Modeling hierarchical and recursive structures using parametric Petri nets

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

KEYWORDS: *Petri nets, components, parameterization, dynamic structures*

ABSTRACT

In order to facilitate modeling of complex hierarchical and dynamically structured concurrent systems this paper proposes extending high-level Petri nets with parametric and dynamic net structures based on a Petri net component concept. It proceeds to show how to apply these techniques to non-trivial design problems while retaining generality and thus reusability of the resulting models. Furthermore, these structuring mechanisms facilitate the understanding of complex models and their component-based design as well as the application of modern software engineering concepts such as design patterns (Naedele and Janneck 1998) to Petri net based modeling.

1 INTRODUCTION

Designing and modeling complex systems in any discipline is a difficult task – such systems are often highly organized and structured, they consist of a large variety of subsystems interacting in a plethora of diverse communication patterns. Moreover, most engineering disciplines require these systems to be ever more flexible and dynamic and be designed in less time. This last point, together with the fact that the design of such a system itself often represents a substantial investment, calls for an increased reuse of flexible and configurable design elements. In the production of software, this is one of the central motivations for the use of object-oriented techniques and corresponding languages (Meyer 1988) as well as formal approaches to specifying architectures and component infrastructures (Shaw and Garlan 1996).

While Petri nets have a long tradition as a formal description technique for concurrent systems, their original formulation suffered from a lack of constructs for compositionality and abstraction (Jensen 1990) – key elements for reusability. Several approaches have since added expressiveness to Petri nets in order to overcome these problems and make them a useful tool for specifying and designing complex real-world systems: colored Petri nets define tokens to carry data values and provide some concepts of modular and hierarchical net structures (Jensen 1990), (Jensen 1992), several object-oriented extensions added capabilities like inheritance, encapsulation and abstraction (Buchs and Guelfi 1991), (Christensen and Toksvig 1993), (Lakos 1994), (Valk 1996).

In spite of these efforts, many modeling tasks remain difficult using a Petri net approach and efficient reuse of designs is unsatisfactory. We will argue that this is in part due to a lack of a means to specify *parametric* Petri net components, a technique well known from both conventional and object-oriented languages (cf. (Meyer 1988) for a more extensive treatment). In this paper we will show how parameterization combined with a component model differing from traditional approaches of place and transition substitution can be used to efficiently utilize the expressiveness of Petri nets for concurrent systems modeling. Furthermore, we will demonstrate how these concepts may be generalized to a simple and orthogonal construct providing dynamic network structures.

This paper first introduces in section 2 the basic component model and discusses the instantiation of components and their parameterization, which will then be applied to a real-world example. In section 3 we will generalize these concepts to include dynamic network structures and demonstrate their usefulness by improving our design of the example in the previous section. Section 4 concludes with a summary and a discussion of topics for further work.

2 STATIC PARAMETRIC STRUCTURES

This section will introduce our concept of components and their composition. For a formal treatment of this subject see (Janneck 1998). We assume a suitable high-level Petri net base-formalism to underly our structuring constructs. Though these are orthogonal to the kind of inscriptions used, we will in the following use inscriptions in the manner defined for Colored Petri Nets (Jensen 1992), though we will not use color sets.

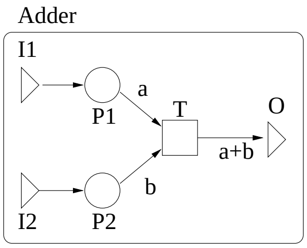

Figure 1: A very simple component template

2.1 Components and their composition

Components are a key concept in modern software architecture and software engineering (Shaw and Garlan 1996). Formally, they may be regarded as stream processing functions, taking streams as input and producing streams as output – see (Broy 1995; Broy 1996) for a formal definition.

The approach to composition suggested in this work is based on this notion of component and on the structuring concept for high-level Petri nets that was developed in (Esser 1996). Instead of trying to directly refine either places or transitions, we will consider components to be subnets with inputs and outputs, contained in other components and connected to their environment by input and output *connectors*.

For example, Fig. 1 shows the definition of a component template[1] *Adder* that has two input connectors and one output connector 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 connector. Also, connectors are named: each *Adder* component has two input connectors named $I1$ and $I2$, respectively, and one output connector O.

[1]We will address the relation of components to component templates below.

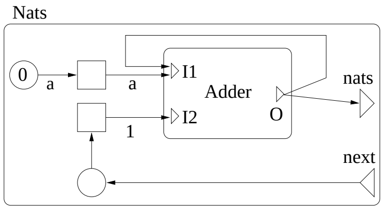

Figure 2: A component template embedding a component.

In Fig. 2 we see the use of an *Adder* in the definition of another component template, *Nats*, generating the sequence of natural numbers. It has an input connector *next* through which the next number may be requested and an output connector *nats*, at which the natural numbers are delivered.

We can therefore define static component templates formally as follows:

Definition 1 (Static component template.) *A* static component template T *is a tuple*

$$(P, T, I, O, C, A, Init, G, E, Par)$$

where P, T, I, O, and C are mutually disjoint sets of places, transitions, inputs, outputs, and embedded components, respectively. We assume two disjoint sets $\mathcal{I}$ and $\mathcal{O}$ of all input and output identifiers, with $I \subset \mathcal{I}$ and $O \subset \mathcal{O}$.

We define the set $\overline{P} = P \cup O \cup (C \times \mathcal{I})$ and $\overline{T} = T \cup I \cup (C \times \mathcal{O})$ as the sets of P-objects *and* T-objects, *respectively. Then the set A of arcs is*

$$A \subset \overline{T} \times \overline{P} \cup P \times T$$

Assuming a set $\mathcal{E}$ of all expressions,

$$Init : P \cup C \to \mathcal{E}$$

maps places and components to their initialization/instantiation expressions (more on the latter in section 2.2). $Expr : A_T \to \mathcal{E}$ maps all arcs starting from/ending at a transition (the set A_T) to an expression, and $G : T \to \mathcal{E}$ maps transitions to their guard expressions.

Finally, $Par \subset \mathcal{V}$ is a set of instantiation parameters *from the set $\mathcal{V}$ of variable names (more on this in section 2.3).*

2.2 Instantiation

As stated above, models like the Figs. 1-5 are templates for actual components. Given such a template

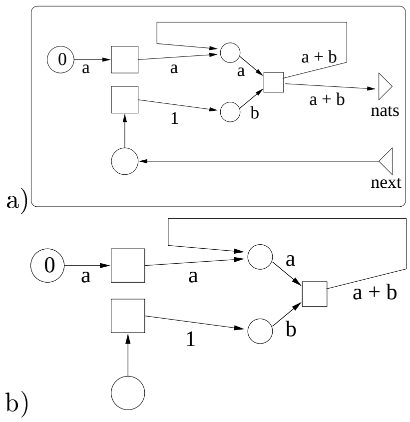

Figure 3: A flat natural number generator.

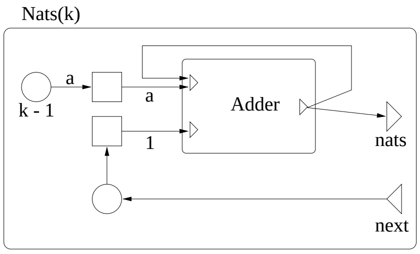

Figure 4: A parametric natural number generator.

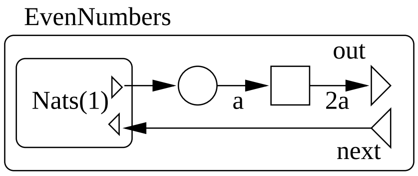

Figure 5: Embedding a parametric component template.

and values for all its parameters, one can construct a concrete component and subsequently a Petri net in a process called *instantiation*.

In the example in Fig. 2, the *Adder* component is instantiated by evaluating the term *Adder*, its *instantiation expression*. Such an expression may consist only of the template name (such as *Adder*), but in general it may be of arbitrary complexity. This expression returns a new component of the structure shown in the *Adder* template, with all initialization expressions of its places evaluated (0 for the top left place) and the results put into the places as its initial marking.

The new component can be thought of essentially as a copy of the *Adder* structure. In particular, each component has its own nodes (P, T, I, O, and C), and therefore its own state, independent of the states of other components.

This copy can now be thought of to be 'pasted' into its surrounding component, resulting in a *(fully) substituted component*, defined as one where $C = \emptyset$, such as in Fig. 3a. This replaces all arcs beginning or ending at interfaces with arcs going from or to the transitions/places inside the embedded component that are connected to those interfaces.

If a substituted component is part of an even larger system, it is in turn pasted into its surrounding component and so on. Eventually, the top-level component is simply turned into a Petri net (which is a component such that $I = O = C = \emptyset$ and $Par = \emptyset$) by removing all its connectors and the corresponding arcs (there being nothing on the outside they could be connected to). The result is shown in Fig. 3b.

This high-level Petri net can now be treated by the standard procedures for execution and analysis.

2.3 Instantiation parameters

Component templates such as the ones above are still very inflexible and cumbersome to use. *Instantiation parameters* allow us to configure component behavior by providing different actual values for them at the different points where a component is embedded.

Suppose that we wish to define the $Nats$ component template in Fig. 2 in such a way as to make the starting value of the sequence variable (instead of a constant 1). In this case we have to introduce a parameter as shown in Fig. 4.

Note that the parameter k appears in the initialization expression of the top left place, resulting in one token with the value $k - 1$. Using such a parametric component template in the definition of another one might look like Fig. 5.

2.4 Example: a Huffman decoder

Let us now apply these concepts of parametric components and their instantiation to a concrete modeling problem, namely that of building a Huffman decoder for a given Huffman tree.[2] We will assume the tree to be given, and indeed for the moment we will assume it to be constant – though for the sake of reusability we will strive to minimize the points where we actually refer to this assumption.

Huffman decoding converts a stream of bits into

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

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 and starts the procedure at the root again. Otherwise it simply continues with the subtree it went to in this step.

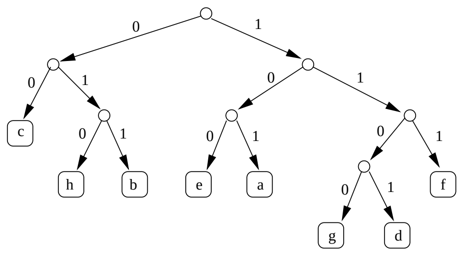

Figure 6: A Huffman tree.

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 provides the following mapping between bit sequences and the characters from a to h:

character	bit sequence
a	101
b	011
c	00
d	1101
e	100
f	111
g	1100
h	010

In the following we assume the tree to be represented by a data structure such that predicates $isNode(t)$ and $isLeaf(t)$ are defined to be true iff t is a node or a leaf, respectively, that for any node t $t.left$ and $t.right$ are the left and right subtree, and that for any leaf t, the character at that leaf is $t.c$.

One compositional solution to this problem makes use of instantiation expressions and embedded components. We compute for a given Huffman tree a tree-specific component by taking the first bit and directing the remaining bits to subcomponents that represent decoders for respective subtrees – until a complete character has been recognized. Thus the construction of such a component is obviously recursive, since the component itself contains subcomponents that are constructed as decoders for the subtrees.

Clearly, for a given tree the corresponding net could be produced by hand, which would be tedious and error-prone for any but the smallest applications. We are looking for a component template that exhibits the recursive construction and allows us to automatically generate a component for any tree.

First we will consider the task of computing *one* output character from the input stream. The interface of a component for this purpose is shown in Fig. 7. The `in` connector accepts the incoming bits, after the component has been sent a token on the `select` connector. Each input is acknowledged by an output token at the `ack` connector. If a complete code has been received, the corresponding character is output at the `out` connector. For an unambiguous decoding, we will also assume the dynamic constraint that the character is sent *not after* the acknowledgment of the last bit of its code.

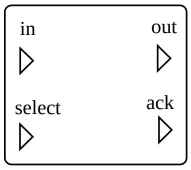

Figure 7: The Huffman decoder interface.

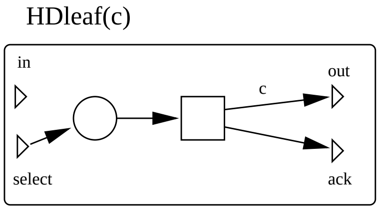

Figure 8: Decoder representing a leaf in the Huffman tree.

We will provide two component templates with 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 shown in Fig. 8. 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 9 is more complex. It contains two embed-

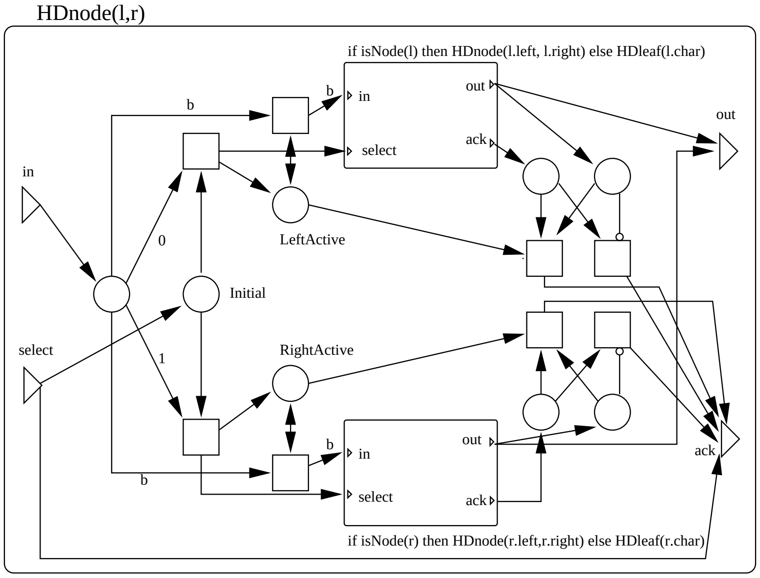

Figure 9: Decoder representing a node in the Huffman tree.

ded components representing its right and left subtree and 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.

Finally, Fig. 10 shows how this component is wrapped to produce a sequence of outputs for an arbitrary sequence of input bits, using a simple handshake protocol.

This provides us with a closed, formal, and complete description of a Huffman decoder net for any Huffman tree.

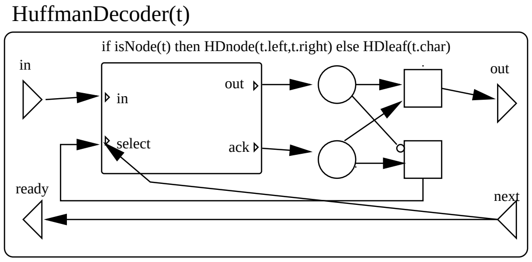

Figure 10: Wrapping a Huffman decoder.

3 DYNAMIC STRUCTURES

Although the component concept introduced in the last section facilitates a certain degree of flexibility in constructing systems, the systems themselves are still static, i.e. their network structure does not change once they are instantiated. Many real-world systems, however, are of a much more dynamic nature, which can only be adequately modeled in a formalism supporting dynamic structures.

3.1 Dynamically contained components

A common approach for moving embedded components around in a net is by making them, or rather their *designators*, tokens that reside on places (Lakos 1994; Lakos 1995; Esser 1996). We adopt this idea, and define a way in which those components are hooked up to their environment.

Designators are computed as results of the instantiation expressions in the last section, i.e. expressions such as $Nats(3)$ or *Adder* have the side effect of initializing a component the designator of which is their value. When such an expression is the initialization expression of a place, it results in a component designator residing on the place. In order to connect the component to the surrounding net, we introduce arcs that begin or end not simply at the place itself, but at a *connector* on the place. Thus we replace the P/T-objects $C\times\mathcal{I}$ and $C\times\mathcal{O}$ in Def. 1 with $P\times\mathcal{I}$ and $P\times\mathcal{O}$, respectively. Fig. 11 shows the $Nats$ component template from Fig. 2 with the embedded component being the value of the central place (such a place with connectors containing components will be called *container place* in the following). Note that now *Adder* is simply the initialization expression of that place.

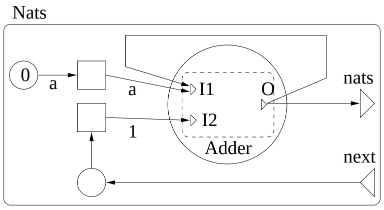

Figure 11: An isolated container place.

Apart from arcs starting or ending at connectors, there is in this case no arc to or from the place itself, i.e. it is isolated, and its marking will never change. Static embedding as in Fig. 2 can then be seen as a notational shorthand for isolated container places. Pasting the contained component from into Fig. 11 (according to the rules outlined in the previous section) results in the substituted component in Fig. 12, which is identical to the substitution in Fig. 3a except for the isolated container place.

Dropping embedded components in favor container places thus gives us the following more general definition of component template.

Definition 2 (Component template.) *A component template T is a tuple*

$$(P, T, I, O, A, Init, G, E, Par)$$

with P, T, I, and O as in Def. 1.

We define the set $\overline{P} = P \cup O \cup (P \times \mathcal{I})$ and $\overline{T} = T \cup I \cup (P \times \mathcal{O})$ as the sets of P-objects *and* T-objects*, respectively. Then the set A of arcs is*

$$A \subset \overline{T} \times \overline{P} \cup P \times T$$

Now

$$Init : P \to \mathcal{E}$$

only maps places to their initialization expressions.

Expr, G, and Par remain as defined in Def. 1.

Of course, we must now phrase the substitution rule slightly more generally: Since any number of component designators (including zero) may reside on a container place, instead of pasting the structure of exactly one embedded component into the surrounding

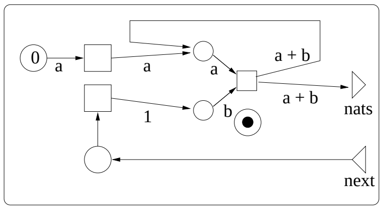

Figure 12: Substituting dynamically contained components.

component and hooking it up at the appropriate connectors, we have to perform this process for every token on a container place that designates a component.

In contrast to Fig. 11, Fig. 13 shows a 'truly' dynamic component template, since its container place is not isolated and therefore its marking may change.

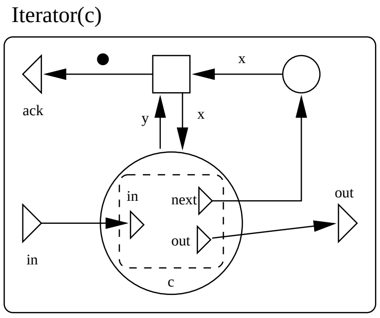

Figure 13: A dynamic component template.

Dynamic components such as (instances of) Iterator in Fig. 13 cannot be executed by translating them into a Petri net only once at the beginning. Consider the dynamics of such a component: Inputs at connector `in` are processed by the component on the container place, which produces outputs at `out` and values that are put on the top-right place. When the transition fires, the original embedded component is removed from the container place and the value on the top-right place is put on the container place instead.

Whatever Petri net structure we had computed before will probably not reflect the current state of the system. In general, we will have to recompute a Petri each time before we make a step, because any state change (affecting a container place) might result in a change of the structure of the resulting Petri net.

3.2 Example: the Huffman decoder revisited

While the decoder in Figs. 8-10 is completely general, it is not very efficient in terms of the number of transition firings per input bit. The reason is that each input bit is channeled through all activated sub-components from the root onward, until it either activates the next sub-component or produces an output character. Instead, we would like to pass an input bit directly to the component that is responsible for its processing. This means we have to dynamically exchange the component that receives the input.

Fig. 14 shows the elementary Huffman decoder cell for this purpose. It has three parameters, being the complete Huffman tree t, and the left and right subtree l and r, respectively, that are to be traversed depending on the next input. It has the input connector and two output connectors: one at which the resulting characters are output, and one, to which the Huffman decoder cell for the next step is produced.

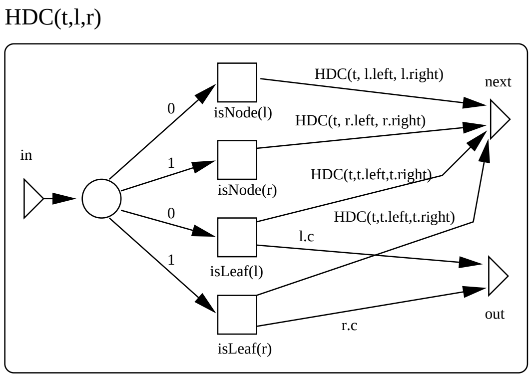

Figure 14: A step in an iteration.

If the next input bit leads to a node, the cell produces only a next cell with the two subtrees of the node as its l and r parameters. If the next input bit leads to a leaf, the corresponding character is output as well as a component with the left and right subtrees of the root node as its l and r parameters.

Now we can create a Huffman decoder for a given tree t simply by instantiating

$$Iterator(\mathrm{HDC}(t, t.left, t.right))$$

4 CONCLUSION

In this paper we suggested two kinds of extensions to high-level Petri nets: a concept of parametric component classes which was subsequently generalized by a construction facilitating dynamic network structures.

In the static case, any model using components may be straightforwardly transformed into an equivalent flat high-level Petri net. Therefore, while adding much to the expressiveness of the modeling formalism, from a theoretical point of view our component model adds only syntactic sugar to those static Petri nets.

This, however, changes for dynamic network structures. Our second extension greatly enhances the expressiveness of classical statically structured Petri nets. Still, the semantics largely rests on the semantics of standard high-level Petri nets in much the same way as in the static case, which in fact turns out to be a special case of dynamic nerworks, and one that can be easily identified from the structure of the component templates involved.

Starting from here, research is needed in a variety of areas:

- Modeling environments have to be built to support these constructions during the various activities of interest in systems modeling: design, simulation, analysis/verification, debugging, evaluation/validation, possibly implementation. This is done by the authors in the context of the Moses project.
- While strictly not necessary, practical modeling would greatly profit from a type system. This would allow checks for correct connectivity and expression typing to be performed statically. The Moses tool supports an object-oriented type system for this purpose.
- Analysis methods should be developed to take advantage of the component structures. Though concrete static models may be analyzed by transforming them into Petri nets and applying traditional analysis methods to them, it may not necessarily be the best thing to do, and it does not help for dynamic networks. The structure of the system may contain valuable information, in particular if it is labeled with descriptions of high-level properties or invariants for components.
- Libraries of reusable Petri net components would have to be built to allow prospective designers to concentrate on their application domain.
- Integrating other formal or semi-formal specification languages with this approach – both, in terms of a semantic description as well as in tools supporting this – might be another step to enhance its utility and acceptance in practice.

This last point deserves special attention: Since the interfaces of components are defined in a very general way that does not really require Petri nets as their implementation formalism, one could embed components specified in other description techniques (such as finite state machines, data flow graphs or indeed any formulation of a discrete event system) in a transparent and orthogonal manner.

References

Broy, M. (1995). Advanced component interface specification. In T. Ito and A. Yonezawa (Eds.), *International Workshop on Theory and Practice of Parallel Programming*. Springer-Verlag.

Broy, M. (1996). Towards a mathematical model of a component and its use. In *Proceedings Componentware Users Conference*. SIGS Publications.

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

Christensen, S. and J. Toksvig (1993). DesignBeta v2.0.1 - BETA code segments in CP-nets. Lecture Notes OO&CPN No. 5, Computer Science Department, Aarhus University.

Esser, R. (1996). *An Object Oriented Petri Net Approach to Embedded System Design.* Ph. D. thesis, ETH Zurich.

ITU-T (1996, March). Video coding for low bit rate communication. ITU-T Recommendation H.263.

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

Jensen, K. (1990). Coloured Petri nets: A high level language for system design and analysis. In G. Rozenberg (Ed.), *Advances in Petri Nets*, Volume 483 of *Lecture Notes of Computer Science.*

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

Lakos, C. A. (1994). Object Petri nets - definition and relationship to coloured nets. Technical Report TR94-3, Computer Science Department, University of Tasmania.

Lakos, C. A. (1995). 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.

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

Naedele, M. and J. W. Janneck (1998). Design patterns in Petri net system modeling. In *Proceedings ICECCS'98*, pp. 47–54.

Shaw, M. and D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall.

Valk, R. (1996). On processes of object Petri nets. Technical Report 185, Fachbereich Informatik, Universität Hamburg.

BIOGRAPHIES

Jörn W. Janneck graduated in computer science from the University of Bremen, Germany, in 1995, working on computer graphics and artificial intelligence.

From 1995 to 1996 he worked as a researcher for the Fraunhofer Institute for Material Flow and Logistics (IML) in Dortmund, Germany, in the field in business modeling and simulation. Since 1996 he is holding a position as a research assistant at the Swiss Federal Institute of Technology (ETH), Zurich, Switzerland. He is lecturing on OO Software Engineering at the Polytechnical College (HTA) Chur. His research interests include software architecture, visual language semantics and various aspects of discrete event systems including simulation, formal analysis, and modeling languages such as Petri nets.

Martin Naedele studied electrical engineering at Ruhr-University, Bochum, Germany, and Purdue University, West Lafayette, IN. He received his MS degree in electrical engineering from Ruhr-University in 1997. Currently, he is working at the Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, as PhD student and research assistant in the Computer Engineering and Networks Lab (TIK). His research interests include embedded systems, fault-tolerant real-time computing, software engineering, and computer security.