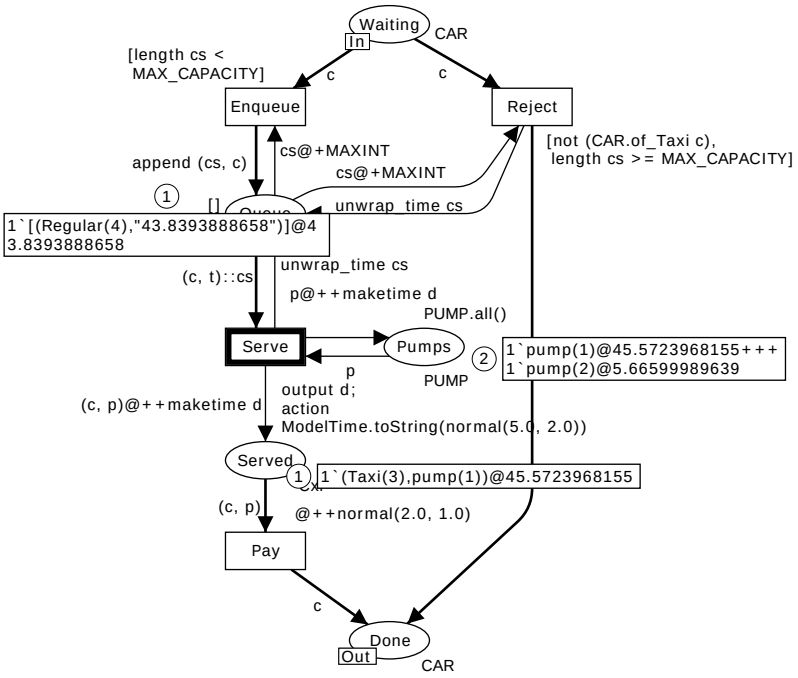

Fig. 22. Real timed gas station drawing average times from a normal distribution.

a bit more complex, as we want the car and pump to complete at the same time, so we cannot just add two calls to normal, as that would draw a random value for each. Instead, we have added a *code segment* to Serve, which outputs a variable d and draws a value from a normal distribution with mean 5 and variance 2, converting it to a string (as variables cannot have type time). The time inscriptions use variable d and convert it into a time stamp.

6.5 Time and Priorities

The interplay of priorities on transitions as explained in Sect. 5.1 and time as explained earlier in this section is nontrivial, as they are both global properties. The basic rule is that *time takes precedence over priorities*, so if a low-priority transition is enabled at time 12, but a high-priority transition is enabled at time 15, the low-priority transition will be executed first. The intuition of the interplay of transition priorities and time is that simulation progresses along a time-line. At each point in time, we evaluate which transitions are enabled and execute the ones with the highest priority first. We then execute all transitions enabled at the current model time in a highest priority first order until no more transitions are enabled, at which point we increase the current model time and start anew.

In the gas station example, if we combine the model in Fig. 15 with the one in Fig. 19(left) (i.e., we add time stamps to the model in Fig. 15), we obtain the model in Fig. 23. Here, Served is enabled even though there are enough tokens for Enqueue to be enabled. The reason is that Regular(4) arrived at time 16, and the current model time is 16, at which point Serve is enabled (if tokens were enabled at the time, it would be enabled already at time 0 for pump(1) and at time 13 for pump(2)). Enqueue is not enabled until time 26, when Taxi(5) becomes available. So, even though Enqueue has high priority and all available tokens, it is preempted by Serve, which is enabled earlier.

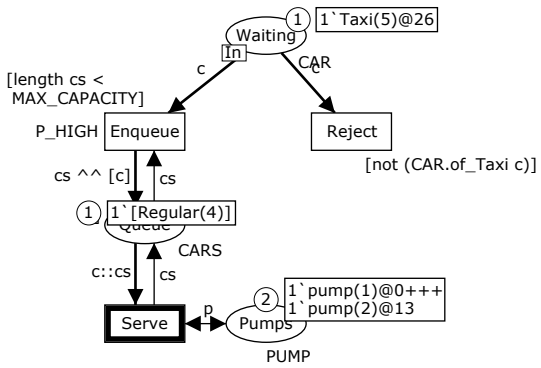

Fig. 23. Gas station with priorities and time stamps. The current time is 16.

6.6 Different Timing Concepts

Many different timing concepts have been introduced in the literature. Authors associate time to transitions, places, or arcs. In most timed Petri net models, transitions determine time delays. In only a few models, time delays are associated to places or arcs. Independent of the choice where to put the delay (i.e., transitions, places, or arcs), several types of delays can be distinguished, for example, deterministic (the delay is fixed), nondeterministic (the delay is a non-deterministic choice over an interval [1,23]), and stochastic (the delay is sampled from some probability distribution [22]). Even when the location of the delays and the type of delay are determined, there are still many possibilities. Adding time to Petri nets requires a redefinition of the enabling and firing rules. For example, when time is associated to transitions and delays are stochastic, one can use preselection semantics (i.e., first the transition to be fired is selected and only then the delay is determined) or race semantics (i.e., transitions are competing for tokens and the transition that finishes first takes the tokens). In the later case one needs to select a memory policy (age memory, enabling memory, or reset memory). This illustrates there are many ways to add time to Petri nets.

Generalized Stochastic Petri Nets (GSPNs) [9] are probably the most widely used Petri net model focusing on the time extension. This model allows for two types of transitions: transitions that do not take time (immediate transitions) and transitions that have an enabling time that is sampled from a negative exponential distribution. Due to the memoryless property of the exponential distribution and the race semantics, it is possible to convert a GSPN into an embedded Markov chain, thus allowing for all kinds of analysis [22].

Most of the timed Petri net models described in literature have been tailored towards a particular analysis technique. Unfortunately, Petri net modeling languages using stochastic delays tend to impose restrictions to allow for Markovian analysis. Most Petri net modeling languages using fixed times or time bound by intervals also impose restrictions to allow for model checking.

CPNs aim at the modeling of large and complex processes. As shown in earlier sections, this requires tokens to be colored. Places may have types that cannot be enumerated (e.g., lists). In fact, Markovian analysis and model checking are unrealistic for applications that use the patterns described earlier; the state spaces of such models are simply too large to allow for exact analysis. Therefore, we need to resort to simulation. As a result, we do not have to put any restrictions on time and, therefore, we can select the most convenient and intuitive time extension.

Since tokens already have a value, it is most natural to also attach time stamps to tokens [1,15,18]. Since time inscriptions can be put on both input and output arcs and it is possible to inspect the current time (see Fig. 22), the designer has full control over the time in CPN Tools. Any of the timed Petri net models described in literature can be emulated. However, performance analysis is restricted to simulation.

7 Simulation

As indicated in the previous section, we need to resort to simulation to analyze the performance of complex processes. Therefore, we focus on this type of analysis. First, we position simulation in the broader spectrum of analysis techniques. Then, we show how to construct simulation models and how to monitor them. Finally, we explain how to interpret the results and illustrate this by comparing different redesigns for our running example.

7.1 Overview of Analysis Techniques

The idea of simulation is to take an executable model (e.g., a CPN) and let it run several times. Each run can be seen as an experiment and corresponds to a "random walk" in the state space of the model. Even if it is impossible to construct the entire state space, it is possible to do such experiments; just "play the token game" repeatedly. A model may allow for infinitely many scenarios, that is, possible runs. Because only a finite number of scenarios can be executed, only a fraction of the entire state space can be explored. Consequently, simulation

can, unlike verification, be applied to verify only the presence of errors and not their absence.

Verification of CPNs is challenging. Because of the introduction of data and time, the state space tends to be large, if not infinite. The only way to use model checking techniques effectively, is to limit the use of data and time. Typically, one needs to abstract from time and use types with a limited number of possible values. We refer to other papers in this ToPNoC volume that focus on this topic and that introduce techniques such as reachability graphs, coverability graphs, unfoldings, invariants, siphons, and traps.

Simulation is widely used for performance analysis. Performance analysis tries to make predictions about key performance indicators, such as response time and flow time, and to detect possible bottlenecks. The goal is to understand the *as-is* situation and to compare this with possible *to-be* situations.

Lion's share of Petri net research has focused on *model-based analysis*; that is, by analyzing a model, one hopes to be able to make meaningful statements about an as-is or to-be situation. However, such analysis only makes sense if the model reflects reality. Simulation results are irrelevant if the model has little in common with reality. Therefore, we advocate the use of *process mining* techniques in case event logs are available. Process mining [2] aims to discover process models from example behavior captured in different data sources (e.g., databases, transaction logs, and audit trails) and to relate existing models to such behavior. As shown in [27], it is possible to discover CPNs from events logs. Such CPNs model the control-flow of cases, resources, resource allocation, dataflow, routing probabilities, and routing conditions. In the context of workflow management systems, it is even possible to upload the current state of such a system into a CPN model and conduct *short-term simulation* [28]. Unlike classical simulation approaches that focus on steady-state behavior, the goal of short-term simulation is to make predictions about the near future to answers questions such as "How many orders will we have in the pipeline next week?", "What is the expected average response time tomorrow?", and "What will be the average flow time by the end of next week if I temporarily add two workers?". The focus of short-term simulation is on the transient behavior. This allows for a "fast forward button" into the future.

In the remainder of this section, we assume that we are able to create a CPN model that adequately reflects reality. Moreover, we focus on the steady-state behavior of processes. Nevertheless, we encourage the reader to consult [27,28] for more information about the alignment between reality and simulation models.

7.2 Adding Stochastic Behavior to a CPN

In Sect. 6 we showed how to add time to models. In our running example, cars are generated using a Poisson arrival process. This means that the time between two subsequent arrivals is sampled from a negative-exponential probability distribution. In any situation where there is a large population of potential entities that can generate requests (e.g., customers refueling their cars), a Poisson arrival process is most natural. One can show that if these entities are in steady

state and independent, their behavior will always resemble a Poisson arrival process. To model the time it takes to refuel a car, we also need to use a probability distribution. Earlier, we used the normal distribution. CPN supports many probability distributions suitable for modeling time durations. Table 1 shows some examples.

Table 1. Random distribution functions.

Function	Description
uniform(a:real,b:real):real	For b > a, uniform(a,b) samples a value from a uniform distribution with mean (a + b)/2.
exponential(r:real):real	For r > 0, exponential(r) samples a value from an exponential distribution with mean 1/r.
erlang(n:int,r:real):real	For n ≥ 1 and r > 0, erlang(n,r) samples a value from an Erlang distribution; that is, the sum of n independent exponentially distributed values with parameter r. The expected value is n/r.
normal(n:real,v:real):real	For v ≥ 0, normal(n,v) samples a value from a normal distribution with mean n and variance v.

Let us now revisit our running example. Figure 24 shows another variant of the gas station. At the highest level, we now distinguish between cars that were served (place DoneS) and cars that were rejected (place DoneR). We use a Poisson arrival process to generate cars. The mean time between subsequent arrivals of taxis is 12 minutes. Regular cars arrive, on average, every 6 minutes. Hence, on average, $5 + 10 = 15$ cars arrive per hour. In our initial situation we have only one pump; that is, FreePumps contains only one token. We split the service transition into a StartServe and EndServe transition to explicitly show when a pump is busy or free. Transition Pay is still atomic and is an "infinite server"; that is, the transition takes time, but cars do not need to wait for one another.

Figure 24 uses the following types:

```
colset STIME = string;
colset BCAR = product INT * STIME;
colset CAR = union Regular: BCAR + Taxi: BCAR timed;
colset CARS = list CAR;
colset PUMP = index pump with 1..1 timed;
colset CARxPUMP = product CAR * PUMP timed;
```

The types are self-explanatory, except for the addition of STIME. Every car has a unique ID of type INT and a creation time of type STIME. We use function modelTime defined earlier to inspect the global clock and convert the current type to a string such that it can be stored in a token. The creation time of a car has been added to measure flow times; that is, we measure the difference in time between the moment a car leaves module Environment and when it returns.

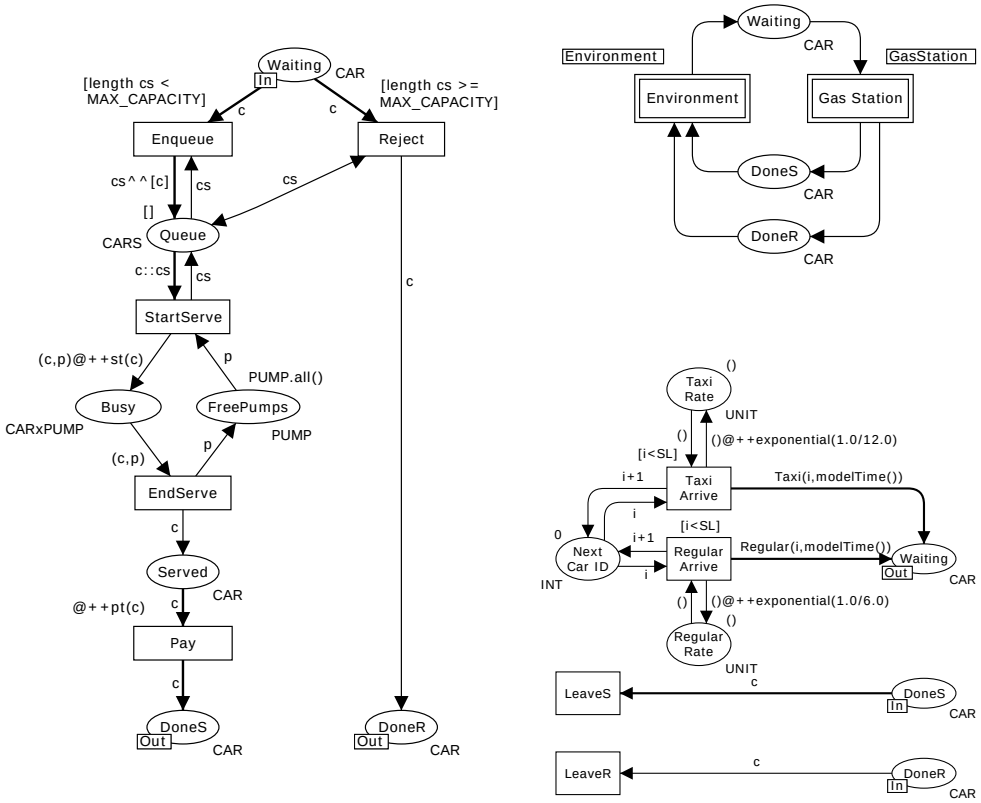

Fig. 24. The initial CPN used for simulation. Taxis and regular cars arrive using a Poisson process. The mean time in-between two taxis is 12 minutes. The mean time in-between two regular cars is 6 minutes. The service time and the time it takes to pay are sampled from a uniform distribution.

Additional declarations used in Fig. 24 are:

```
val MAX_CAPACITY = 3;
fun st(c) = uniform(2.0, 5.0);
fun pt(c) = uniform(1.0, 2.0);
val SL = 100000;
```

In our initial simulation model, we allow for 3 cars to queue. The time it takes to refuel a car is between 2 and 5 minutes (uniform distribution). The time it takes to pay is between 1 and 2 minutes (uniform distribution). For each simulation run, we create 100,000 cars. This is reflected by parameter SL used in module Environment.

7.3 Monitoring a CPN

Figure 24 only models the process we would like to analyze without modeling the measurements required for analysis. However, to extract simulation results, this is not sufficient; we need to indicate what kind of results should be collected (e.g., flow times, utilization, costs). In general, we would like to avoid "polluting" the model with extensions to collect statistics. This is why CPN Tools offers the possibility to add *monitors*. The idea of a monitor is to collect data from markings that are reached and bindings that are enabled during the simulation runs. For example, a *Marking size* monitor counts the average number of tokens in a place. In Fig. 24, we add two such monitors; one for place FreePumps and one for Busy. The average number of tokens in Busy divided by the number of pumps, indicates the average utilization of these pumps. To measure the average number of cars queueing, we add a *List length data collection* monitor for place Queue.

These monitors can be added without any effort. Just select the desired monitor and attach it to the corresponding place.

Measuring the flow time requires a bit more effort. We added the creation time of a car to the type CAR. In addition to an ID, a car also has a value of type STIME indicating the time it was created by the Poisson arrival process in module Environment. Transition LeaveS in module Environment consumes cars that have been served (see Fig. 24). This transition fires the moment the car has been refueled and payment has been completed. Hence, the difference between the time LeaveS fires and the time stored in STIME field of the car token is the flow time of a car. Obviously, we are interested in this duration. Later, we explore various alternatives and analyze their effect on the average flow time of cars that have been served. Using a *Data collection* monitor, we can measure the flow time. This monitor simply stores measurements. In this case, we need to specify that we are interested in the difference between the creation time and current time. Subsequently, these measurements are used by CPN Tools to calculate statistics such as average, variants, upper bounds, and lower bounds.

Cars that have not been served have a flow time of 0. Therefore, we do not need to measure the flow time for such cars. However, we want to measure the fraction of cars that is rejected. Again we use a *Data collection* monitor. However, now we do not measure the flow time, but measure whether a car was rejected (record a value 1) or not (record a value 0). Hence for each car we record 0 or 1. By computing the average over these values, we know the fraction of cars that was rejected in the simulation run.

7.4 Interpreting the results

After extending the simulation model shown in Fig. 24 with monitors, we can execute a simulation run in which 100,000 cars are generated (see parameter SL) and inspect the simulation results. For a particular simulation run, we find that:

- The average length of the list token in Queue is 0.915835;

– The mean number of tokens in Busy is 0.802938;
– The mean number of tokens in FreePumps is 0.197062;
– The average flow time of served cars is 9.001130 minutes; and
– The fraction of cars rejected is 0.089430.

Hence, the utilization is approximately 80% and on average about one car is waiting to be served. The average flow time is approximately 9 minutes and about 9% of the cars is rejected.

Based on one simulation run, we cannot make any conclusions. In another simulation run, the previous results could be different. Therefore, we need to compute *confidence intervals*. By repeating the simulation experiment several times, we can get an idea of the reliability of the results. Suppose that we repeat the experiment 10 times and measure the flow time for each simulation run. If the 10 values are close to one another, then we can be confident that the result is reliable. If the 10 values are far apart, then this is an indication that more or longer simulation runs are needed. Using standard statistical methods, one can compute confidence intervals based on subruns. For an introduction to subruns and confidence intervals in the context of CPNs, we refer to [6]. Here, we just show the results.

If we compute confidence intervals based on 10 simulation runs in which 100,000 cars are generated (i.e., the behavior of 1,000,000 cars is analyzed), then we get the following 90% confidence intervals:

– The average length of the list token in Queue is 0.900 ± 0.006; that is, with 90% confidence the average is between 0.894 and 0.906;
– The mean number of tokens in Busy is 0.798 ± 0.001; that is, with 90% confidence utilization is between 0.797 and 0.799;
– The mean number of tokens in FreePumps is 0.201 ± 0.001; that is, with 90% confidence the mean number of free pumps is between 0.200 and 0.202;
– The average flow time of served cars is 8.945 ± 0.020 minutes; that is, with 90% confidence the average flow time is between 8.925 and 8.965; and
– The fraction of cars rejected is 0.088 ± 0.001; that is, with 90% confidence we can conclude that between 8.7% and 8.9% of cars is not served.

Confidence intervals allow us to compare different alternatives. CPN Tools automatically calculates these intervals when the user uses the command:

```
CPN'Replications.nreplications 10
```

To create more subruns, the parameter of this command can be modified. As discussed in [6], there is a tradeoff between the number of subruns and the length of each run.

7.5 Comparing Alternatives

After creating the CPN shown in Fig. 24 and adding the monitors, it is easy to explore various alternative models and compare them. Table 2 shows the results for the original model (i.e., Fig. 24) and three alternative models.

Table 2. Statistics of the original process and three redesigns (confidence interval of 90%).

Process	average queue length	average number of pumps free	average number of pumps busy	average flow time	fraction of cars rejected
original	0.900 ± 0.006	0.201 ± 0.001	0.798 ± 0.001	8.945 ± 0.020	0.088 ± 0.001
extra places	1.732 ± 0.014	0.154 ± 0.001	0.846 ± 0.001	12.157 ± 0.050	0.032 ± 0.001
2nd pump	0.107 ± 0.001	1.129 ± 0.002	0.871 ± 0.002	5.431 ± 0.003	0.004 ± 0.001
faster pump	0.389 ± 0.002	0.401 ± 0.001	0.599 ± 0.001	5.541 ± 0.009	0.022 ± 0.001

In the original CPN, MAX_CAPACITY was set to 3, indicating that at most three cars can queue. If there are three cars in the queue and a fourth one arrives, it is rejected (transition Reject fires). Table 2 shows what happens if three more places are added; that is, MAX_CAPACITY is set to 6 meaning that up to six cars can queue. More cars are being served, but waiting times and the average queue length get longer. The flow time increases from approx. 9 minutes to 12 minutes, whereas the percentage of rejected cars decreases to approx. 3%. As the confidence intervals are narrow and non-overlapping, it is justified to make such conclusions.

Instead of adding additional space to wait, we also consider adding an extra pump. This can be done by adding another token to place FreePumps. This change is costly, but has a positive effect on all performance indicators. As Table 2 shows, the flow time drops to approx. 5 minutes and less than 0.5% of cars are rejected.

The last row of Table 2 shows what happens if we replace the original pump by a new one that is 30% faster. This alternative is realized by changing the function that models the service time:

```
fun st(c) = uniform(1.4,3.5);
```

The flow time of this alternative is comparable to adding another pump, but more cars are rejected (approx. 2%).

Each of the three redesigns mentioned in Table 2 can be modeled in less than a minute. This illustrates that simulation using CPN Tools allows for a quick exploration of different alternatives.

To conclude this section, we look at three more redesigns that all distinguish between regular cars and taxis. Figure 25 shows the adaptations needed to add an extra pump such that the new pump is only used by taxis whereas the existing pump is used for regular cars. The type PUMP has now two possible values P_taxi and P_regular and initially there is one of each. The guard of StartServe ensures that the pumps are used for serving the right type of cars.

Table 3 shows the simulation results. The performance is not as good as adding an extra pump that can be used by any type of car. The flow time increases from 5.431 ± 0.003 to 6.800 ± 0.008 and the fraction of rejected cars increases from 0.004 ± 0.001 to 0.033 ± 0.001. There are two reasons for this. First

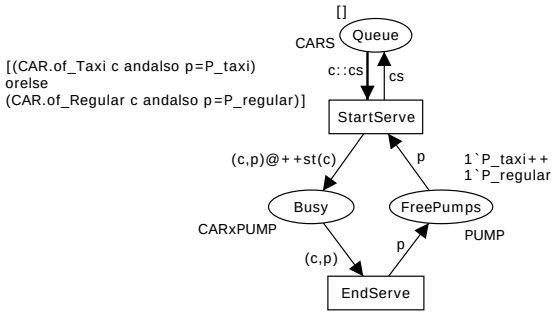

Fig. 25. Changes required to model that taxis have a dedicated pump, i.e., there are two pumps, one for taxis and one for regular cars.

Table 3. Statistics for the redesign with a dedicated pump for taxis and a dedicated pump for regular cars (confidence interval of 90%).

	average queue length	average number of pumps free	average number of pumps busy	average flow time	fraction of cars rejected
all cars	0.435 ± 0.002	1.153 ± 0.001	0.847 ± 0.001	6.800 ± 0.008	0.033 ± 0.001
taxis				6.281 ± 0.010	0.033 ± 0.001
regular cars				7.058 ± 0.008	0.033 ± 0.001

of all, it may be that there are three taxis waiting while the pump for regular cars is free; that is, capacity is unused at times. Second, it may even be that the first car in the row is "blocking" other cars that could be served. For example, consider the scenario with two taxis and two regular cars. If one taxi is being served while the other taxi is first in line, then the two regular cars get blocked even though their pump is idle.

Table 3 shows results for all cars and results for taxis and regular cars separately. Note that the waiting time of taxis is lower, because on average only 5 taxis arrive each hour, whereas on average 10 regular cars arrive each hour.

Next, we consider the situation where taxis have priority; that is, as long as there are taxis waiting, regular cars are not served. This can be realized by simply sorting the queue such that taxis are always in front of the queue. Table 4 shows the results. The overall flow time is comparable to the original situation. However, taxis have, on average, shorter waiting times than regular cars.

The last redesign we consider, reserves the places in the queue for taxis. Taxis can queue as long as there are free slots in the queue. However, regular cars can only enter the queue if no other cars are waiting. Hence, regular cars are always rejected unless the queue is empty. Figure 26 shows how the model can be adapted to realize this redesign. The guards of transitions Enqueue and Reject are modified to enforce the new policy. Table 5 shows the results. Compared to

Table 4. Statistics for the redesign with a single queue where taxis have priority (confidence interval of 90%).

	average queue length	average number of pumps free	average number of pumps busy	average flow time	fraction of cars rejected
all cars	0.900 ± 0.004	0.201 ± 0.001	0.798 ± 0.001	8.944 ± 0.014	0.088 ± 0.001
taxis				6.857 ± 0.007	0.088 ± 0.001
regular cars				9.988 ± 0.021	0.088 ± 0.001

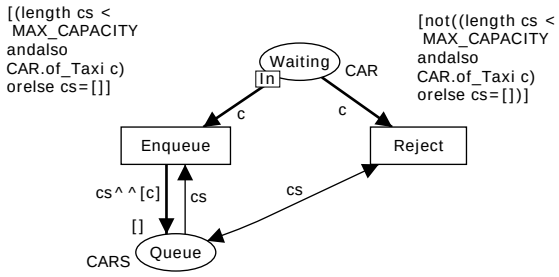

Fig. 26. Only queue if empty

Table 5. Statistics for the redesign in which regular cars can only queue if the queue is empty (confidence interval of 90%).

	average queue length	average number of pumps free	average number of pumps busy	average flow time	fraction of cars rejected
all cars	0.346 ± 0.001	0.300 ± 0.001	0.700 ± 0.001	6.730 ± 0.007	0.200 ± 0.001
taxis				7.475 ± 0.014	0.005 ± 0.001
regular cars				6.204 ± 0.003	0.297 ± 0.001

the original situation, the fraction of rejected cars increased from 0.088 ± 0.001 to 0.200 ± 0.001; that is, approximately twice as many cars are rejected. However, fewer taxis are rejected (only 0.005 ± 0.001) and the flow time reduced decreased for all cars (both taxis and regular cars).

The last three redesigns show that it is easy to use properties of the car when defining policies. This demonstrates the power of CPNs compared to timed Petri nets that do not incorporate data.

8 Conclusion

Colored Petri Nets (CPNs) enhance classical Petri nets with commonly agreed upon extensions such as data, hierarchy, and time. The resulting modeling lan-

guage is highly expressive and is supported by CPN Tools, a powerful software tool for the modeling and analysis of CPNs.

This paper used a running example to explain several design patterns for modeling in terms of CPNs. These patterns guide users in modeling complex processes that require interplay of control-flow and data-flow. We showed examples of simple patterns and more involved ones.

We also introduced the new features of CPN Tools. Version 3.0 supports priorities and one can now use real values as time stamps. Moreover, the new simulator is up to twice as fast and is now also supported on 64 bit platforms. These improvements facilitate simulating complex processes. Using our running example, we showed that it is easy to generate various alternative designs and using simulation to compare them.

We hope that this paper will help students, researchers, system designers and process analysts to make better models in less time. For a more in-depth discussion on modeling in terms of CPNs, we refer to [6,19]. For the software and examples, we refer to CPN Tools Web page `http://cpntools.org`.

References

1. W.M.P. van der Aalst. Interval Timed Coloured Petri Nets and their Analysis. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 453–472. Springer-Verlag, Berlin, 1993.
2. W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer-Verlag, Berlin, 2011.
3. W.M.P. van der Aalst, P. de Crom, R. Goverde, K.M. van Hee, W. Hofman, H. Reijers, and R.A. van der Toorn. ExSpect 6.4: An Executable Specification Tool for Hierarchical Colored Petri Nets. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 455–464. Springer-Verlag, Berlin, 2000.
4. W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
5. W.M.P. van der Aalst, A.J. Mooij, C. Stahl, and K. Wolf. Service Interaction: Patterns, Formalization, and Analysis. In M. Bernardo, L. Padovani, and G. Zavattaro, editors, *Formal Methods for Web Services*, volume 5569 of *Lecture Notes in Computer Science*, pages 42–88. Springer-Verlag, Berlin, 2009.
6. W.M.P. van der Aalst and C. Stahl. *Modeling Business Processes: A Petri Net Oriented Approach*. MIT press, Cambridge, MA, 2011.
7. C. Alexander. *A Pattern Language: Towns, Building and Construction*. Oxford University Press, 1977.
8. J.C.M. Baeten, T. Basten, and M.A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1 edition, December 2009.
9. G. Balbo. Introduction to Generalized Stochastic Petri Nets. In M. Bernardo and J. Hillston, editors, *Formal Methods for Performance Evaluation*, volume 4486 of *Lecture Notes in Computer Science*, pages 83–131. Springer-Verlag, Berlin, 2007.
10. M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley & Sons, 2005.

11. W. Fokkink. *Introduction to Process Algebra*. Springer-Verlag, Berlin, February 2010.
12. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison Wesley, Reading, MA, USA, 1995.
13. H.J. Genrich and K. Lautenbach. System modelling with high level Petri nets. *Theoretical Computer Science*, 13:109–136, 1981.
14. G. Girault and R. Valk, editors. *Petri Nets for Systems Engineering: A Guide to Modeling, Verification, and Applications*. Springer-Verlag, Berlin, 2003.
15. K.M. van Hee. *Information System Engineering: a Formal Approach*. Cambridge University Press, 1994.
16. G. Hohpe and B. Woolf. *Enterprise Integration Patterns*. Addison-Wesley Professional, Reading, MA, 2003.
17. K. Jensen. Coloured Petri Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Advances in Petri Nets 1986 Part I: Petri Nets, central models and their properties*, volume 254 of *Lecture Notes in Computer Science*, pages 248–299. Springer-Verlag, Berlin, 1987.
18. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1996.
19. K. Jensen and L.M. Kristensen. *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer-Verlag, Berlin, 2009.
20. K. Jensen, L.M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems. *International Journal on Software Tools for Technology Transfer*, 9(3-4):213–254, 2007.
21. O. Kummer, F.Wienberg, M. Duvigneau, J. Schumacher, M. Köhler, D. Moldt, H. Rölke, and R. Valk. An Extensible Editor and Simulation Engine for Petri Nets: Renew. In J. Cortadella and W. Reisig, editors, *International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency (ICATPN 2004)*, volume 3099 of *Lecture Notes in Computer Science*, pages 484–493. Springer-Verlag, Berlin, 2004.
22. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. Wiley series in parallel computing. Wiley, New York, 1995.
23. P. Merlin and D.J. Faber. Recoverability of Communication Protocols. *IEEE Transactions on Communication*, 24(9):1036–1043, Sept 1976.
24. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML*. The MIT Press, May 1997.
25. N. Mulyar and W.M.P. van der Aalst. Patterns in Colored Petri Nets. BETA Working Paper Series, WP 139, Eindhoven University of Technology, Eindhoven, 2005.
26. N. Mulyar and W.M.P. van der Aalst. Towards a Pattern Language for Colored Petri Nets. In K. Jensen, editor, *Proceedings of the Sixth Workshop on the Practical Use of Coloured Petri Nets and CPN Tools (CPN 2005)*, volume 576 of *DAIMI*, pages 39–48, Aarhus, Denmark, October 2005. University of Aarhus.
27. A. Rozinat, R.S. Mans, M. Song, and W.M.P. van der Aalst. Discovering Simulation Models. *Information Systems*, 34(3):305–327, 2009.
28. A. Rozinat, M. Wynn, W.M.P. van der Aalst, A.H.M. ter Hofstede, and C. Fidge. Workflow Simulation for Operational Decision Support. *Data and Knowledge Engineering*, 68(9):834–850, 2009.

29. N. Trcka, W.M.P. van der Aalst, and N. Sidorova. Data-Flow Anti-Patterns: Discovering Data-Flow Errors in Workflows. In P. van Eck, J. Gordijn, and R. Wieringa, editors, *Advanced Information Systems Engineering, Proceedings of the 21st International Conference on Advanced Information Systems Engineering (CAiSE'09)*, volume 5565 of *Lecture Notes in Computer Science*, pages 425–439. Springer-Verlag, Berlin, 2009.
30. R. Valk. Object Petri Nets: Using the Nets-within-Nets Paradigm. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 819–848. Springer-Verlag, Berlin, 2004.
31. B. Weber, M. Reichert, and S. Rinderle-Ma. Change Patterns and Change Support Features: Enhancing Flexibility in Process-Aware Information Systems. *Data and Knowledge Engineering*, 66(3):438–466, 2008.
32. Workflow Patterns Home Page. http://www.workflowpatterns.com.

Appendix: CPN Patterns

In this paper, we used and described several CPN patterns without explicitly enumerating them. This way we could focus on the running example and the process of modeling in terms of CPNs. In this appendix, we list the patterns described in [25,26]. This was the first systematic collection of design patterns for CPNs. Note that this collection was based on constructs described in books such as [6,15,18,19] and examples on the CPN Tools website.

1. *ID Matching*: to make identical information objects distinguishable. This pattern is used in most CPN models shown in this paper. For example, individual cars are distinguished using type CAR.
2. *ID Manager*: to ensure uniqueness of identifiers used for distinguishing identical objects. The places labeled Next Car ID in Figs. 1, 7(left), 19(right), 21, and 24(right) are used to realize this pattern, i.e., each car gets a unique ID.
3. *Aggregate Objects*: to allow manipulation of a set of information objects as a single entity. The queue pattern is a specialization of this pattern and used in various versions of the gas station module (see below). Place Receivers in Fig. 16 is another example implementing this pattern.
4. *Queue*: to allow manipulation of the queued objects in a strictly specified order. The places labeled Queue in Figs. 12(left), 18(left), 19(left), 20, and 24(left) are used to realize this pattern, i.e., cars are put into a list to enforce a particular order.
5. *FIFO Queue*: to allow manipulation of objects from the collection in a strictly specified order such that an object which arrived first is consumed first. The places labeled Queue in Figs. 12(left), 18(left), 19(left), 20, and 24(left) are used to realize this pattern, i.e., cars are put into a list and when a pump becomes available the car that queued first is taken.
6. *LIFO Queue*: to allow manipulation of objects from the collection in a strictly specified order, such that the mostly recently added object is retrieved first. Most queues in this paper use a FIFO order. By taking the last element from the list rather than the first, a FIFO queue can be converted into a LIFO queue.

7. *Random Queue*: to allow manipulation of objects from the collection such that objects are added to the queue in any order, and an arbitrary object is consumed from it.
8. *Priority Queue*: to allow manipulation of objects from the collection in the order of the objects' priority. Table 4 shows simulation results for a CPN model using this pattern.
9. *Capacity Bounding*: to prevent over-accumulation of objects in a certain place. This was the main topic of Sect. 4.1. See place Being Served in Fig. 9 (middle) and place Queue in Fig. 9(right). Both places are bounded by a complement place. The pattern is also used in Figs. 10, 11, 12, 13, 14, 17, 18(left), 19(left), 20, 24(left), and 25.
10. *Inhibitor Arc*: to support "zero-testing" of places. Transition Reject in Fig. 10(left) can only fire if place Queue contains two tokens and, hence, all free places are taken. Fig. 10(right) models this more explicitly by adding a place that counts the number of free places. In Fig. 26, regular cars can only queue if no taxis are waiting.
11. *Colored Inhibitor Arc*: to support "non-containment" property of places.
12. *Shared Database*: to enable centralized storage of data shared between multiple transitions, supporting different levels of data visibility (i.e. local, group, or global).
13. *Database Management*: to specify the interface of accessing data, stored in a shared database for read-only and modification purposes.
14. *Copy Manager*: to make data stored in the shared database available at other locations for local use, while maintaining the consistency of data in all places.
15. *Lock Manager*: to synchronize access to shared data by means of exclusive locks. The fuel stations are shared resources that can only be used by one car at a time. In Fig. 25 taxis and regular cars have a dedicated pump. This can be seen as a form of locking.
16. *Bi-Lock Manager*: to synchronize access to shared data for reading and writing purposes by means of shared and exclusive locks.
17. *Log Manager*: to record the information about actual process execution by means of a data log.
18. *Blocking State-Independent Filter*: to prevent data non-conforming to a certain property from passing through.
19. *Blocking State-Dependent Filter*: to prevent data non-conforming to a property involving the state of an external data-structure, from passing through. Transition Reject in many of the variants of the gas station module is only allowing cars to queue if the queue is not full yet. Consider for example the two variants shown in Fig. 10. Taxis that cannot enter the queue will not be rejected; they are blocked until the queue is not fully occupied anymore. This can be seen as a variant of this pattern.
20. *Non-Blocking State-Independent Filter*: to filter-out data fulfilling a certain property while avoiding accumulation of non-conforming data in the filter input place.

21. *Non-Blocking State-Dependent Filter*: to filter-out data non-conforming to a property, involving the state of an external data-structure, while avoiding accumulation of non-conforming data in the filter input. Consider for example the two variants shown in Fig. 10. Regular cars that cannot enter the queue are immediately rejected; as long as the queue is fully occupied, regular cars are filtered out. This can be seen as a variant of this pattern. Another example is Fig. 26 where regular cars can only queue if no taxis are waiting.
22. *Translator*: to enable coordinated communication between two actors with originally different data formats.
23. *Asynchronous Transfer*: to allow transportation of data from one location to another, while avoiding the sender to block. The environment module in Fig. 6 does not block while sending cars to the gas station module.
24. *Synchronous Transfer*: to allow transportation of data from one location to another, ensuring that an actor, which posted a request, is blocked until it does not receive the requested information.
25. *Rendezvous*: allow multiple actors to broadcast and discover data objects concurrently.
26. *Asynchronous Router*: to enable asynchronous transfer of data from a single source to a dedicated target, providing loose coupling between the source and targets connected to it.
27. *Asynchronous Aggregator*: to provide a holistic view of data, produced by multiple unrelated sources through asynchronous data aggregation.
28. *Broadcasting*: to allow broadcasting of data from a single source to multiple targets, while avoiding direct dependencies between them. Section 5.2 shows how to realize a broadcast (see for instance Fig. 16).
29. *Redundancy Manager*: to prevent the transfer of duplicated data between loosely-coupled actors who communicate asynchronously.
30. *Data Distributor*: to support parallel data processing by distributing data between several independent actors.
31. *Data Merger*: to compose a single information object out of several smaller ones when all parts required for composition become available.
32. *Deterministic XOR-Split*: to allow at most one transition out of several possible to execute, based on fulfillment of mutually excluding data conditions. Transitions Enqueue and Reject in Fig. 24(left) form a deterministic XOR-split.
33. *Non-Deterministic XOR-Split*: to allow any transition out of several possible, but satisfying the same data condition, to execute. Transitions Serve and Reject in Fig. 1 form a (partially) non-deterministic XOR-split; regular cars can be served or rejected in a non-deterministic manner.
34. *OR*: to allow any number of tasks to be selected for execution based on the fulfillment of a certain data condition.

Some of the patterns just mentioned (e.g., the *Queue* and *Capacity-bounding* patterns) are used frequently in this paper, others are not used at all. The 34 patterns described in [25,26] focus on data management and communication.

This explains why several patterns are not used in our running example. Some of the patterns presented in this paper are closer to the workflow patterns mentioned in Sect. 1 [4,32] rather than the original CPN patterns. For example, the *Region Flush* pattern discussed in Sect. 5.3 is closely related to the *Cancel Region* pattern in [4,32]. Moreover, compared to the original CPN patterns in [25,26], we put more emphasis on time (see Sect. 6 and Sect. 7).