published in:
Workshop on Software Engineering and Formal Methods, Petri Nets 2002, Adelaide, Australia

Higher-order Petri net modelling – techniques and applications

Jörn W. Janneck
EECS Department
University of California at Berkeley
Berkeley, CA 94720, U.S.A.
Email: janneck@eecs.berkeley.edu

Robert Esser
Department of Computer Science
The University of Adelaide
Adelaide, S.A. 5005 Australia
Email: esser@cs.adelaide.edu.au

Abstract

Higher-order Petri nets are a class of high-level Petri nets, in which Petri nets themselves are first-class objects. Here tokens may represent Petri nets and Petri nets may be the values of parameters and variables, as well as the result of computations performed during the occurrence of transitions. These features facilitate a number of very powerful higher-order modelling techniques, making Petri nets much more flexible, compositional, and the resulting models more reusable. This work explores the usefulness of some of these techniques by looking at them from an application point of view and by illustrating them with small to medium-sized application examples.

Keywords: higher-order Petri nets, component models, modeling techniques

1 Introduction

Petri nets are a powerful modelling notation for concurrent systems, but in their basic form they lack any notion of compositionality or abstraction. Over the years various approaches of composing Petri nets models have been used with more or less practical relevance. We will discuss some of them in section 2.

As part of the Moses project (Moses 2001) we developed a simple extension to Petri nets – a form of *higher-order* Petri nets. The term 'higher-order' is taken from the domain of programming languages, where it describes procedures or functions that can take other procedures or functions as parameters, produce them as results, or simply manipulate them as first-class data objects. Similarly, the extension we propose allows a Petri net to appear wherever any other data object can appear, as a token, as a parameter, or as the value of a computation.

These facilities have been used in practical modelling for some time now, and this paper tries to extract some of the fundamental modelling techniques that are made possible by higher-order modelling constructs.

The rest of this paper is structured as follows. After reviewing some related work in section 2, section 3 informally introduces the basic notions and concepts, as well as some terminology. The main body of this paper is section 4, where we present a number of techniques of varying complexity that are all made possible by the higher-order constructs introduced in the preceding section. Finally, we conclude in section 5 with some discussion and an outlook for future research.

2 Related work

One key extension to basic Petri nets that make them more useful for practical modelling are high-level Petri nets (such as Colored Petri Nets (Jensen 1992) or Predicate/Transition Nets (Genrich &Lautenbach 1981)). They provide powerful means for modelling concurrency and synchronization in a formal yet intuitive manner. Like basic Petri nets, however, they lack constructs that make them compositional.

A modelling technique is considered compositional if it facilitates modular decomposition techniques that conform to the *open-closed principle* (Meyer 1988): It supports the development of individual model parts ('components') that are *closed* in the sense that knowledge of their internal structure is not necessary in order to use them (provided one has a sufficiently precise description of their behavior), while at the same time *open* in the sense that such a component can provide an arbitrary degree of configurability, has documented interfaces which can be connected to the interfaces of other components, can be used independently from its contexts and is free to make use of other components.

Previous work on composing Petri net models includes various techniques for refining places and transitions (Brauer, Gold &Vogler 1994, Vogler 1987). (Jensen 1992) also defines a concept of hierarchy and composition for Colored Petri Nets (CPN) that allows transitions to be replaced by subnets. According to (Lakos 1997) transition and place refinement approaches cannot even be considered to support proper abstraction, since the refinements do not retain the properties of the places and transitions.

Object Petri Nets (OPN) (Lakos 1994, Lakos 1995) allow tokens to represent arbitrary subnets. Tokens are instances of classes, and a class may be instantiated dynamically any number of times. This mechanism allows the abstraction and encapsulation of activity. A net may access public attributes of subnet tokens in transition actions. OPN also defines a variant of place refinement: A net may change the state of a subnet (if the latter is defined to refine a so-called *superplace*) by sending it tokens and retrieving tokens from it, based on a synchronous communication protocol (suggested in (Christensen &Hasen 1994) for CPN). Acceptance and offering of tokens is controlled by the logic of the subnet, which can lead to inconsistencies and anomalous situations (cf. (Lakos 1996) for a more detailed discussion).

Object Systems (Valk 1996) define two-level nets (with an informal description of what n-level nets might look like) where network components communicate by synchronizing transitions. This reduces some

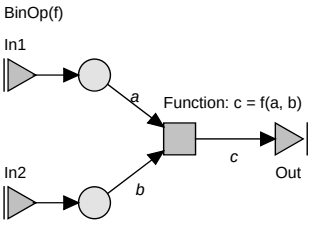

Figure 1: A simple Petri net component with ports.

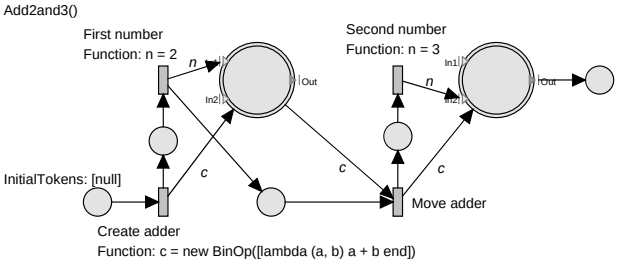

Figure 2: A component creating and using the component from Fig. 1.

of the problems of the synchronous communication suggested for OPN, but relating transitions to one another along with the restriction to two levels (or indeed, the binding of components to a level) makes 'object nets' less compositional than OPN objects.

The component scheme used in this paper borrows from the above approaches in that components communicate asynchronously and tokens are allowed to 'contain' (designate) arbitrary subnets. Instead of refining places or transitions, contained components are hooked up to their environment when they reside on a place. We will now explain the details of this mechanism.

3 Higher-order Petri nets

In this section we will informally present a few Petri net extensions that facilitate higher-order modelling—a formal definition and semantics can be found in (Janneck 1998). Although integrated into the Moses tool suite (Esser &Janneck 2001, Moses 2001), we believe that these extensions could be combined with essentially any high-level Petri net dialect, such as e.g. Colored Petri Nets (Jensen 1992). In addition to the specific higher-order constructs, we assume that tokens can carry arbitrary data objects, as well as functions—and of course Petri net components.

The first addition to Petri nets is a notion of *component*, which has *parameters*, can be instantiated, and which can be connected to its environment via *ports*. Fig. 1 shows a very simple Petri net component. It has one parameter f, which happens to be a function, and two input ports and one output port. In this particular component model, we only allow input ports to be connected to places, and output ports to be connected to transitions. Intuitively, the idea is that tokens enter the component via its input ports and are then placed onto the connected places, while tokens produced by a transition firing leave the component via a connected output port.[1]

The next extension concerns the *use* of such a component in the context of another Petri net. Assume

[1] Other component models are conceivable, notably one that allows a type of input port that can be connected to transitions, and a kind of output port that can be connected to places. However in this case structural conflicts can cross component boundaries.

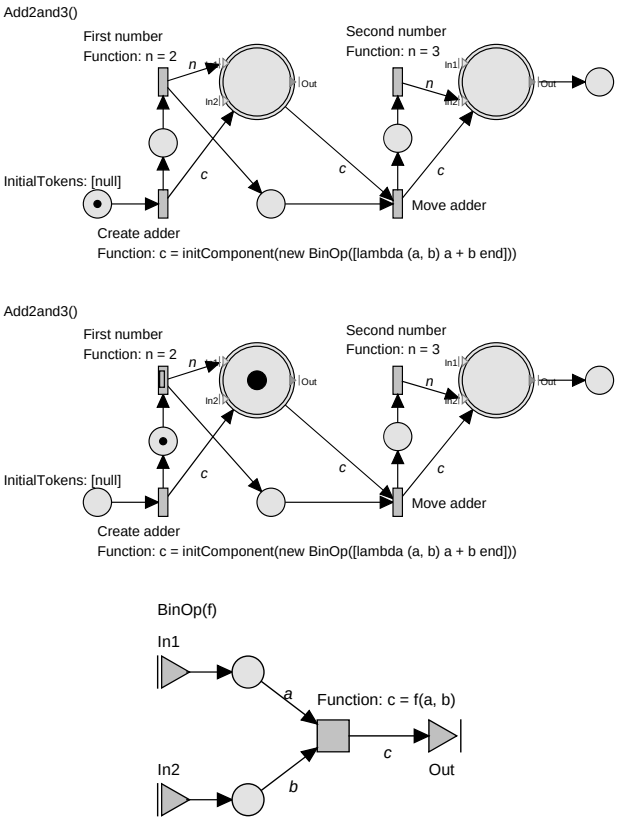

Figure 3: A somewhat roundabout way to add 2 and 3. (A)

that the component resides as a token on a place in some other net—we now would like to connect it to the places and transitions in the net in which it resides. We do this by adding ports to places, and allowing connections to these ports, in addition to connections to the place itself. Fig. 2 shows two instances of this: the two large double-rimmed places[2] each have three ports (labeled `In1`, `In2`, and `Out`), and arcs are going to and from the places themselves, as well as (some of) the ports. We call places that have ports *container places* or sometimes also *higher-order places*, and the ports *container ports*.

Note that transitions may be connected to input container ports, and the output container ports connected to places. Intuitively, when a transition that is connected to an input container port produces a token, that token is *not* put onto the container place itself, rather it is 'sent' to the correspondingly named input ports of all components currently residing on that place. Similarly, when any contained component produces a token at one of its output ports, that token 'appears' at the corresponding output container ports of all container places it resides on, and is then put onto all places connected to that port.

Figs. 3, 4 and 5 together show a run of the component in Fig. 2—it depicts the marking of the `Add2and3` as well as the `BinOp` component for the six steps of the run, the former on top of the other.

In the first state, there is no `BinOp` component. When the transition marked *Create adder* fires, it instantiates a `BinOp` component with a function that adds two numbers as a parameter. The next state shows the `BinOp` component residing on the left large place, with no tokens as its initial marking. Now the

[2] The double rim is really only a visual cue that these places are expected to contain components—apart from having ports, they are otherwise entirely normal places.

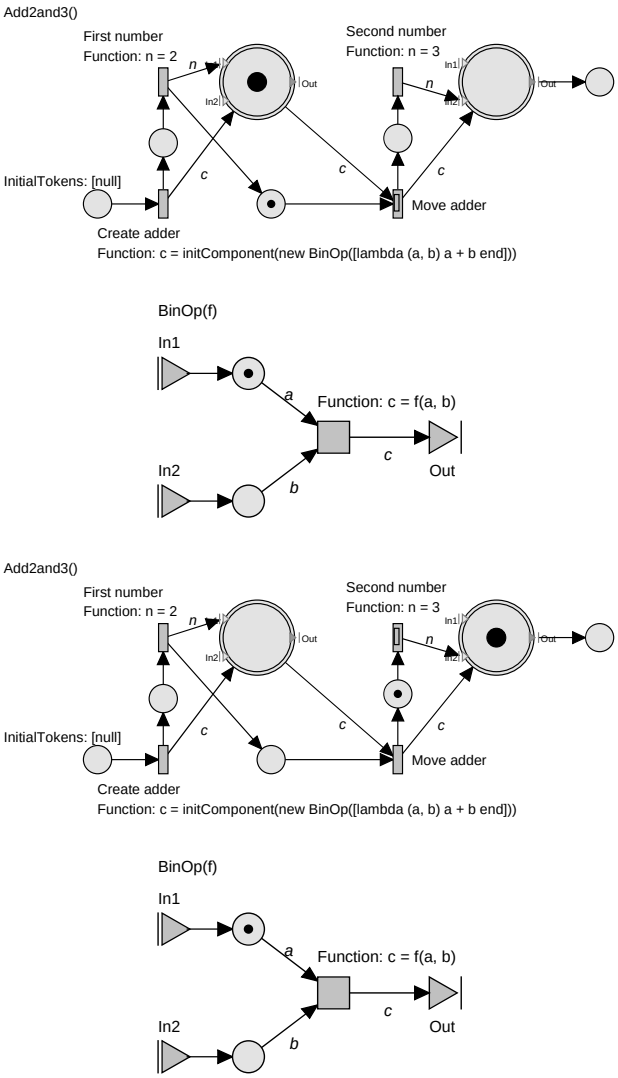

Figure 4: A somewhat roundabout way to add 2 and 3. (B)

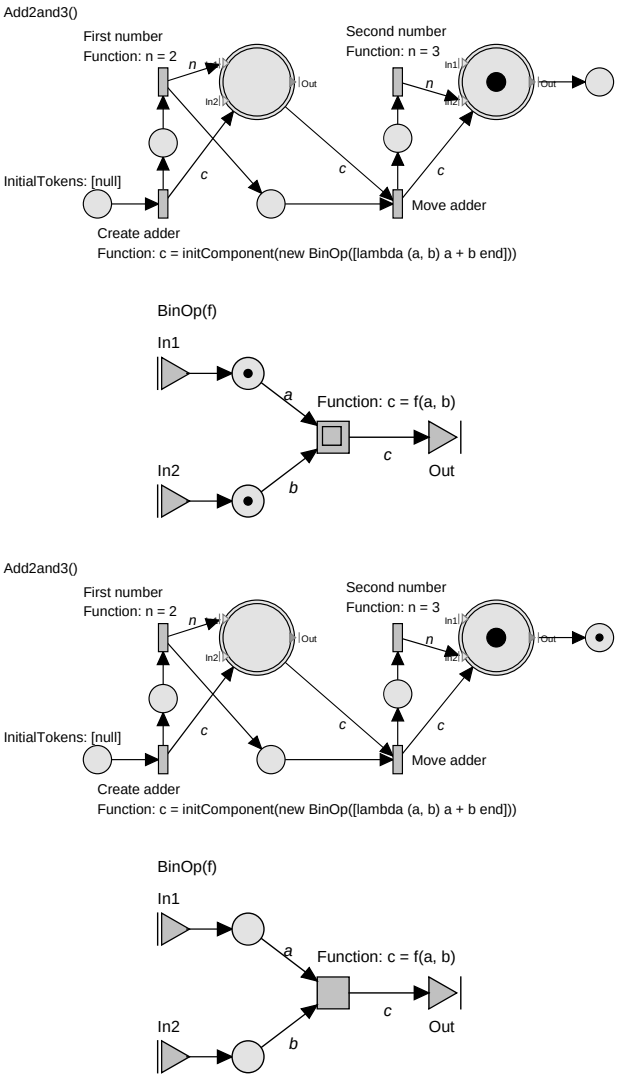

Figure 5: A somewhat roundabout way to add 2 and 3. (C)

transition labeled *First number* fires and sends a token into the `BinOp` component, which is then put onto the top place.

The transition labeled *Move adder* then fires, removing the `BinOp` component from one place and adding it onto another (fig. 4). Note that the state of the `BinOp` component itself does not change, in fact it has no way of telling that it has been 'moved' by this state transition. Now the transition labeled *Second number* sends another token to the other input port of the `BinOp` component, finally activating its transition. When it fires in the next step, it sends a token to its output port, which then appears on the place connected to the corresponding place output port in the `Add2and3` component (fig. 5).

Along with the two new syntactical elements (ports and container ports and places) we assume the existence of a facility for creating new components and for parameterizing them (the `new` operator used in the *Create adder* transition). We will now discuss a number of more serious applications of these constructs.

4 Techniques and applications

In this section we will discuss and illustrate, via concrete examples, a variety of modelling techniques that take advantage of higher-order capabilities. Due to space constraints, we will not be able to present each application example in detail, but the models we give are complete,[3] and the explanation is intended to illustrate the use of higher-order constructs in each case.

4.1 Regular parametric structures

Many models contain structures that are static, regular, and repetitive, but which either are not completely known at model construction time, or which are so big that constructing them by hand is impractical.

Consider for example the task of modelling a structure that performs Huffman decoding of a bit stream based on a given Huffman tree.[4] 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,

[3] They are available for download at (Moses 2001), together with a tool that executes them.

[4] As e.g. in the recommendation H.263 of the ITU for video coding (ITU-T 1996).

goes down one step in the tree to the left or to the right, respectively. If it hits a leaf, it writes the code and starts the procedure at the root again. Otherwise it simply continues with the remaining subtree.

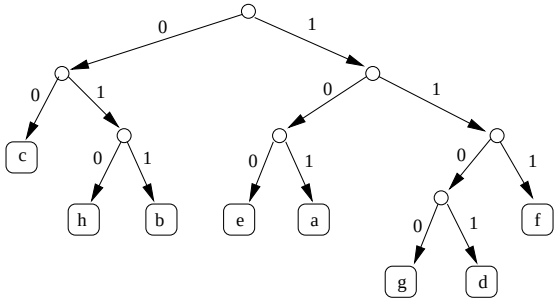

Figure 6: A small Huffman tree.

Depending on the structure of the tree, a single code will be emitted for a varying number of input bits. For instance, the tree in figure 6 yields the code "c" for the sequence "00", the code "e" for the sequence "100", and the code "d" for the sequence "1101".

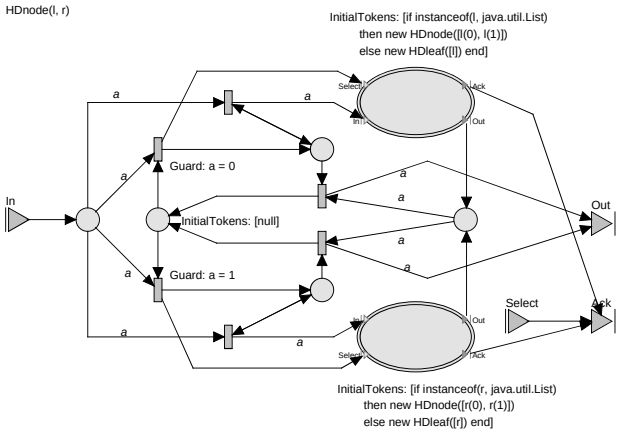

Figure 7: Huffman decoder node component.

One way of implementing this would be by having each node of the Huffman tree represented by one part of the model, and bits are forwarded down the tree structure until a leaf node is encountered, producing the output code.[5]

Obviously, we would not like to reconstruct the entire model for each different Huffman tree. Instead, we would like to use the higher-order facilities to construct a model that, when parameterized with a specific Huffman tree, automatically instantiates the corresponding model structure.

Fig. 7 shows this model for an internal node. Its parameters are the left and the right subtree, respectively, and it constructs a corresponding submodel for them by creating an initial token inside the two large container places on the top (for the left subtree) and the bottom (for the right subtree).

Initially, it contains a token on the central place, and when some input arrives on its `In` port, it will send a token to the `Select` input port of the corresponding submodel, depending on whether the input token is 0 or 1. It will also remove the central token and place one on one of the two places next to the container places, ensuring that future input is directly forwarded down the hierarchy.

The submodel will respond with either a token at its `Ack` output port (which is immediately forwarded to the corresponding port of this model), or by producing an output code at its `Out` port. In that case, the initial state is restored, and the code is sent to the `Out` port of this model.

Note that any token arriving on the `Select` input port of this model is immediately sent to the `Ack` output port.

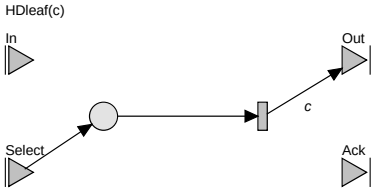

Figure 8: Huffman decoder leaf component.

Fig. 8 shows the model for the leaf nodes. Whenever it receives a token on the `Select` port, a leaf has been reached and therefore the output code (a parameter of the leaf component) is produced.

Note that the proper functioning of this system relies on every component observing a certain *protocol*—e.g., new input must not be sent until the component has produced a token at either output port. We will encounter similar protocols in the following examples, and discuss them in section 5.

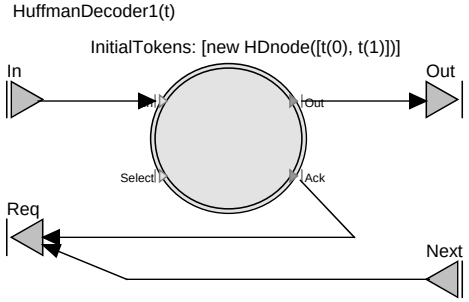

Figure 9: The top level Huffman decoder.

Fig. 9 shows how the above two components are used to build a parameterized Huffman decoder Petri net. The decoder realizes a simple dataflow protocol in which data tokens are sent only in response to an explicit request—each token emitted from `Req` is answered by one token coming in at `In`, and similarly tokens are only sent to `Out` in response to request tokens coming in at `Next`. The protocol requires that no two request tokens are sent without an intervening reply, and that no 'data' token is sent without a previous request.

4.2 Incremental net construction

Even though parametric, the network in the previous example was static, in the sense that—once constructed—it did not change its structure during the runtime of the system. For some models, the model structure needs to be built at runtime, based on the computation performed by the model.

The classical example of such a system is the Sieve of Eratosthenes, a simple method for finding prime numbers, that proceeds by eliminating multiples of previously found primes from the sequence of natural

[5] There are of course simpler ways of doing this, but explicitly representing the tree structure in the model structure may be, e.g., an interesting step towards a hardware realization, where parts of the model would directly correspond to pieces of silicon.

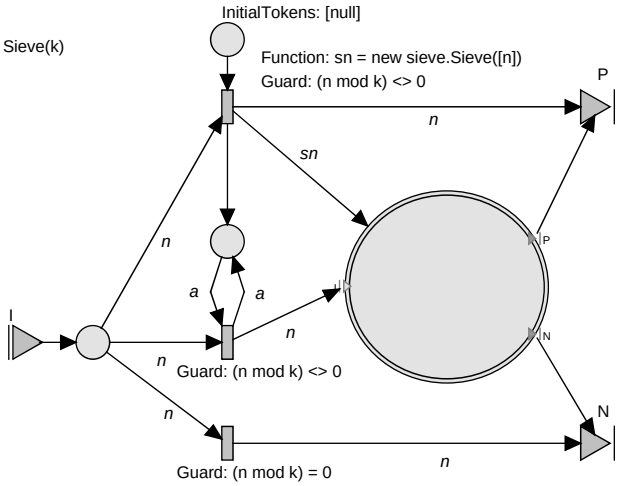

Figure 10: The sieve of Eratosthenes.

numbers.[6]

The model in Fig. 10 finds the prime numbers (starting at 3), assuming it is fed (to its input port `I`) the natural numbers in ascending order starting from 2, and it is instantiated with its parameter `k` set to 2.

Initially, only the top place has a token. For any token arriving at port `I`, the component checks whether it is divisible by `k`, and if so, sends it out to output port `N`. If it is *not* divisible by `k`, then it is a new prime, and it is sent to the output port `P`. At the same time, a new component is created, with the newly found number as its parameter, and is sent to the container place. From then on, every other number that is not divisible by `k` will be sent to the input port of that contained component, and its output will be forwarded to the corresponding output ports of this component.

It is easy to see that the sequence produced at the top-level `P` output port is just the prime numbers, and that all other numbers will be sent out at the `N` output port. It is also simple to prove these properties—we will come back to this in section 5.

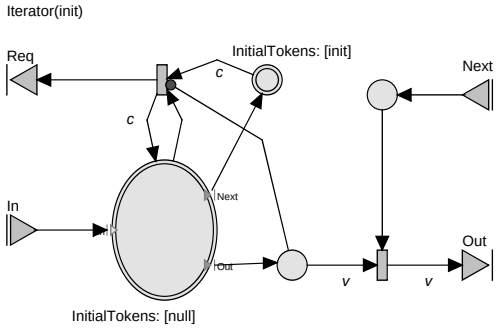

Figure 11: A higher-order iterator.

4.3 Dynamic iteration

The next technique (as many other ideas in this work) is inspired by the functional programming technique called *continuation*. Continuations are a way to add some imperative flavor to a functional program—basically, a function performs one part of the computation, and returns another function whose invocation would perform the next 'step', which also returns such a function and so on.

Fig. 11 shows a higher-order component which essentially uses the idea of a continuation for implementing a dataflow component that realizes the same protocol as the Huffman decoder in Fig. 9, i.e. data is explicitly requested by a receiver before it is sent.

The `Iterator` component has another component as a parameter, which it initially places on the small container place, which happens to be without ports, as nothing needs to be connected to it. A null token is placed on the large container place, activating the top transition. When this fires, it removes the null token from the large container place, and adds the component passed as a parameter onto it instead. It also emits a token via its `Req` output port.

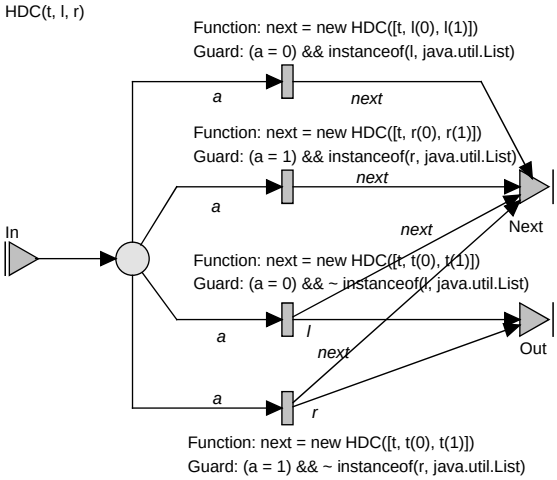

Figure 12: An iterable Huffman decoder component.

At some point, this will be answered by a token received on the `In` input port. This is immediately sent to the `In` port of the contained component, which is now expected to behave in either one of two ways. It may just acknowledge the token by simply emitting a token at its `Next` output port. In that case, the cycle starts again, i.e. that token will be put onto the small container place, activating the top transition, which will fire and replace the emitting component on the large container place with the token it emitted, while sending a token out at its `Req` port. In other words, the token emitted at the `Next` port is the component that will process the next input token, i.e. the continuation of the currently contained component.

However the contained component may also additionally produce an output token at its `Out` port (which must be produced earlier than the continuation token). In this case, the token inhibits the firing of the top transition until it is removed from that place. This will happen once a token is received at the `Next` input port, and the rightmost transition fires, sending the output token to the `Out` output port.

Fig. 12 shows a Huffman decoder component that can be iterated in this way. It has three parameters: the complete Huffman tree, and the current right and left subtrees. When it receives a token, and the corresponding subtree is a `List` (indicating it is not a leaf), it constructs a new instance of the same component class, but with the (smaller) subtrees of the subtree chosen by the incoming bit as parameters, sending that component to the `Next` output port as its continuation.

If the subtree chosen by the input bit is not a list, it is a leaf, and the continuation is a component instance parameterized with the top two subtrees, while

[6] Once again, depending on the inscription language there may be simpler ways than actually representing the entire filter in the structure of the Petri net. This sieve is merely illustrating the technique—more elaborate real-world examples, such as adaptive signal filters, required too much space.

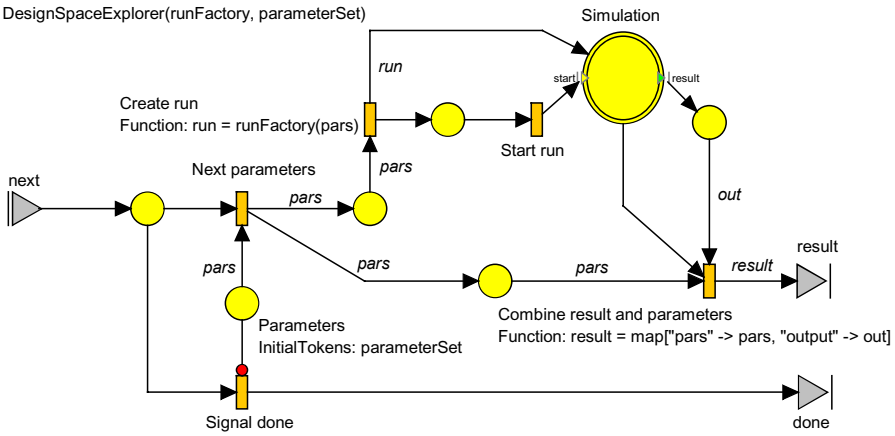

Figure 14: A design-space explorer.

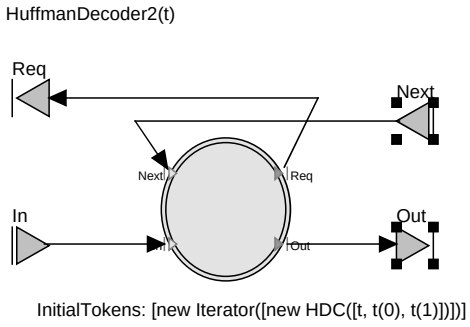

Figure 13: A Huffman decoder using the `Iterator`.

at the same time the corresponding output token is generated.[7]

Fig. 13 shows how to construct a Huffman decoder component from the `Iterator` and the `HDC` components above.

4.4 Component factories

In the previous section, components constructed other components, which were intended to be used as continuations for their creators. We will now look at a generalization of that technique, where the key job of a component is to construct other components—inspired by design pattern terminology (Gamma, Helm, Johnson &Vlissides 1995), we call such components *component factories*.

Consider the component in Fig. 14.[8] In this model a collection of *parameter tuples* and a function called `runFactory` are used to create a series of instances of a parameterizable system, in an exploration of the system's design-space. When applied to an element of that collection, the `runFactory` is expected to produce a component that is started by sending it a token, and terminates with some result token. The `DesignSpaceExplorer` component does this for each token sent to it on its `next` input port, sending the result of the run to its `result` output port, until the set of parameter tuples is exhausted, and then it responds by sending a token to its `done` output port.

Even though there is a *factory function* involved that creates components we would not call the `DesignSpaceExplorer` itself a component factory. This is because it does not make the instantiated components available to its environment—it just runs them and forwards the results. Nevertheless it is an interesting higher-order technique in its own right.

Although this is a powerful component for exploring a large number of parameter combinations without having to manually create a model for each, in practice design-space exploration is usually not done like this. The design-spaces of many systems are just too big to explore in this way—most often, exploration is performed in *generations*, where an initial set of parameter settings is created, its results are collected, and from this a new set is generated and so on. Many exploration and optimization strategies, such as hill-climbing, simulated annealing, or evolutionary methods work in this way.

Such a system can be very elegantly built using a component factory: The strategy that generates one generation from the results of the previous one can be modeled as a component that receives input data (the results from the current generation) and when triggered creates a complete new `DesignSpaceExplorer` component that runs the next generation. Fig. 15 shows a simple evolutionary strategy, which effectively is a factory for `DesignSpaceExplorer` components.

The details of this component are not very important—basically, it received results on its `addResult` input port, acknowledging them on its `resultAck` output port. When it receives a token on its `nextGen` input port, it computes a new set of parameter tuples (using the usual selection/mutation/crossover operations of evolutionary algorithms), and then instantiates a `DesignSpaceExplorer` component, which it sends to its `dseOut` output port.

Fig. 16 shows how the component factory (the *Strategy*) is used to build a generational design-space explorer.

Again, we could have built the generational explorer without a component factory, by simply designing the original design-space explorer such that we could send it new parameter tuple sets (rather than making these a parameter, and thus immutable). However, that would imply that we either foresaw the kind of generational design space exploration required, or that we are prepared to change the `DesignSpaceExplorer` to allow this. Also, the current design is significantly more flexible, as it allows a strategy to instantiate a different kind of design-space explorer, which may be more appropriate for its purpose.

[7] Obviously, constructing a new component for each bit is terribly wasteful, and something we have done here for simplicity. In practice, if this modelling technique is used, the components are often cached and reused.

[8] This component is derived from work on automated design-space exploration in (Esser &Janneck 2000).

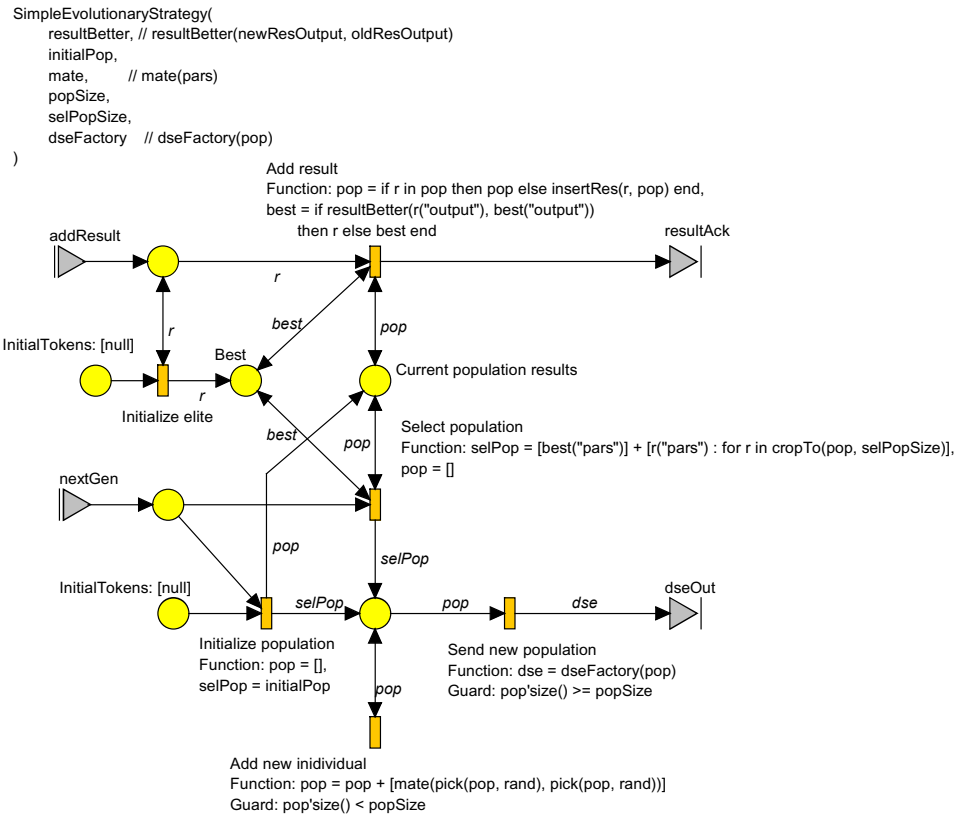

Figure 15: An evolutionary strategy acting as a design-space explorer factory.

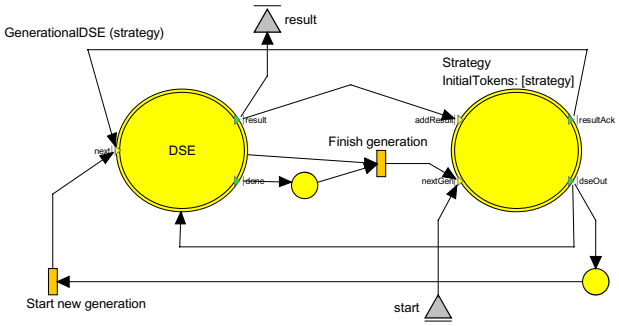

Figure 16: A generational design-space explorer using design-space explorer factories.

4.5 Coordinating tasks

So far, the examples had (at least in the parts we discussed) very little concurrency, which is rather unusual for Petri net models. This was merely for conceptual simplicity, and the following example will in fact be massively concurrent.

The task is to compute (an approximation to) the Mandelbrot set. We divide this task into two components: one, the `MandelbrotSlave`, computes an iteration count for a given point, the other, the `Master`, coordinates any number of those slaves and provides them with new points to compute, collects the results, and produces the appropriate output. Fig. 17a shows the `MandelbrotSlave`, taking a delay (we want to simulate and analyze the performance, say) and a maximal iteration count as parameters. It has an input port, where it receives *tasks*, which are tuples of the form $[slave, point]$. If the *slave* part does not identify a given slave component, it will simply remove the task token without doing anything else. If it *does* identify the slave component, then the task is accepted, and an iteration takes place, until the limit is reached or the function value diverges. Then the result (number of iterations) is written out.

The `Master` is shown in Fig. 17b. This is a generic component, its parameters are a set of slaves, an initial task (here: the starting point), a function computing the next task (the next point to be computed), and boolean function that determines whether there are any more tasks to be done. The master takes one of the available slaves (those contained in the bottom right place), computes a new task, and sends that task together with the slave to all slaves—note that sending a token to a container input port has multicast semantics, i.e. all components currently residing on the container place will receive the token.[9]

When a slave finally computes a result, the slave is again registered as available and the result is sent to the `TaskResult` output port. By giving the `Master` different sets of slaves with different delay distributions, system performance can be analyzed.

4.6 Collecting data

When evaluating a system such as the one computing the Mandelbrot set, it is often necessary to look at the steps performed by the individual components and gather statistical data about their operation. Assume we want to track for each slave when an iteration starts and ends. Obviously, this data has to be sent outside of the component that produces it.

We could do this by adding a port or two to the slave. Unfortunately, we would have to change the `Master` component as well, since it would have to

[9] Other solutions are possible: Slaves could be generated with their task as a parameter, and destroyed when they are finished. They could be taken to a place where they are guaranteed to be alone, to avoid the multicast/filter overhead. Or one could extend the higher-order language with an addressing construct. Which of these is most appropriate depends on the concrete system to be modeled.

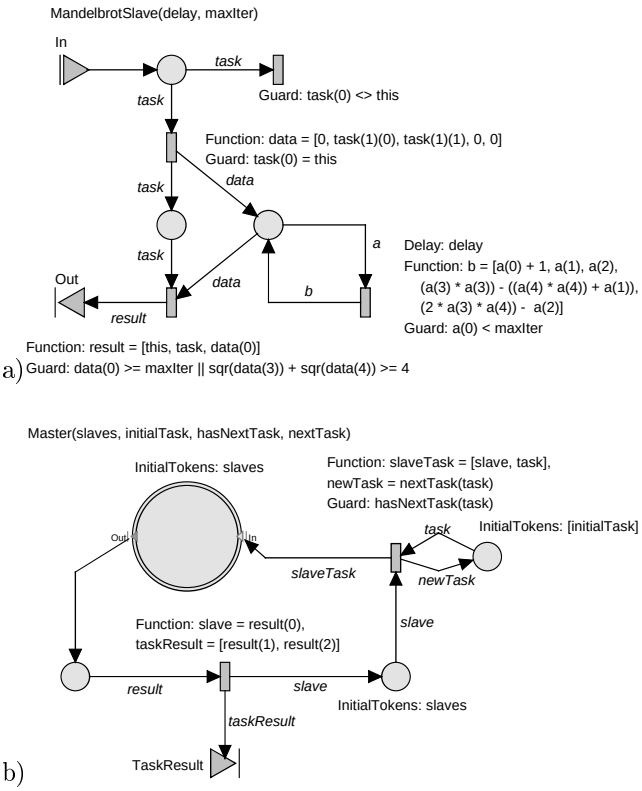

Figure 17: Mandelbrot point computation—a slave, and a master driving a set of slaves.

forward those 'statistical' tokens, and so would any component around it, until we reach the top level, where we would hopefully catch those tokens and do something useful with them, such as write them to a file or a database. This is a very annoying solution, as we would have to modify generic components such as `Master`, and quite possibly would have to do so repeatedly, for each new piece of statistics that we would like to generate.

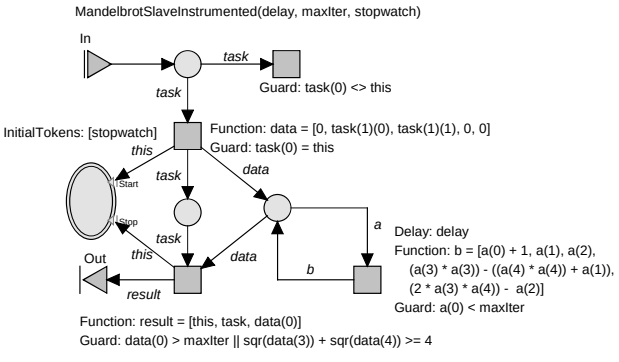

Figure 18: A version of 17a using a simple performance monitor

A better solution is suggested in Fig. 18: rather than communicating with the performance monitor (here a simple component receiving tokens at its `Start` and `Stop` input ports) 'upward' across any number of levels of containment, we pass the monitor component *into* the slaves. All slaves will be handed (as a parameter) the same identical performance monitor, which is therefore contained in (at least) as many container places as there are slaves. This way only the slaves need to be adapted (and of course whoever creates them), and the performance monitor component can evaluate, store, or print the statistics without the surrounding components (such as the `Master`) even being aware of it.

5 Discussion and conclusion

In this paper we have presented a number of practical modelling techniques that are a consequence of introducing higher-order constructs into a high-level Petri net language. They rely on a component model for Petri nets, and on a construct that allows Petri nets to be treated as tokens, and be dynamically connected to and disconnected from other Petri net components.

None of the modelling tasks presented here *needed* higher-order constructs, in the sense that there was always a way to do without them. So these constructs do not, in a formal sense, necessarily extend the expressiveness of a Petri net language. On the other hand, we do believe that they are a significant contribution to the practical modelling convenience of a Petri net language. They allow much more flexible compositions of Petri net models, and much more rigorous abstraction and encapsulation than many other ways of composing models. This abstraction facilitates new ways of constructing models: models can be constructed like software, in teams, with people relying on defined interfaces between modules/components. This in turn has an impact on possible model sizes and complexity. In our experience, higher-order constructs open many new application areas to Petri net modelling.

Throughout the examples, we had to assume certain behavior of the environment, and in turn could guarantee certain component behaviors—this was the essence of the *protocols* that we described in several places. The relatively simple formal semantics of Petri nets together with an assume-guarantee style of compositional reasoning about component interfaces and their connection could be the key to proving useful properties about many classes of dynamic, unbounded size systems. This is an exciting research direction that we are exploring.

We had already mentioned that there are a number of alternatives to the particular component model chosen in this work. The key rationale for this model is that it is not reliant on Petri nets, and in fact allows a large number of different modelling languages implement a component. This was one of the key design goals of the Moses project (Moses 2001). However in a Petri net context, much richer component models are conceivable, and exploring these is another interesting direction of research.

On the more practical side, the work outlined in this paper could be approached more systematically, resulting either in a catalog of *design patterns* for Petri net modelling (cf. (Naedele &Janneck 1998, Gamma et al. 1995)) (or maybe even for generic higher-order modelling), or possibly in a library of reusable model components. Both will tremendously improve the efficiency of model creation.

Acknowledgments

Part of this work was conducted in the Ptolemy project (http://ptolemy.eecs.berkeley.edu) and supported in part by the MARCO/DARPA Gigascale Silicon Research Center (http://www.gigascale.org). Their support is gratefully acknowledged.

Part of the work was conducted in the context of the Moses project (Moses 2001). All models were created using the Moses tool suite, which can be downloaded free of charge from the web site.

References

Brauer, W., Gold, R. &Vogler, W. (1994), A Survey of Behaviour and Equivalence Preserving Refinements of Petri nets, *in* 'Advances in Petri Nets'

Christensen, S. &Hasen N.D. (1994), Coloured Petri Nets Extended with Channels for Synchronous Communication, *in* 'Proceedings of the 15th International Conference on the Application and Theory of Petri Nets', Vol. 815, pp. 159-178, Lecture Notes in Computer Science, Springer-Verlag.

Esser, R. &Janneck, J. W. (2000), Exploratory Performance Evaluation using Dynamic and Parametric Petri Nets, *in* 'Proceedings of the HPC 2000', pp. 357-364, Society for Computer Simulation.

Esser, R. &Janneck, J. W. (2001), Moses: A tool suite for visual modeling of discrete-event systems, *in* 'Symposia on Human-Centric Computing (HCC '01)', pp. 272-279, IEEE Computer Society.

Gamma, E., Helm, R., Johnson, R. &Vlissides, J. (1995), 'Design Patterns, Elements of Reusable Object-Oriented Software', Addison-Wesley.

Genrich, H. J. &Lautenbach, K. (1981), System Modelling with High-Level Petri Nets, *in* 'Theoretical Computer Science', Vol. 13, pp. 109-136, Theoretical Computer Science, North-Holland.

Janneck, J. W. (1998), 'Compositional Petri net structures', *Computer Engineering and Networks Laboratory, ETH Zürich* **60**

Jensen, K. (1992), Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 1: Basic Concepts, EATCS Monographs in Computer Science, Springer-Verlag.

Lakos, C. A. (1994), Object Petri Nets - Definition and Relationship to Coloured Nets, *Computer Science Department, University of Tasmania* **TR94-3**

Lakos, C. A. (1995), From Coloured Petri Nets to Object Petri Nets, *in* 'Proceedings of the 15th International Conference on the Application and Theory of Petri Nets', Vol. 815, Lecture Notes in Computer Science, Springer-Verlag.

Lakos, C. A. (1996), The Consistent Use of Names and Polymorphism to Achieve an Elegant Definition of Object Petri Nets, *in* 'Proceedings of the 17th International Conference on the Application and Theory of Petri Nets', Vol. 1091, pp. 380-399, Lecture Notes in Computer Science, Springer-Verlag.

Lakos, C. A. (1997), On the Abstraction of Coloured Petri Nets, *in* 'Proceedings of the 18th International Conference on the Application and Theory of Petri Nets', Vol. 1248, pp. 42-61, Lecture Notes in Computer Science, Springer-Verlag.

Meyer, B. (1988), 'Object-oriented Software Construction', Prentice-Hall.

Moses (1999-2001), The Moses Project, Computer Engineering and Communications Laboratory, ETH Zurich. (*http* : $//www.tik.ee.ethz.ch/ \sim moses$)

Naedele, M. &Janneck, J. W. (1998), Design Patterns in Petri Net System Modeling, *in* 'Proceedings ICECCS'98', pp. 47-54

ITU-T (1996), 'Video coding for low bit rate communication', ITU-T Recommendation H.263.

Valk, R. (1996), 'On Processes of Object Petri Nets', *Fachbereich Informatik, Universität Hamburg* **185**

Vogler, W. (1987), Behaviour preserving refinements of Petri nets, *in* 'Graph-Theoretic Concepts in Computer Science', Vol. 246, pp, 82-93, Lecture Notes in Computer Science, Springer-Verlag.