Introducing Design Patterns for Petri Nets

Jörn W. Janneck, Martin Naedele *

Abstract

Petri nets are an established and well researched means for systems modeling and simulation, but its use in the engineering community is not as widespread as the applicability of the formalism would suggest. A reason for this might lie in the fact that there is no established concept for the concise presentation of reusable Petri net design knowledge. This paper proposes Petri net design patterns as a style of presentation of such design knowledge ranging from building blocks to architectural considerations. The template for description is introduced using a number of examples taken from our design experience.

---

*Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology Zurich, CH-8092 Zürich, Switzerland, email: {janneck,naedele}@tik.ee.ethz.ch

Contents

1 Introduction

Petri nets are an established and well researched model of computation that is applicable in various domains, such as concurrent system design, economics, work flow modeling, and theoretical computer science. However, users are mostly found in academic and research environments, whereas the engineering community seems to be reluctant to adopt Petri nets for systems modeling and simulation. A reason for this may be found in the fact that while there exists a fair number of introductory literature on Petri nets as well as a large body of advanced theoretical papers, there are to our knowledge no references available that would teach a practising engineer how to apply the basic concepts of Petri nets to the modeling of systems of greater complexity.

From our experience in introducing Petri nets to design engineers we conclude that it is easy to grasp the basic concepts of places, transitions, and concurrent sequences, and that one is very quickly able to apply this to the modeling of systems that have an internal structure that only consists of applications of those basic concepts. The situation changes when one tries to model complex technical systems. It appears to us that there is a need for a structured collection of easily accessible design experience that can be used as reference for "best practices" by the design engineer who is not so much interested in learning about the theoretical implications of Petri net semantics but rather wants to use the formalism to specify, simulate, and evaluate systems.

The main objective of this paper is to give an initial impulse to collect and spread Petri net modeling knowledge in the form of building blocks and pattern descriptions in order to make Petri nets more popular for the use in real world applications. The representation of design knowledge and experience in the form of patterns is increasingly being used in the software engineering domain (see section 2), and we believe that it can successfully be transfered to the area of Petri net modeling as well. We intend to provide a starting point for discussion by introducing the concept of Petri net patterns with a small number of patterns taken from our practical experience. The principal issue of this paper, though, is to demonstrate a specific way of illustrating patterns, not the presentation of the patterns we selected as examples. Although we are not aware that any of the patterns in this paper have been described elsewhere, we do not claim that they are original or overly sophisticated. We do, however, know from our experience that they are sufficiently intricate to be considered valuable information in addition to descriptions of the basic Petri net elements.

The paper is organized as follows: Section 2 contains a short reference on the Petri net language we use, including an explanation of certain non-standard features and the hierarchy concept that is referred to in the pattern description later on. In chapter 3 we give an introduction to design patterns and suggest a notational template. The main part of the paper, section 4, presents a detailed description of five Petri net design patterns. We close with a discussion of ideas we have on further work to be done in the area of Petri net design patterns in order to make the Petri nets formalism more useful for engineers.

2   Basic modeling concepts

In this section we will informally introduce a notation for Petri nets that will serve as a basis for illustrating the design patterns in the following section. It is based on colored Petri nets [18] and an object-oriented component model [12].

2.1   Components and their composition

The absence of any concept of compositionality and abstraction in the pure Petri net formalism has long been observed (see for instance [17]) and given rise to several approaches to overcome this deficiency. A common way to add hierarchical composition to Petri nets is the *refinement* of either or both of its standard atomic building blocks, place and transition [18], [19], [24]. A typical problem with this approach is that these refinements destroy some of the key properties of the atomic places and transitions they are supposed to refine [6].

In this paper we will therefore use a different approach. It is based on a notion of colored Petri nets [18], i.e. each token is assigned a 'color' from a given set of possible colors. Since our model is object-oriented, these color sets are classes, and thus the colors are instances of these classes. Each place is assigned such a class which denotes the set of objects which are permitted to be put on the place. In these conventions we follow the ones made for Object Petri Nets [19], [20].

Instead of trying to refine either places or transitions, we will consider components to be subnets with inputs and outputs, which can be embedded into other components. For example, Figure 1 shows the definition of a component of type (class) `Adder` that has two inputs and one output depicted as triangles (we will denote the direction of the token flow by the direction of the triangle towards the component edge). We will require inputs to be connected to places and outputs to transitions to ensure that any component output can be meaningfully connected to any component input (see the corresponding rules below). Also, inputs and outputs (collectively referred to as *interfaces*) are named and typed: each **Adder** component has two inputs named `in1` and `in2`, respectively, and one output `out`. All interfaces allow tokens of type `Number`, which we assume to be the class of all numbers. We also assume the operator `+` to be meaningfully defined for these entities.
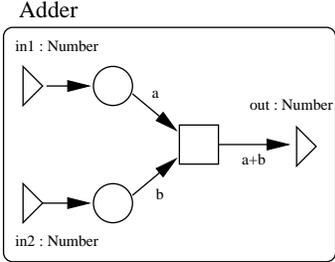

Figure 1: A component.

In Figure 2 we see the use of an `Adder` instance in the definition of another component, `Nats`, generating the sequence of natural numbers. It has an input

4

(`next`) thru which the next number may be requested and an output (`nats`), at which the natural numbers are delivered. The embedding rules allow regular Petri net places and transitions to be connected to the interfaces of embedded components and these interfaces to be connected to each other or to the interfaces of the embedding component in one of the following ways:[1]

1. Transitions can be connected to embedded component inputs.

2. Input interfaces can be connected to embedded component inputs.

3. Embedded component outputs can be connected to embedded component inputs.

4. Embedded component outputs can be connected to places.

5. Embedded component outputs can be connected to output interfaces.

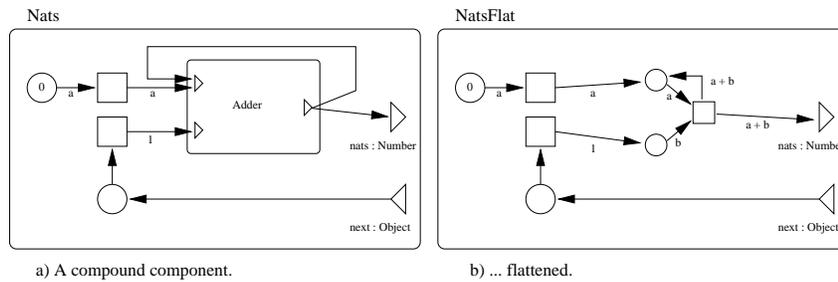a) A compound component.   b) ... flattened.

Figure 2: Embedding a component.

Such a net containing embedded components can be transformed into a *flat* colored Petri net by substituting an embedded component by its definition. Connections between places and transitions that cross component boundaries (via interfaces) are substituted by the corresponding direct connection. The result of flattening the `Nats` component in Figure 2a can be seen in Figure 2b. In this way we can recursively flatten any hierarchical system to arrive at a non-hierarchical colored Petri net.

2.2 Inheritance

In order to facilitate reuse of components we will take definitions as those in Figure 1 as describing a component *class* instances of which may be used (*instantiated*) in several places and which may be subclassed in the manner closely modeled after the way this is typically done in object-oriented languages.[2]

In Figure 3 we see a component class `RealFunc` that does not specify anything except for (the names and types of) its interfaces, which are then inherited by its subclass `Doubler`. Requiring subclasses to provide at least the interfaces of their superclasses ensures that they conform to the *principle of substitutability* [27] – which is a purely syntactical property and does not say anything about the behavior of the respective components.

---

[1]In fact, this form of embedding is a special case of a more general form, as will be outlined below.

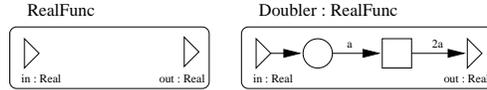[2]Other object-oriented extensions to Petri nets can be found, e.g., in [19], [7], [12].

Figure 3: Simple inheritance.

2.3 Parametric classes

In order to foster reuse of component classes and at the same time facilitate static type checking and polymorphism, we allow the definition of *parametric classes*. When creating instances of these classes, one has to specify values for variables used in their definition. We distinguish between two differentkinds of parameters depending on the places they can be used in the definition of a component: *generic parameters* and *instantiation parameters*. While the former may be used anywhere in the definition of a component class, the latter are only allowed in places that do not affect its outward appearance (e.g. they could *not* be used as the types of the interfaces). We will distinguish them syntactically by writing generic parameters between angular brackets and instantiation parameters between parantheses. See for instance Figure 4 for a component class with both, a generic and an instantiation parameter. Of course, in general a component need not have either.
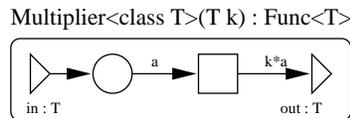

Figure 4: Generic and instantiation parameters.

Note that the type of the instantiation parameter `k` is the value of the generic parameter `T`. As a consequence of the rule above, if two components differ only in the instantiation parameters, they have identical outward interfaces and can therefore be taken as of the same type. In contrast, each different generic parameter setting defines a new type. For example, if we create instances `Multiplier<Integer>(2)` and `Multiplier<Integer>(3)`, they would both belong to class `Multiplier<Integer>` (thus being entirely substitutable), in contrast to `Multiplier<Real>(3.14)`, which would be an instance of `Multiplier<Real>`, which is a different type.[3]

2.4 Instantiation expressions

Now we can reconsider the questions of instantiating an embedded component in the definition of a new component class. Our parametric `Multiplier` component class from Figure 4 could be used inside a component generating all even numbers starting from 2 as shown in Figure 5. We instantiate a `Multiplier<Integer>` component by `Multiplier<Integer>(2)`, which is an

---

[3]We will not address here the problems arising from the question how some `A<X>` relates to some `A<Y>` if we know `X` to be substitutable (i.e. a subtype of) `Y`

6

instantiation expression. Such an expression could be any expression that evaluates to a component, i.e. which has a type (`Multiplier<Integer>`, in this case) that is a component class. Since it is evaluated at the time when the component is instantiated (just as the expressions describing the initial token setting in the places), it can only depend on the parameters of the component class.
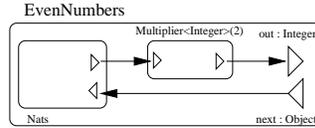

Figure 5: Even numbers generator instantiating `Multiplier<Integer>(2)`.

3 Patterns

3.1 History of patterns and previous work

Attempts to capture the building blocks and architectural considerations of a design as so-called "patterns" have their origin in the work of the architect Christopher Alexander [1], and have recently, become more and more popular in the software engineering domain, especially in the context of object-oriented techniques (e. g. [14], [9], [11], [26]).

A pattern in this sense is the description of a recurring problem or problem type and a generalized solution for this problem. Gamma et al. [14] state that a pattern consists of four essential elements: a name, a problem, a solution, and a description of consequences. A suitable and concise pattern name transforms the complex array of elements that make up the solution to an entity in its own right, that can be manipulated and, more importantly, be communicated and discussed as a whole. The problem part of a pattern describes the situation that calls for the application of this particular pattern as well as conditions that need to be met to allow its use. The solution section presents a collection of elements and relationships that are necessary to solve the problem. The designer should regard the "solution" as a starting point and template which he will in most cases need to modify and extend to suit the context of his particular problem set. Finally a pattern may contain a section discussing the consequences of the application of this pattern, the trade-offs and possible alternatives, to allow an informed decision between patterns that solve similar problems. This section will become more and more important after the pattern catalog for a particular design domain has reached a certain size.

Alexander introduced not only patterns, he also suggested to arrange the patterns of a domain in the form of a so-called "pattern language" to systematically guide the designer from the system level specification to a detailed design of a certain quality. This paper however does not yet suggest or discuss a possible Petri net modeling pattern language.

The idea of using patterns, though not known under this name, is in principle not new to the Petri net community. There exists a rather small body of recurring examples to illustrate certain behaviors of a net that can very well be

7

called patterns. However, the problem with those particular examples like dead-lock (e. g. [2, 21]), dining philosophers (e. g. [21, 15, 5]), producer/consumer and reader/writer (e. g. [2, 21, 15, 22, 23]) etc. is that, while they do illustrate their particular theoretical point, the style of presentation used is not intended to show how to integrate those examples into models of real systems. The idea of general purpose building blocks is hinted at in [10], but the authors of this case study do not put strong emphasis on the reuse aspect. The presentation of building blocks for communication protocols in [4] and [3] is also heading in the same direction as what we suggest in this paper, but the focus there is more on the demontration of a hierarchy and component concept than on the concise description of fundamental design principles.

3.2 A template for the description of Petri net patterns

One of the most important issues when discussing patterns as design aid is that the pattern description should be detailed, so that there is enough information to apply the pattern, and at the same time it should be possible to quickly survey of a number of patterns. Last but not least, pattern descriptions should be uniform to a certain extend to facilitate comparisons between patterns.

We do not claim to know the "correct" way to describe a pattern. Some people may prefer actual application domain examples to explain a pattern, others might like a condensed description of the architectural skeleton. The most suitable form may also depend on the particular pattern itself, its problem context or level of granularity. As a starting point for discussion we suggest the following description template, which follows the four principal elements of a pattern that were mentioned in the previous section. It is, with some modifications, taken from [14]. Certain descriptional sections may not apply to a specific pattern.

Name block:

> **Name:** A name to identify the pattern and distinguish it from others. The name should be such that it clearly conveys the main idea of the pattern.

Problem block:

> **Problem:** The problem and problem context which the pattern addresses.

> **Example:** A concrete example of the problem within some application domain.

> **Required net formalism** : Most of the patterns in this paper rely on the availability of certain extensions of the basic Petri net formalism, such as inhibitor arcs, or CPN-like values on tokens.

Solution block

> **Solution:** The basic idea of the pattern (if non-trivial)

> **Sample structure:** A graphical representation of a Petri net that im-plements/uses the pattern. The sample structure may be a neutral skeleton or refer to the example given before. In the latter case the description needs to make clear which parts are fixed elements of the solution and which belong to the example only.

Description: A detailed description of the function of the Petri net building block, also discussing design considerations, variations and options. As far as possible, the explanation should make use of other patterns contained within the pattern under consideration.

Consequences block:

    Uses: References to other patterns that are contained within the described pattern.

    Similar to: References to and comparisons with other patterns that are similar in some aspect like net structure or targeted problem.

    Further sections may be used to highlight special aspects or trade-offs of using a certain pattern.

4 Pattern Descriptions

4.1 A-before-B

Name: A-before-B

Problem: On synchronizing two concurrent subnets it may be interesting to know the order in which the enabling tokens arrive at the synchronization point.

Example: In a manufacturing cell some product is assembled from two components. One component is previously tooled within the same cell and the other is added from another cell with the aid of a robot manipulator. Ideally, the manipulator would deliver component 2 the very moment component 1 gets ready for the assembly, otherwise one component will have to wait for the other. In some realistic situation the manipulator would not wait for component 1 at the end position, but will insert an additional stop on the way. The total delay then does not only depend on the pure time to wait for the slower component to arrive, but also on the the additional time to accelerate the manipulator every time the manipulator had to wait.

Required net formalism: Petri nets with a concept of time (so that there exists a notion of "before") and inhibitors (or suitable replacement constructions)

Solution: The pattern is asymmetrical and consists of two stages: First, it is detected whether token B is already in place when A arrives, and second, there is an additional synchronization so that an early token A has to wait for the later arrival of B (for the other case the arrival of A effects the synchronization).

Sample structure: see Figure 6

Description: Token A arrives first on P1. Only T1 can fire as T3 would need a token on P2 to be enabled. T1 fires and places one Token on P3. Now a token arrives on P2 and T2 fires (A before B). Otherwise, token B arrives first on P2. Transition T1 cannot be fired upon arrival of Token A because
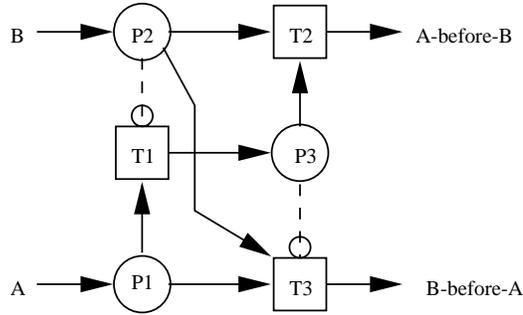
Figure 6: A-before-B

of the inhibitor arc from `P2` to `T1`, therefore `T3` fires (B before A). The additional inhibitor arc between `P3` and `T3` is necessary to prohibit the firing of `T3` before `T2` in case of an arrival sequence A-B-A.

For high-level Petri nets with distiguishable tokens this pattern may be extended to deliver always the first (or the second) token at the output.

4.2 Token multiplier

Name: Token multiplier

Problem: With the firing of a certain transition it is desired to produce several tokens on the outgoing arc. The number of tokens to be produced is determined by an input token value at run time, so constant arc weights cannot be used to achieve this.

Example: For the model of a piece of semiconductor manufacturing machinery it is necessary to generate the tokens representing the dies from one token symbolizing the wafer. For this model there can be a variety of wafer tokens that differ in the value of the wafer diameter attribute. From the wafer diameter a function in the transition calculates the corresponding number of dies, this number is therefore only known at run time.

Required net formalism: High-level Petri nets (necessary are value tokens and guard conditions)

Solution: The basic idea is to replace the single transition with a loop where a token circulates a varying number of times. This token is the loop counter and its value is decremented in each iteration.

Sample structure: see Figure 7

Description: The pattern requires three items of input information: the *number* and *type* of tokens to be produced, and a *starting signal*. In the example the starting signal is combined with the token that carries the wafer size from which the initial value of the loop counter, the number of dies on the wafer, are calculated. The type of tokens to be produced at the output is always "DieType". The firing of `Process` places this "type template" token in `Data` for use with each firing of `Continue`. Depending on
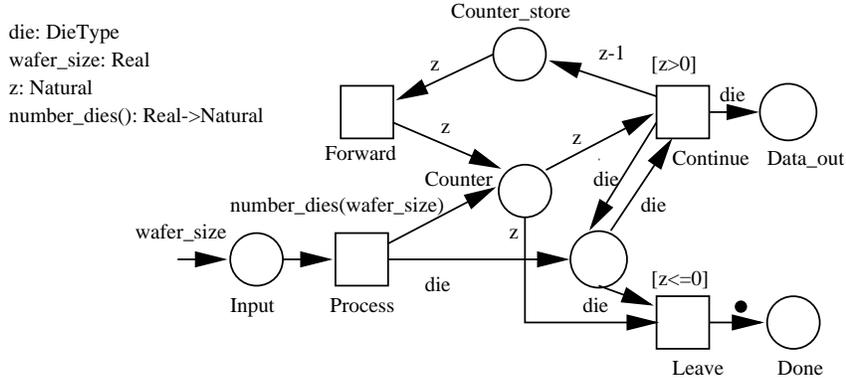
10

die: DieType
wafer_size: Real
z: Natural
number_dies(): Real->Natural

Counter_store

z   z-1   [z>0]

Forward   die   Continue   Data_out

z   z

Counter   die

number_dies(wafer_size)   die

wafer_size   z

Input   Process   die   [z<=0]

die

Leave   Done

Figure 7: Token multiplier

the counter value the guard condition of either the `Leave` or the `Continue` transition is fulfilled. In case of `Continue` the counter value is decremented and the counter token is circulated for another round. In case of `Leave`, `Data` is cleared and a token is passed into `Done` to signal termination of the process.

Variations to the pattern may deal with different ways to set the type of the produced tokens, to remember/store this token type information between the transition firings, or include additional processing steps for each iteration.

Similar to: This pattern is similar to the Petri net extension of "marking controlled arc weights" suggested in [5],[25] and [13]. Whereas these refer to a notational extension of the basic Petri net formalism that determines the weight of an arc by the number of tokens in a certain place, we discuss here a more flexible, architectural approach to the solution of the underlying problem.

4.3   Complete removal

Name: Complete removal

Problem: All tokens stored in a particular place are to be removed from that place while the number of tokens is not known a priori.

Example: see pattern "quasi-continuous movement" later in this paper

Required net formalism: C/E nets with inhibitor arcs [8]. If the intention is to remove all tokens en bloc from the place before the tokens that were removed first cause any other transition firings in the net then it is either necessary to utilize some transition priority concept provided by the formalism or the pattern needs to be embedded in an environment like the one shown in Figure 8 b). All inhibitors except for the on between `Place_to_be_emptied` and `End_removal` can be substituted with auxiliary constructions.

Solution: A transition is fired repeatedly and each time removes one token from the place to be emptied. This continues until the place is empty.
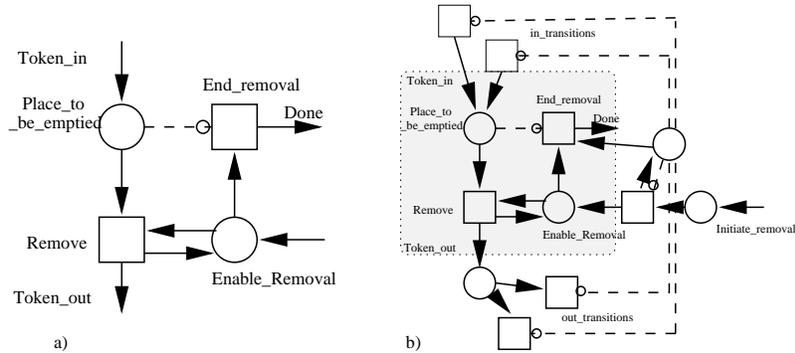
Sample structure: see Figure 8

Figure 8: Complete removal a) basic pattern b) without priority support in the formalism

Description: After a token is placed into `Enable_removal` the `Remove` transition can fire. Because the `Enable` token is returned to the `Enable removal` place after each firing, the transition can fire repeatedly, until there are no more tokens in `Place to be emptied`. Then the inhibitor arc enables the `End removal` transition which removes the token from the `Enable removal` place.

Similar to: This pattern is similar to the concept of "reset arcs" ("Abräum-kanten") [5] or "erase arcs" [8]. The "complete removal" pattern can be regarded as a replacement of reset arcs valid for Petri nets extended only with inhibitor arcs. "Erase arcs" are introduced mainly to realize some sort of reset functionality for parts of a system, consequently erasing is regarded as an atomic operation, and the tokens taken from the "erase place" are destroyed. The "complete removal" pattern on the other hand is more flexible as it allows to embed additional actions into the removal process and the tokens removed from the place are not necessarily lost.

4.4 Quasi-continuous movement

Name: Quasi-continuous movement

Problem: A token progressing through several processing stages is usually modeled using a simple sequence of places and transitions. This is not possible if either the number of discrete positions varies at run-time, the processing stages are partly overlapping, the speed of progress is variable or a quasi-continuous movement from one processing station to the next has to be modeled. More generally, this pattern may be used to model all sorts of problems where numerical attributes of a collection of tokens

are changing synchronously, transitions are to be fired whenever those attributes reach certain values, and where the amount of change per step or the limit values to fire the transitions are only known at run-time.

Example: We consider the model of an assembly line (see Figure 9) with a conveyor belt of fixed physical length and one or more processing stations on which a (continuous) stream of parts to be processed is flowing, one immediately after the other. The parts moving on this assembly line (represented as tokens) are assumed to be of variable length so that the position of a certain part can not be given in terms of it being the n-th part counting from the start of the line, but only in terms of the numerical position of e. g. the front edge of the part. Each processing station then has a certain physical range on the belt within which it can operate on parts.
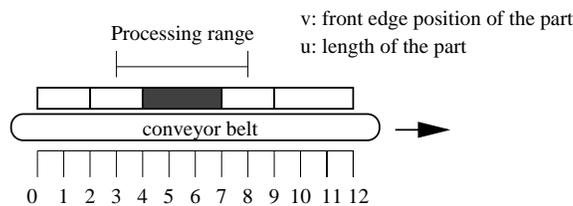

Figure 9: Conveyor belt with transported parts of varying size

Required net formalism: Coloured Petri net (numerical token attributes), inhibitor arcs

Solution: The "quasi-continuous movement" pattern does not use a sequence of places and transitions to model the position of the parts on the assembly line but circulates the tokens in a loop with one transition to increment the token attribute value and other guarded transitions to initiate actions for tokens whose attribute values fall within a range specified in the guard predicate.

Sample structure: see Figure 10

Description: The pattern consists mainly of two applications of the "complete removal" pattern and a deterministic branching stage. In the first application of "complete removal" (places 'Storage and Enable_2 with transitions Increment and Start_2) (Figure 10 (b), "A") there is an additional inhibitor arc necessary between Temp_storage_1 and Enable_2 to ensure that all tokens have passed the branching stage before they get pumped back to the Storage place. The second application (Figure 10 (b), "A") (places Temp_storage_2 and Enable_1 with transitions Move and Start_1) uses the unmodified basic pattern.
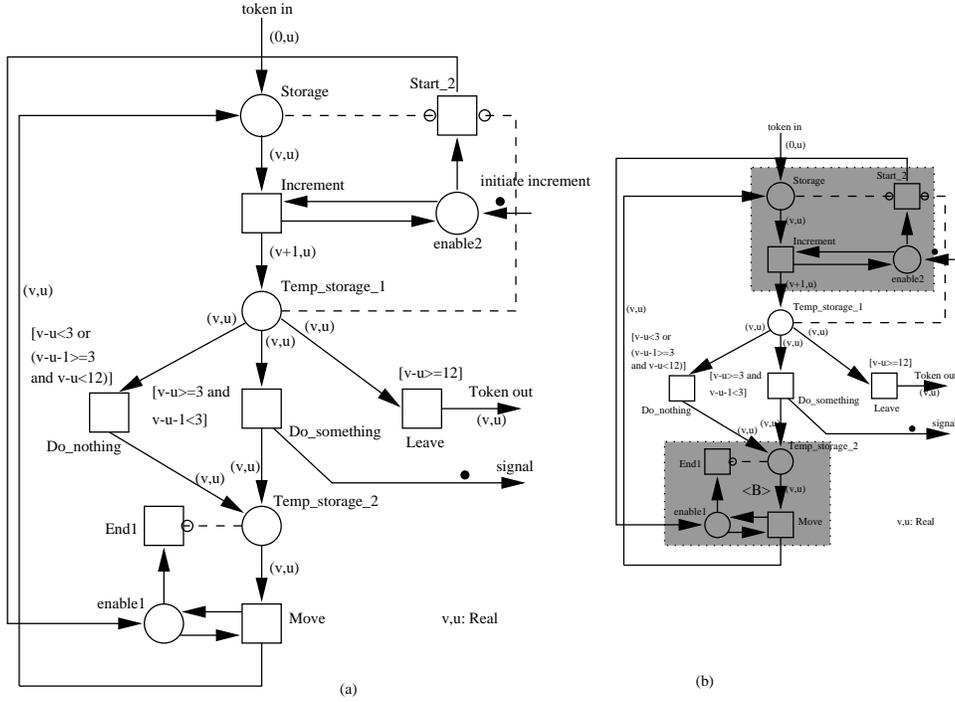
Uses pattern: Complete removal

Figure 10: a) Quasi-continuous movement b)Pattern "complete removal" within pattern "quasi-continuous movement"

4.5 Recursive stream processing

In this section we will introduce a pattern describing an architectural approach to the design of recursive model structures for the manipulation of streams of data [16]. Before we start into this, however, we will first give a simple pattern that shall serve as our implementation strategy for the stream concept throughout the subsequent presentation.

Name: Stream processor

Problem: When modeling processing elements that transform an ordered sequence (or *stream*) of tokens into another stream of tokens as Petri net components, we have to deal with the fact that incoming tokens are not serialized once they reside on a place. Therefore we have to establish a communication protocol between a component that acts as the *source* of a stream and one that acts as its *destination*. A processing element of the above kind will then act as either.

Example: Suppose we want to design a component that takes two consecutive elements from an input stream, performs some function on them and outputs its result to an output stream. We have to take care that elements are processed in order - both on the input and on the output side of out processing element.

14

Solution: We introduce a handshaking protocol between sender and receiver of a stream (similar protocols have been discussed as Petri net solutions to related problems, e.g. in [4]). The two sides might be declared as in Figure 11.

StreamSource<class T>   StreamDest<class T>

out : T   in : T
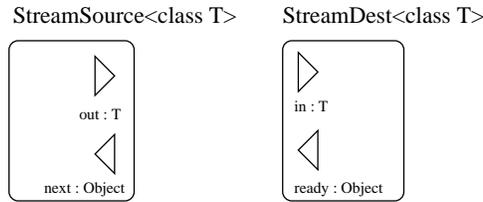
next : Object   ready : Object

Figure 11: Component classes for a stream protocol.

In order to make this a valid protocol, we also have to impose some constraints on the behavior of sender and receiver components. If we want the protocol to be 'pull-driven' (i.e. activity is initiated by the receiver), which is the convention we will use in the following, we have to require that the sender never issues more tokens on its `out` interface than it received on its `next` interface. The receiver, in turn, must never produce more than one token more on its `ready` interface than it received on its `in` interface. Thus when we connect the `out`/`in` and `next`/`ready` interfaces of two components obeying these rules, it is easy to see (and to prove) that tokens are sent by the `out`/`in` connection only after they are requested by a token along the `next`/`ready` connection. A push-driven protocol could just as easily be specified by corresponding constraints on the dynamics.

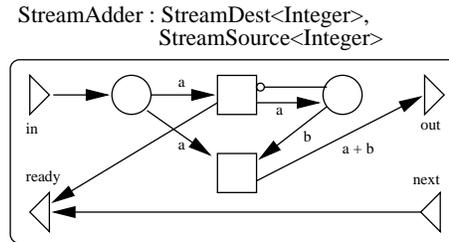**Sample Structure:** Coming back to the problem above, we might model this as shown in Figure 12.

StreamAdder : StreamDest<Integer>,
StreamSource<Integer>

in   a   a   out
ready   a   b   a + b   next

Figure 12: Performing a function on consecutive stream elements.

Description: This component class inherits from both **StreamDest** and **Stream-Source**. It sends its first 'ready' token when it receives a 'next' token. Then it moves the next incoming token to a buffer place, sends another 'ready' and adds it to the next incoming token, sending the result to its output.

With this we can now approach the larger problem of designing a stream processor which performs some recursive operations on its input.

Name: Recursive stream processor

Problem: An ordered sequence of incoming tokens has to be processed in a recursive fashion.

Example: Let us consider the processing performed by a Huffman decoder on a stream of bits. Recall that Huffman decoding converts a stream of bits into a stream of codes (say, for this example, characters) by using a binary tree data structure called a *Huffman tree*, the leaves of which contain individual characters. Initially starting at the root of the tree, it reads a bit from its input and depending on whether its value was 0 or 1, goes down one step in the tree to the left or to the right, respectively. If it hits a leaf, it writes the code contained in the leaf and starts the procedure at the root again. Otherwise it simply goes on with the subtree it went to in this step.
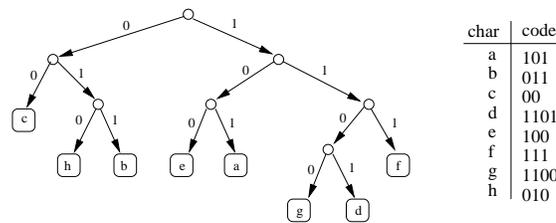

Figure 13: A Huffman tree and its code table.

Depending on the structure of the tree, a single character will thus be emitted for a varying number of input bits. For instance, the tree in Figure 13 provides the indicated mapping between bit sequences and the characters from a to h. The reader should convince himself that any sequence of bits can be uniquely mapped to a corresponding sequence of characters.

Clearly, this decoding algorithm could be implemented either iteratively or recursively. For the sake of this discussion, we will consider the latter approach – which will also be very concise when formulated in the language outlined in section 2.

Solution: The most elegant solution to this problem makes use of instantiation expressions as sketched in section 2. We construct for a given Huffman tree a component which has an instantiation structure isomorphic to it – it contains embedded components that are constructed according to the respective subtrees. This recursion takes place during the instantiation controlled by instantiation expressions.

Required net formalism: Since in the static case (i.e. unless we instantiate components at runtime) instantiation expressions may be transformed into a net without any instantiation expressions in a purely syntactical way, we do not rely on this language feature in order to apply this pattern – without it, the corresponding net could be produced by hand according

to the algorithm that would be implemented by the instantiation expression. Clearly, though, this can be tedious and error-prone for any but the smallest applications.

Structure/Description: First we will consider the task of computing *one* output character from the input stream. The abstract interface of a component for this purpose can be seen in Figure 14. In `in` interface accepts the incoming bits, after the component has been sent a token on the `select` interface. Each input is acknowledged by an output token at the `ack` interface. If a complete code has been received, the corresponding character is output at the `out` interface. In order to allow for an unambiguous decoding, we will also assume the dynamic constraint that the character is sent *not after* the corresponding acknowledgment.
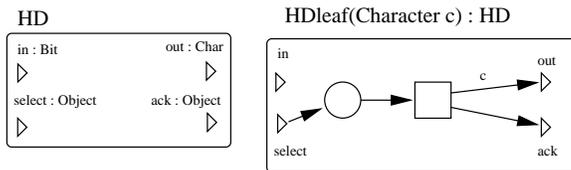

Figure 14: Abstract HD class and HDleaf subclass.

We will provide two concrete implementations for this interface - one corresponding to the simple case of a leaf in the Huffman tree, the other to that of a node. The leaf component is also shown in Figure 14. Its operation is simple: once it is activated by a token on its `select` interface, it outputs both an acknowledgment and the character it has been initialized to code. Doing this simultaneously, it obviously conforms to the behavioral constraint above.

The operation of the node component depicted in Figure 15 is more complex. It contains two embedded components representing its right and left subtree instantiated by an instantiation expression:

```
if isNode(t.left) then HDnode(t.left) else HDleaf(t.left.c)
```

This instantiates a `HDnode` component for a node subtree and a `HDleaf` componentn for a leaf. A node component operates in three phases to be distinguished by the marking of the places `Initial`, `LeftActive`, and `RightActive`. Initially, all are empty. When the component receives a `select` token, it acknowledges it and marks `Initial`. The next bit (on interface `in`)decides whether `LeftActive` or `RightActive` gets marked, and where all subsequent bits will be sent to. This continues until the respective subcomponent produced a character, which returns the component to its initial state (at this point we make use of a variant of the *A-before-B* pattern to determine whether we received a character before or with the acknowledge).

Finally, we have to wrap such a component into one that obeys the stream protocol defined above, which is shown in Figure 16.

This pattern has many uses beyond this simple example. For instance, a typical compiler can be viewed as a program that recursively processes
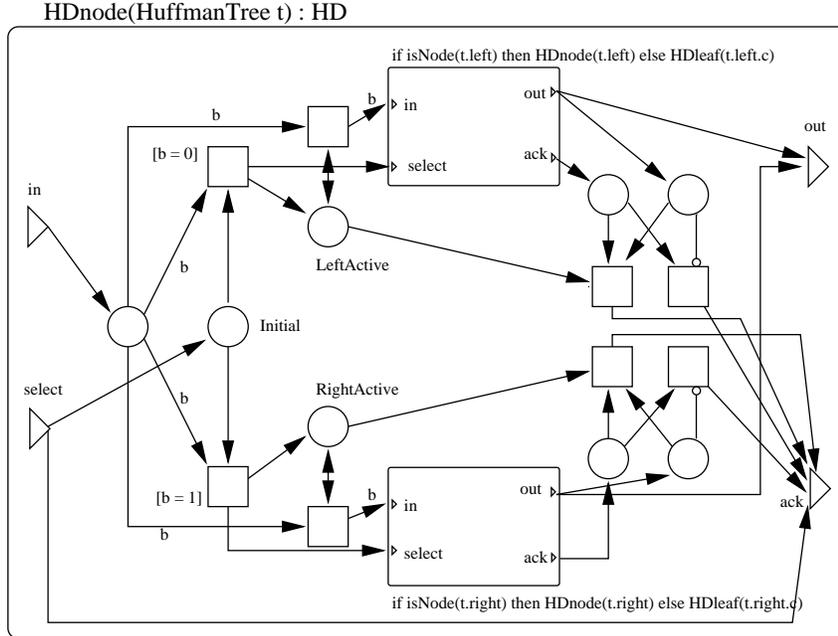
17

Figure 15: Decoder representing a node in the Huffman tree.

an input, constructs a tree-like data structure representing the abstract syntax and uses this structure to produce another data stream representing the compiled program.

Uses: *Stream processor* and a simplified variant of *A-before-B* which we call a *Switch*.

5   Conclusion

In this paper we suggested a style of presentation for design knowledge in Petri net systems modeling which is by no means complete or finished - on the contrary, we hope to spark a discussion in the Petri net community about how to present, categorize, and select this kind of knowledge for modeling tasks in the various fields where Petri nets might be used. It seems very likely to us that a design pattern catalog for workflow models might look different from one to be used by an engineer designing hardware systems – if only because the audiences in these two areas are so different.

While in this respect much can be learned from the software community, there is hope that the idiosyncrasies of Petri nets as the modeling formalism might make some things easier for us. In contrast to 'regular' programming languages, when describing design principles based on Petri nets, we are already starting from a well-defined, formal, and structured basis which, in combination with appropriate composition and hierarchy constructions, should simplify many modeling tasks. In this context, we suggested some constructions for composing hierarchical systems that we found useful in practical situations.

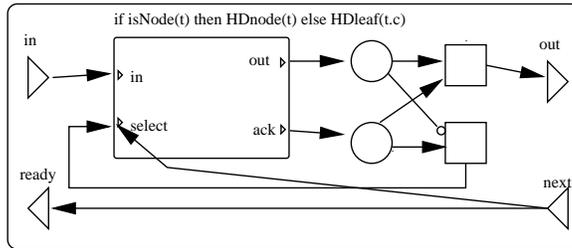HuffmanDecoder(HuffmanTree t) : StreamDest(Bit),
StreamSource(Char)

Figure 16: Wrapping a HD-component.

Furthermore, the formal definition of Petri nets suggests an equally formal description of at least some of the design patterns – not only in structural, but also in behavioral terms. This could be an essential building block of a *pattern language*, something already suggested by Christopher Alexander [1], which seems to be an ambitious endeavor in the heterogeneous and often informally defined world of programming languages, but which we believe might hold more promise for the domain of Petri nets – at least for a significant number of patterns relevant in practice. Such a set of formally defined patterns might also allow a formal and generic analysis of some of their properties that might be interesting when constructing a system from them.

Thus, there is still much work to be done before we can offer a catalog Petri net design patterns to be applied to practical modeling tasks. We see the most important areas as follows:

- Definition of appropriate pattern presentation styles and templates.

- Collection, identification, abstraction, and categorization of the existing body of Petri net design knowledge and the patterns behind it.

- Design of a sufficiently general Petri net language allowing for powerful mechanisms for composition and parametrization. The experience from and with different Petri net languages [7], [12], [18], [19], [24] must be taken into account as well as the huge body of knowledge from programming language design.

- Design of a pattern language, at least for a restricted set of patterns.

Experience shows that application knowledge of this kind enhances the acceptance of a tool or a language. Not only does it raise the productivity of individuals using it, a systematic categorization of design experience also promotes a common terminology and thus makes communication and discussion much more efficient. This paper is intended as a contribution towards this end.

19

References

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* Oxford University Press, 1977.

[2] Helmut Balzert. *Lehrbuch der Software-Technik*, volume 1. Spektrum Akademischer Verlag, 1996.

[3] B. Baumgarten, H. J. Burkhardt, P. Ochsenschlaeger, and R. Prinoth. The signing of a contract - a tree-structured application modelled with petri net building blocks. In G. Goos and J. Hartmanis, editors, *Advances in Petri Nets 1985*, number 222 in Lecture Notes in Computer Science. Springer, 1985.

[4] B. Baumgarten, P. Ochsenschläger, and R. Prinoth. *Building Blocks for Distributed System Design*, volume Protocol Specification, Testing, and Verification, pages 19–38. Elsevier Science Publishers B.V. (North-Holland), 1986.

[5] Bernd Baumgarten. *Petri-Netze, Grundlagen und Anwendungen.* Spektrum Akademischer Verlag, 1996.

[6] Wilfried Brauer, Robert Gold, and Walter Vogler. A survey of behaviour and equivalence preserving refinements of Petri nets. In *Advances in Petri Nets.*

[7] D. Buchs and N. Guelfi. CO-OPN: A concurrent object-oriented Petri net approach. In *Proceedings of the 12th International Conference on the Application and Theory of Petry Nets*, 1991.

[8] Heinz Juergen Burkhardt, Peter Ochsenschlaeger, and Rainer Prinoth. Product nets, a formal description technique for cooperating systems. GMD-Studien 165, GMD, Gesellschaft Fuer Mathematik und Datenverarbeitung mbH, 1989.

[9] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - A System of Patterns.* Wiley and Sons, 1996.

[10] Andrea Caloini, Gian Antonio Magnani, and Mauro Pezze. Software design of robot controllers with petri nets: a case-study. In *Proceedings of the 1996 IEEE International Conference on Systems, Man and Cybernetics*, 1996.

[11] James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design.* Addison-Wesley, 1995.

[12] Robert Esser. *An Object Oriented Petri Net Approach to Embedded System Design.* PhD thesis, ETH Zurich, 1996.

[13] H. Fuss. P-t-netze zur numerischen simulation von asynchronen fluessen. In J. Hartmanis G. Goos, editor, *GI - 4. Jahrestagung, Berlin 0.-12. Oktober 1974*, number 26 in Lecture Notes in Computer Science, pages 326–335. GI, Gesellschaft fuer Informatik, Springer-Verlag, 1975.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[15] Ralf Guido Herrtwich and Guenther Hommel. *Nebenlaeufige Programme*. Springer Verlag, 1994.

[16] Jörn W. Janneck and Martin Naedele. Modeling hierarchical and recursive structures using parametric petri nets. submitted to International Conference on Application of Concurrency to System Design (CSD98), 1998.

[17] Kurt Jensen. Coloured Petri nets: A high level language for system design and analysis. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 483 of *Lecture Notes of Computer Science*, 1990.

[18] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, volume 1: Basic Concepts of *EATCS Monographs in Computer Science*. Springer-Verlag, 1992.

[19] Charles A. Lakos. Object Petri nets - definition and relationship to coloured nets. Technical Report TR94-3, Computer Science Department, University of Tasmania, 1994.

[20] Charles A. Lakos. From coloured Petri nets to object Petri nets. In *Proceedings of the 15th International Conference on Applications and Theory of Petri Nets*, Lecture Notes of Computer Science, 1995.

[21] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, 1981.

[22] Wolfgang Reisig. *Petri Nets, An Introduction*. Springer-Verlag, 1985.

[23] Wolfgang Reisig. *A Primer in Petri Net Design*. Springer-Verlag, 1992.

[24] Rüdiger Valk. On processes of object Petri nets. Technical Report 185, Fachbereich Informatik, Universität Hamburg, 1996.

[25] Ruediger Valk. Self-modifying nets, a nautural extension of petri nets. In J. Hartmanis G. Goos, editor, *Automata, Languages and Programming, Fifth Colloquium, Undine, July 17-21, 1978*, number 62 in Lecture Notes in Computer Science, pages 464–476. Springer-Verlag, 1978.

[26] John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors. *Pattern Languages of Program Design 2*. Addison-Wesley, 1996.

[27] P. Wegner. Inheritance as an incremental modification mechanism. In *Proceedings of ECOOP 88 - European Conference on Object-Oriented Programming*, number 322 in Lecture Notes of Computer Science, pages 55–77. Springer Verlag, 1988.