Generation of Executable Object-based Petri Net Skeletons Using Design/CPN

Daniel Moldt and Heiko Rölke

University of Hamburg, Dept. for Computer Science, Vogt-Kölln-Str. 30, D-22527 Hamburg, {moldt, 3roelke}@informatik.uni-hamburg.de

Abstract. Object-oriented Petri nets have gained a lot of interest in the last years. We used the tool Design/CPN in several projects, especially for Object-Oriented Coloured Petri Nets (OOCPN). However, Design/CPN does not support OOCPN directly. To allow easier modelling we developed a net generator (GPS = Generator for Petri net Systems). GPS works on class diagrams like those presented in the Unified Modelling Language (UML). The output of GPS are OOCPN, where each class and its objects are represented in one separate specific Coloured Petri Net (CPN). The generated nets are directly executable in Design/CPN as far as creation and deletion of objects are concerned. Methods and variables are prepared as well, but the user has to give some meaning to the method skeletons. Associations are represented as classes.

Keywords: Coloured Petri Nets, Design/CPN, Net Skeletons, Net Generation, Object-Orientation, Prototyping, Computer Tools

1 Introduction

In our group the integration of object-orientation and Petri nets is an ongoing topic (see [BM93], [Mai96], [Mol96], [Val91], [Val98]). One major goal is to allow for the use of Petri nets as the means to execute object-oriented specifications. To achieve this a transformation of the models to Petri nets is necessary. In [Mol96] for several techniques transformation schemes were proposed, but no tool was presented at that time. In this paper we will concentrate on the central technique, the class diagrams. For the class diagrams again only the basic kinds of associations and some tool support by Design/CPN are considered. Another major goal of the transformation is to get a deeper understanding of static models and to provide better semantics to techniques like class diagrams.

Class diagrams exist in several variants. The most important one is that of the Unified Modeling Language (UML) (see [BRJ99,JBR99,RJB99]). This version of class diagrams combines most of the features that have been developed for static relationship modelling so far.

Our planned specification environment aims at the support of several techniques. This includes the integration of the different models. Since object orientation is the central paradigm, the starting point for the integration are the classes (or objects). In [Mol96] three main views on a system are discussed: the static view, the dynamic view, and the functional view. All these views are modelled for an object and can then directly be integrated after they are transformed

into Coloured Petri Nets. On this basis only one formalism is given and hence the integration is easier. Modellers can present an executable specification to users. However, this requires some additional input for the improvement of the user interface. Otherwise the user has to interpret the net simulation directly.

At the current state only fragments have been supported by tools. Due to the fact that tools are a precondition for the successful application of OOCPN, we are working on these tools. In this paper we will show for the central concept of classes (and objects) how to transform class models to OOCPN class models. The basic assumption is that even for the static models (class diagrams) an operational semantics has to be provided. Each relation can e.g. be classified as being mandatory or optional. Mandatory relations cause some actions in the related classes when in the other classes objects are created or deleted. This cascading of operations is usually not modelled elsewhere in UML. To interrupt this cascading would violate the integrity constraints that are expressed by the relationship. When providing an operational semantics for this feature of the class diagrams, a specific protocol can be added to other "standard" protocols for methods, attributes etc. All other kinds of attributes of relations require also some kind of protocol. These protocols may differ considerably.[1]

It is important to notice that we do not present a generator of Petri nets for class diagrams in general including all features and their implicit protocols[2]. So far we have concentrated on to provide the basic infrastructure, which means to generate one Hierarchical Coloured Petri Net (HCPN) for one class. Each usual relationship (association) is also considered to be a class. This allows to assign methods and attributes to associations. The related protocols of the behaviour e.g. to a `(1:N)` relationship are not automatically generated. As many other features these could be integrated on demand. However, this would require some specific programming for which we had no time resources up to now. It is interesting to see that Entity-Relationship-Diagrams (ERD) (see e.g. [You89]) can be handled in the same way as class diagrams.

1.1 Modelling process

Figure 1 shows the steps of how to derive from object-oriented models the Object-Oriented Coloured Petri nets within the Design/CPN tool. In the following description we concentrate on the class diagrams and omit the other techniques. The Petri nets generated for the other techniques would be integrated into the here described models.

The user models the class diagrams and receives by this the executable Petri nets in Design/CPN. The large box represents the overall action. First of all

[1] With protocol we do not mean e.g. TCP/IP, but the rules which determine simple sequences or parallel actions that serve to support the interaction of two or more objects. An example is the instantiation of a second object which needs to be present due to an association between the related classes and the instantiation of an object of one of the classes. This will be explained in the following sections in more detail.

[2] This is the reason why we use object-based instead of object-oriented in the title of this contribution (see also [Weg87])

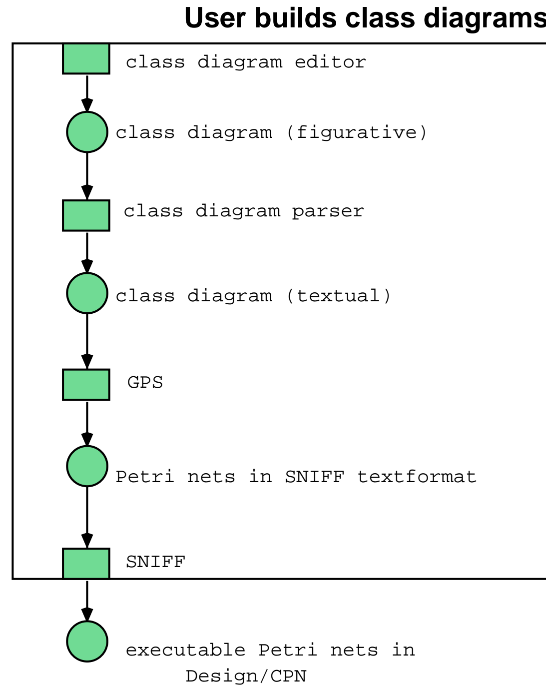

Fig. 1. Steps to derive executable Object-based Petri net skeletons

the user has to draw the class diagrams. There is an ongoing work (diploma thesis) at our department to support the drawing of class diagrams within the Design/CPN environment[3]. Any other tool than Design/CPN can be used as long as its output format can be translated to the class diagram text format that is currently used by the GPS toolset[4]. Its output format are Design/CPN-oriented Coloured Petri nets in the SNIFF text format [MMR98][5]. This text format for Coloured Petri nets can be imported to Design/CPN and can be handled in a way as if the models were drawn by the user.

Except for the drawing of the class diagrams all parts of this procedure are executable within Design/CPN because they are implemented using the Standard ML language [Des93,Pau92]. All functions are written in a way that enables their concatenation: The output of the first function is taken as the input for the next and so on. These function calls can be hidden from the user, who only has to call one function without arguments. The user designs the class diagrams

[3] Remark: One goal is to do all tasks within one environment, here the Design/CPN environment.

[4] This text format is an ad-hoc approach to class diagram text formats. At this point also a standard like IDL (Interface Definition Language) could be used. With the times changing we will surely switch over to a standard notation.

[5] SNIFF is an input/output library for Design/CPN presented at the Design/CPN workshop in Aarhus in 1998. The newest version of Design/CPN has now a tool being build in that allows the im- and export, however, the format is different (see [LM98]).

within one page of Design/CPN and gets the resulting Object-Oriented Coloured Petri nets on just one mouse click. Changes to the underlying diagrams do not require a change of the environment. Therefore, the ML functions acting behind the scene are invisible for the user, they behave like a refined transition of a Petri net. Therefore, the large box in Fig. 1 can also be interpreted as one transition, not showing the required input from the outside in the same way as this is not done for the single transitions (or functions).

Nevertheless all single functions are accessible for the experienced user. This assures maintainability, adaptability, and expandability.

1.2 Why object-based nets?

Petri nets have proven to be a useful formalism to express concurrency. Problems related to concurrent processes and distributed algorithms are getting more and more important to be managed, i.e. solved, even in mainstream computer science. One idea to tackle this situation is to combine the results from different fields of computer science. We have chosen Coloured Petri Nets (CPN) and object orientation (OO) as the main concepts. This allows to combine the structuring facilities of OO with the advantages of a technique with a sound theoretical background. Up to now object-oriented specifications, like those of UML, are hardly executable. With our approach this problem is (partially) overcome. However, the resulting nets are too large to be used directly by a modeller. Therefore, we propose the use of specialised tools and here we concentrate on the GPS. Due to the simple nets, we support up to now, we call the generated nets object-based skeletons. These skeletons have to be completed to contain the functionality. In combination with other tools which cover the modelling of the functionality this could also be automated as well (as far as a generator can help here).

The rest of the paper contains two main parts: Section 2 shortly presents the main relevant concepts of Object-oriented Coloured Petri Nets and section 3 discusses the generator GPS and its implementation. The paper ends with a short conclusion and an outlook for future work.

2 Object-Oriented Coloured Petri Nets

This paper is intended to show the computer aided implementation of Object-Oriented Coloured Petri nets as proposed by Moldt [Mol96]. With respect to this aim other approaches to combine both the advantages of Object-Orientation and Petri nets are not discussed here. For some work of object-orientation and Petri nets see for example [SB94,Lak95,Val98,Mai96,Kum00,BG91,EMNW99,MM99].

Due to limited space the introduction of Petri nets in general and Coloured Petri nets in special is skipped here as well as the presentation of well-known concepts of Object-Orientation and techniquess like UML class diagrams. To become familiar with these concepts see for the study of Petri nets [Pet62,Rei92,Jen92] and for Object-Oriented concepts [Lou93,Fow97,RJB99].

Objects, classes, and methods are the basic components of Object-Oriented programming languages [Lou93]. An object invokes a method (service) of another object by sending it a message. In the following subsections, Moldt's transformation of objects, classes, and a messaging mechanism to Petri net representation will be introduced (see [Mol96]).

2.1 Objects

An object encapsulates its state and behaviour. The only way to change an object's state is through the methods provided at the interface of the object. In an Object-Oriented programming language classes are defined and objects, so-called *instances* of these classes, are created during runtime. Local variables of an object are also called instance variables.

```
class Class_x {
public:
   int method_1 ();
   void method_2 ();
   ...
   int method_n ();
protected
   anytype inst_var;
};
```

Fig. 2. Class definition from Moldt [Mol96]

Transformation of Objects The class definition in figure 2 defines `n` methods[6]. In addition, a local variable `inst_var` is declared which can only be used by the class itself, or by a subclass. An object (instance) of this class, called `OBJECT_X`, is shown as a net in figure 3 and is called Object-Net[7]. Variables are transformed to places, methods to transitions. An object is a marked net page according to Jensen [Jen92][8].

The object receives and sends messages using the two places `in_pool` and `out_pool`. These two places are the interface between the object and the message

[6] The examples are given using a notation similar to C++ without claiming to be syntactically correct in that way.

[7] The net inscriptions are given in the functional language ML as it is used in the Petri net tool Design/CPN. For the sake of simplicity parts of the net inscriptions are omitted to focus on the core concepts. The given nets are therefore not syntactically correct according to Jensen [Jen92] in the sense that arc inscriptions etc. have to be added. Concurrency is not restricted by the model introduced here. Objects can process several messages at the same time.

[8] The marking is normally omitted because the main purpose here is to show the static aspects.

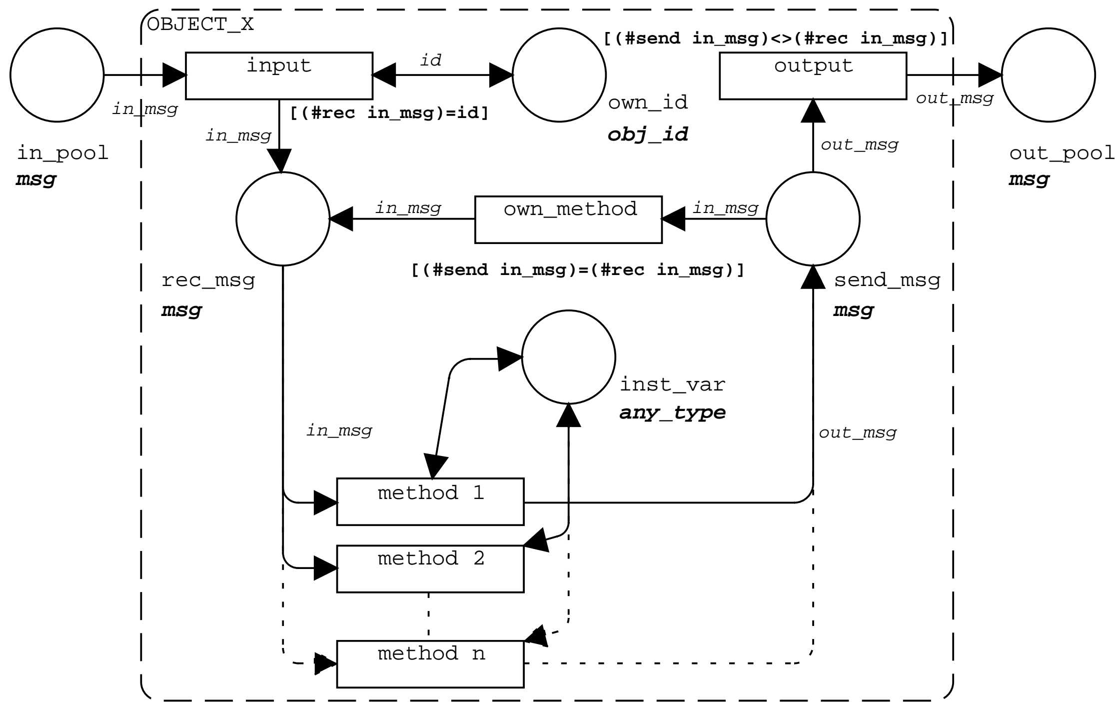

Fig. 3. Object-Net from Moldt [Mol96]

handler. Depending on whether one considers these places as part of the object or not, an object is a place- or transition-bounded net. The message handler is responsible for transporting the messages from `out_pool` to `in_pool` places of objects.

The transition `input` selects the appropriate messages from the `in_pool` using the side condition `own_id`, thus ensuring that only a message for the object reaches the place `rec_msg`. Because a method can only be invoked by a message in the place `rec_msg`, the presented Object-Net realises an encapsulation of the object's state and behaviour.

A method selects its appropriate message using a guard that refers to the method name in the message. The functionality of the method can be given as a net refinement or a code segment as being used in the Petri net tool Design/CPN. After computation a reply message is put into the place `send_msg` that is sent to `out_pool` by the transition `output` or, if the receiver of the message is the object itself, is put to the place `rec_msg` by the transition `own_method`.

One can look at an Object-Net as consisting of two parts. The transitions `input`, `output`, `own_method` and the places `rec_msg`, `send_msg` are common to all objects and can be used as a template for objects. The specific state and behaviour of an object is being realised through the places for instance variables and the transitions for methods.

Object Identity Beside its state and behaviour, an object has a clear identity, i.e. a unique key that is used as a reference. In this proposal a unique key

called Object-ID is used. An Object-ID denotes exactly one Object-Net and is generated during the creation of an object. To get a unique key the following convention is used. The Object-ID consists of the class name and an atomic expression locally unique for this class, usually a number of type *integer*. The class name is assumed to be unique inside the system.

2.2 Classes

A class defines the internal state and the behaviour of its objects and therefore their type. One can look at classes as having a construction plan to manufacture objects. Sometimes they are therefore called "Factory Objects". In addition to defining its objects' type, a class provides operations to create, destroy and manage them. A class can receive and send messages and can therefore be looked at as an object, too.

The creation and deletion of new objects, the refinement of methods, and the concept of relationships will be discussed in more detail in Section 3 but not in this subsection.

Folding Objects to Classes Corresponding to the type definition of variables, objects are declared belonging to a class, assumed the language is typed. Therefore we need a mechanism for nets to define classes and later create and manage objects of that class. This is done by folding Object-Nets with the same state and behaviour[9] to so-called Class-Nets (Figure 4). The type of each place is extended with the Object-ID. A place in the Object-Net with type `any_type` now gets the new type `obj_id*any_type` in the Class-Net. The places `own_id` in the Object-Nets are folded to the place `all_Inst` (all instances) in the Class-Net. The place `class_id` in the Class-Net holds the Object-ID (the name) of the class. The type of the variables used in the arc expressions is extended in the same way. Thus it is ensured that the Class-Net has the same behaviour as the folded Object-Nets before.

In addition, the Class-Net is enriched with class operations like `new`. It is therefore capable of defining the type of its objects and to create and manage them. In a Class-Net, the tuple `obj_id*any_type` is called `inst_type` for the sake of simplicity, `obj_id*msg` is called `msg`. A Class-Net gets the reserved Object-ID `(<classname>, 0)`, called Class-ID. Using this Class-ID one can send a message to a Class-Net, e.g. to invoke the creation of a new object. The way to create or discard objects or to perform other class operations is based on the idea that classes are objects. A class contains method transitions that describe the desired action to be invoked by calling the method.

[9] Again we refer to the static aspects of state and behaviour: All folded Object-Nets must have the same places and transitions but may differ in their markings at runtime.

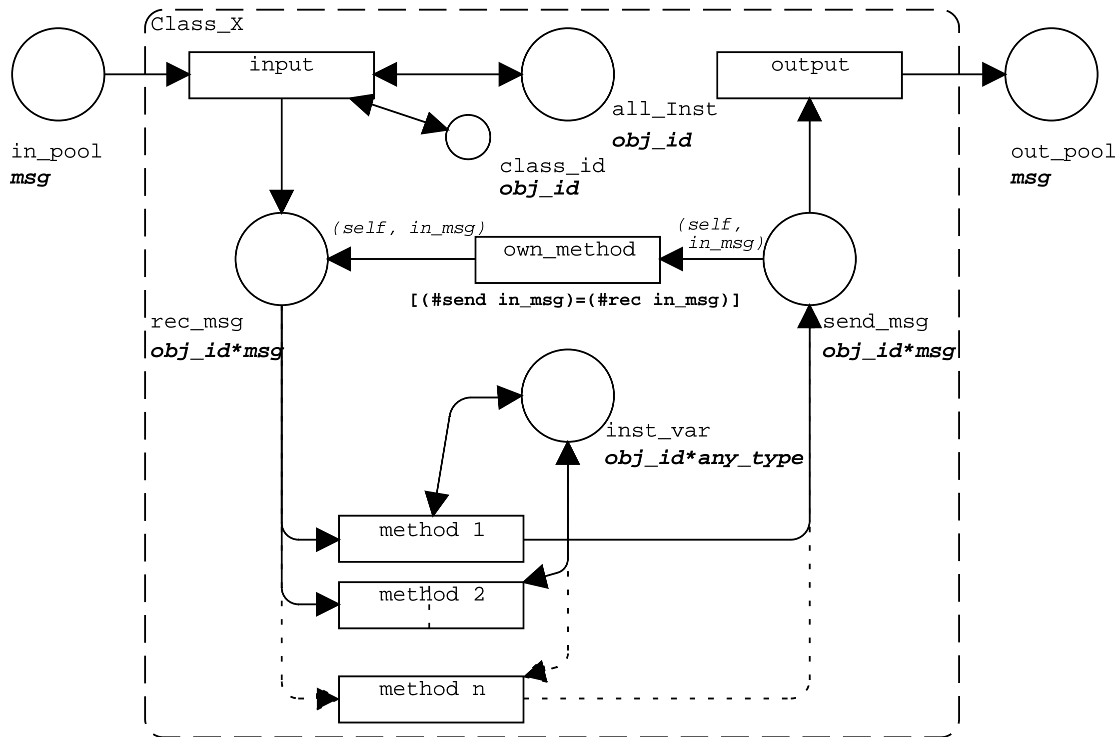

Fig. 4. Folding Object-Nets to Class-Nets

2.3 Messages

Messages are used to invoke a method in the destination object. In this approach messages are realised using tokens that are exchanged between objects. An error-free messaging mechanism is assumed. The message format has to cover information about sender and receiver in term of class, object, and method as well as parameters which have to be passed around. Depending on the kind of message system different message formats are required. In this paper we use a straightforward one which explicitly represents all information directly. Consequently the message parsing has to be done within the net. One could argue that this could be done within the message handler and could therefore be hidden. This is true, however, in [Mol96] the principle structure should be visible and for convenience we have also chosen this format for our generator. In further work we will also support other message formats.

The Message-Handler A message-handler is used to route the messages between objects. One way to realise this message-handler is by simply fusing all the in_- and out_pool places using place fusion. In a distributed environment a more sophisticated version should be used to model different routing strategies or incorporate possible losses of messages. The concept of a message-handler has therefore been included to provide the possibilities to model these relevant aspects of distributed systems.

3 Design

The developer of an environment for computer aided design of Object-Oriented Coloured Petri nets has to take care of certain procedures like the creation of new objects. The traps like the finding of a unique identifier for the new object should be handled by the system in order to aid the user. Object-Oriented Coloured Petri nets offer some features from the Object-Oriented Analysis like associations. These features lead to even more complex protocols that sometimes include the interplay of several different classes. To release the modeller from the burden of explicitly modelling we propose to generate these protocols from more abstract diagrams. The case of an automatic generation of association skeletons without the protocols for the specific behaviour will be handled during this chapter. All Petri nets illustrating the following pages are shown as they are (automatically) generated by the GPS tool set. They are not adjusted in any way.

3.1 Association protocols

A protocol to handle relations between objects serves on the one hand to disburden the modeller of an Object-Oriented system. The protocol is needed to automatically handle a relation and to move it to the desired code. On the other hand a protocol gives a clear meaning to the constructs of a diagram. In larger projects the class diagrams are cut into pieces and given to different programmers. Every programmer has its own view on the system and on how to handle e.g. the relations. This can lead to misinterpretations followed by mistakes. The code of one programmer is not understandable by another.

In order to reach this aim it is necessary to have a clear understanding of relations between classes. Usually relations are categorised being "simple" or "complex". A simple relation should be handled by the related classes itself while a complex relation needs a so-called "relation class" to be implemented. Again it is not clear how to formalise such a distinction. Due to this reason the presented system generates an new class for each relation. This is obviously not necessary for very simple cases like a one-to-one relation without attributes or methods but the computer does not complain about the extra work. The disadvantages concerning performance, however, are obvious. Therefore, we plan to provide protocols for "simple" relations such that no separate relation has to be build. It might even be possible to integrate two classes that have a one-to-one relation. However, this has not been done up to now.

The set of possible relations between objects or classes divides into two subsets with different properties: Mandatory and optional relations. This distinction is motivated by giving example procedures. The principle situation is as follows: Two classes have a certain association between them. This association has a certain multiplicity for each of the objects. It has for each of them the information if the relation between objects of this class is mandatory or optional. If for one of the classes an abject is created the association now implies a certain behaviour (protocol) at the other side of the association. This is discussed in the following.

Mandatory relations An object of a class in a mandatory relation shall be created. We first consider the case of an one-to-one relation, that has to be distinguished from other multiplicities like one-to-many or many-to-many. In the first case exactly one corresponding counterpart of the newly created object and an object of the relation class have to be created. In the latter case the new object may be related to one or more existing objects of the relation class. The relation class itself has to handle the co-ordination between the objects, which can be toilsome especially when the multiplicities of the relations have restricted number areas. The choice of the right relation classes for a newly created object can not be done automatically. The user has to be prompted for that choice or the diagrams have to be extended to cover some information about this issue. This is true in general. However, depending on certain conventions, project specific rules, application specific requirements etc. there are some cases where this could be automated. In the normal case, however, the developer has to decide what has to be done. This process again can be done at different points of time and depends heavily on the available tools. One idea this paper wants to present is that there is a problem and that there are certain ways to solve them. Here only the main stream is discussed. For an efficient and practical application tools are a prerequisite.

The erasure of an object in an one-to-one relation enforces the erasure of the related object and the object of the relation class. The erasure of related objects can lead to troublesome situations[10]. The situation of having some states when the whole system is inconsistent is one of the main reasons to use protocols. Only if a protocol finishes successfully the operation is really performed. Otherwise the system has to reset the already performed actions. Exactly this point makes the protocols so important and so difficult to describe in general. A general simple solution is not available. However, with the appropriate tools the situation becomes more relaxed. It should not be forgotten that in the area of databases this problem has largely been solved. E.g. the automatic generation of database schemes shows how to cope with this kind of problem. For the area of analysis and in specific for Petri nets there are still some open questions.

Optional relations If a relation between objects is not mandatory things are a little bit easier to handle. If it is not necessary for an object to have a related counterpart our environment does not have to take care about the relation class at all. But such a procedure might not be an adequate aid for the user of our system. So the user can be prompted if he is willing to put a new object in some relation or not.

3.2 Class net frontpages

This subsection contains an example of a generated Object-Oriented Petri net. Before we introduce the example with a one-to-many multiplicity, we introduce the example with the artificial one-to-one relationship. The company with only

[10] See for example [Mol96].

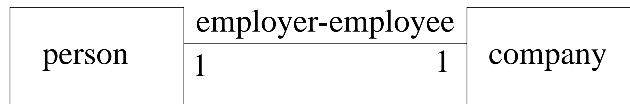

Fig. 5. Simple class diagram with relationship

one employee is very small here and gets larger in the one-to-many case. The example consists of two classes, *company* and *person*, standing in relation employer-employee (see Figure 5). The first trivial example will also be extended by payment as an attribute and by a method to tell the payment for a special instance of the relationship.

A textual description[11] of this relationship would be as follows:

```
interface Person {
   associates
      company 1-1 employer-employee
}

interface company {
   associates
      person 1-1 employer-employee
}
```

To make the example more interesting we enhance the class *company* with an attribute, the company's name and two methods to get and set this name. Figure 6 shows the Petri net frontpage (or rootpage) of this class. The extended textual description is:

```
interface company {
   associates
      person 1-1 employer-employee
   attributes
      string companyName
   methods
      string getName(void);
      bool setName(string);
}
```

[11] See appendix A for the syntax of this description language.

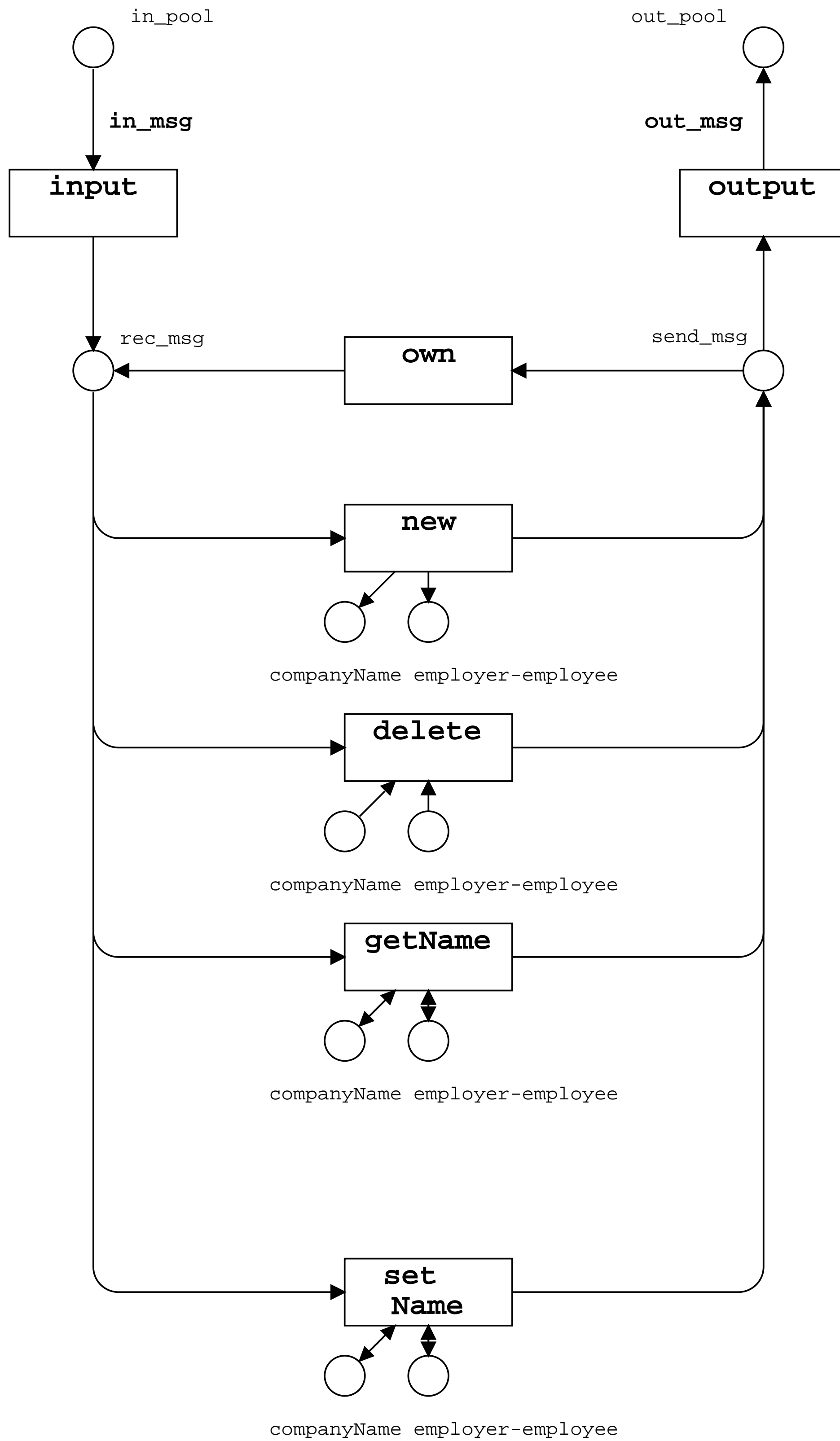

Fig. 6. Frontpage of Class-Net *company*

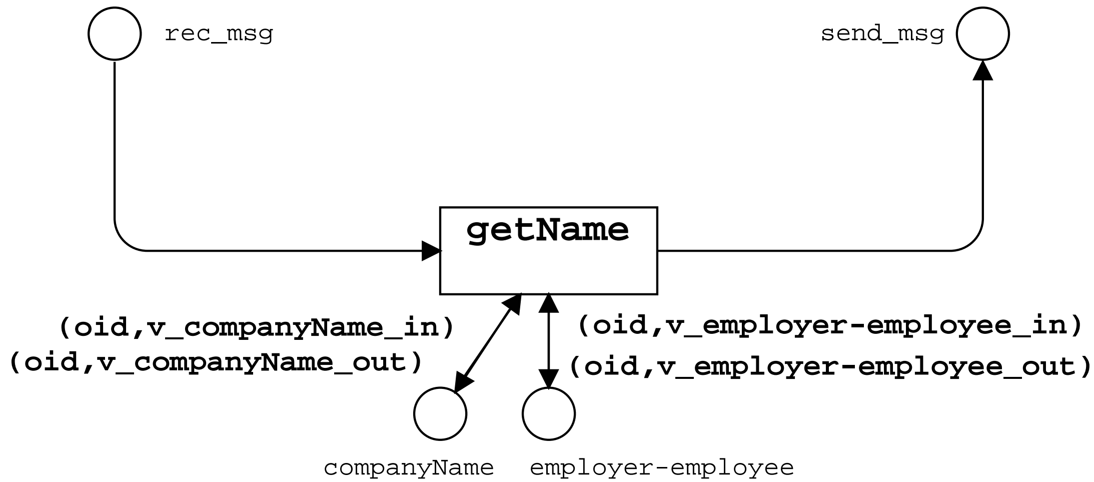

Fig. 7. Subpage of method *getName*

All methods can access the two attribute places. Place *companyName* results from the attribute of the class description above. Note that all places with the same name are connected via the concept of place fusion available in Design/CPN[12]. These places behave like one physical place and always contain the same marking. The attribute place *employer-employee* corresponds to the relationship between the classes company and person. Its name is that of the relation and its marking contains the identifier of the relation class instance that belongs to the actual object. Despite of the few visible inscriptions this net is fully executable in the sense that one can call the class method *new* to create new instances of that class and operate with them at will.

The implementation of the method functionality is generally handled using subpages, the hierarchy mechanism of Design/CPN. This assures a clear frontpage even in complex classes with many methods. Notice that the arc orientation reflects the method's functionality: The method *new* puts new markings to the attribute places, *delete* takes them away. As there is no information on the internals of the other methods they generally get two-way arcs as a default.

3.3 Method subpages

Figure 7 shows the subpage to the method *getName* as it is generated by GPS. On subpages the arcs that access the attribute places are labelled. They get their own names depending on reading (_out) or writing (_in) access. The transition that holds the name of the refined method offers an easy way to implement the methods functionality. It is attached with an predefined code region that is not mentioned in figure 7. This code region imports all attributes that are connected

[12] The tool GPS automatically hides the detailed inscriptions that are necessary in Design/CPN to get things like place fusion going. If the user wants full access to the generated nets he can easily adjust GPS not to hide anything.

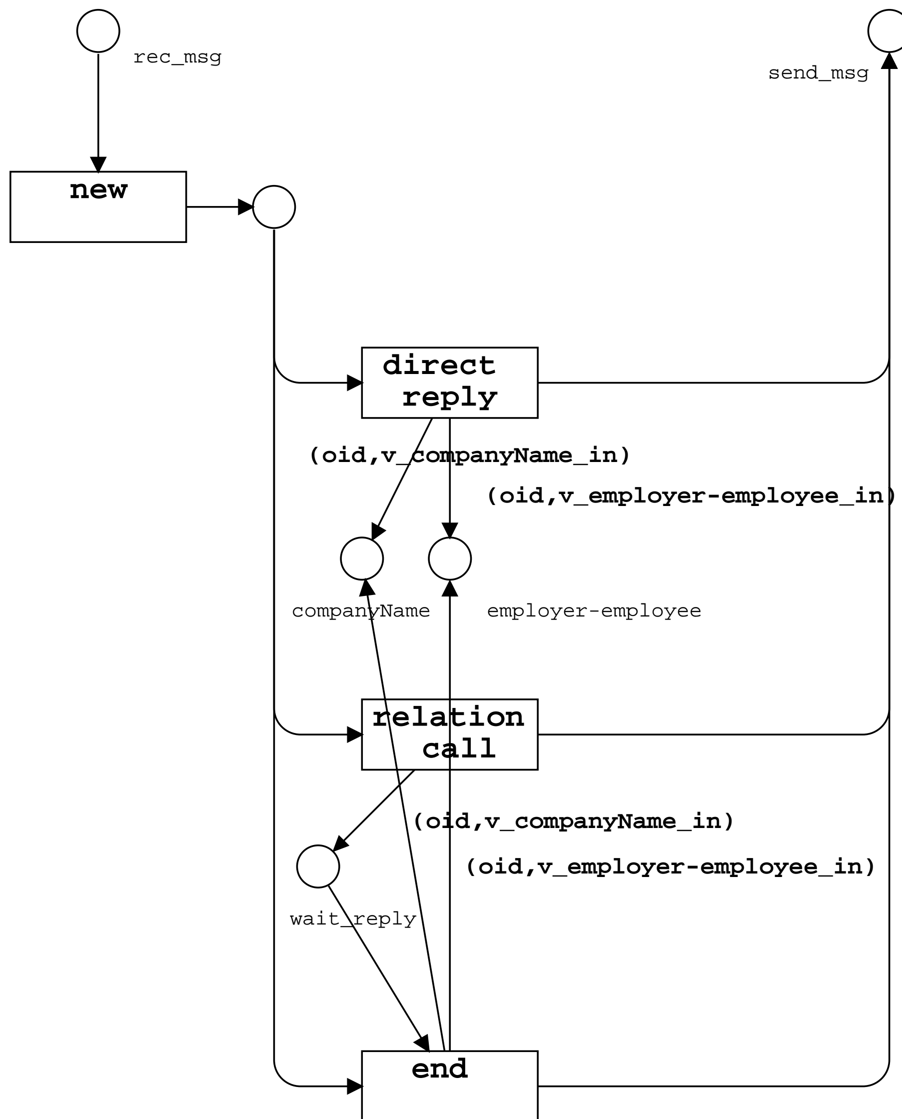

Fig. 8. Subpage of method *new*

with the transition and contains a predefined reply message. Simple methods can be finally implemented by the user in a very short amount of time using this predefined code region. A more complex method can certainly be implemented by the user utilising all the functionality offered by Design/CPN.

The class method *new* can normally be left unchanged by the user. Figure 8 shows the method's subpage in the case of a class connected to another via one relationship. The method has to distinguish two variants of being called: It can be called by any external object that knows the class or by the relation class. The call of *new* by the relation class is easier to handle because all possible dependencies that arise from the relationship are already satisfied by the calling relation class. The method just has to put some default values and the identifier of the calling relation class instance into the attribute places and reply to the

caller. This is done by the transition *direct_reply* in figure 8. If the method is called by an external object there is a little bit more work to be done: A new instance of the relation class has to be created, this instance has to fulfil all dependencies arising from the relationship. After doing this the relation class object reports back to the waiting new method where the transition *end* is activated and finishes the creation of the object. The call of the relation class is done by transition *relation_call*. This transition also puts an appropriate token into place *wait_replay*. The reply message sent back by the newly instantiated relation class object matches to this token, so that transition *end* can fire. Again a reply message is send to the first calling external object.

3.4 Relation classes

The frontpage of the relation class is so much alike to that of a normal class that the picture is skipped here. It shows the class methods *new* and *delete* and the methods *person* and *company* that help detecting the related counterparts of a given object.

Figure 9 shows the new method's subpage of the relation class hiding less net elements then the figures before. This is done to allow a closer look on the complexity of the generated nets. Normally most of these details are not shown to the user but the degree of complexity shown is easily adjustable within GPS. Due to limited space a description of the course of events happening in the relation class is omitted here. The interested reader may refer to [Hei99] where some detailed example procedures are presented as well as the quite complex code regions of the transitions. Furthermore, the declaration nodes and how construct them according to the input by the designer can be found there. Especially the creation of correct inscriptions is a tidious task. This can be supported by tools.

4 Conclusions and Future Work

To design Coloured Petri Nets in an object-oriented way implicit protocols for the object behaviour have to be created. The GPS tool allows the automatic generation of nets from class diagram descriptions. Relationships between classes are also considered to be classes. They are special classes that incorporate a specific behaviour to ensure the operational behaviour that one expects for the associated classes (e.g. when creating or deleting objects or when navigating from one object to another). However, the protocols which cover these different kinds of behaviour are not shown in this paper. They still have to be added. It should also be mentioned that these protocols must cover many different cases. Even for simple standard cases the starting point of an creation of some objects (related by a mandatory relationship) can affect the kind of protocol. Furthermore, the information that has to be added (in programming languages these would be the parameters), has to be provided. When also covering inheritance there is an

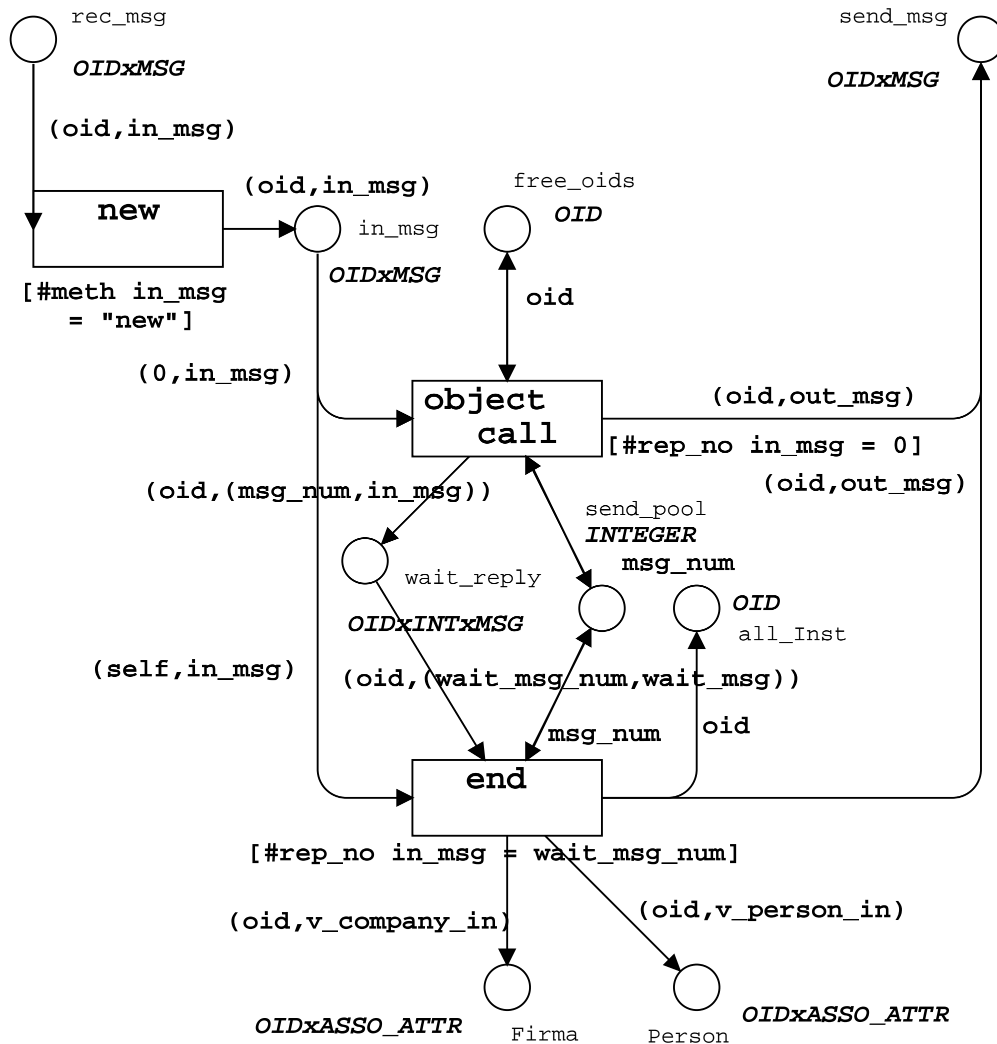

Fig. 9. Subpage of method *new* of the relation class

explosion of the possible entry points[13]. We have drawn some protocols by hand, but did not write the generator for it. Furthermore, more complex associations like inheritance have to be handled. For our general attempt to use OOCPN for specification or at least for execution the missing features have to be provided to provide a reasonable specification environment. The generator is build in a way that it can be easily extended by plug-ins. These plug-ins could then cover the different protocols.

The use of ML for the generator allows the easy integration into the Design/CPN environment, however, hinders the transformation to other environments which are e.g. based on Java.

[13] In [Mol96] delegation is used to implement the inheritance. This concept is quite powerful, however, it requires the explicit representation of aspects which should normally be hidden to a user.

The generated nets and some attempts to apply the OOCPN by hand (without tool support (see [Net99]) show that there have to be some additional, more abstract concepts for users to build OOCPN models. The principal ideas for providing an operational semantics for class diagrams in form of Petri nets have shown to be good, however, the operational side has some disadvantages. Problematic is that diagrams got quite large. Overhead to explicitly model all the operational behaviour induced by the object-oriented concepts could not be ignored. It made up a large percentage of the whole model. The large models to express relatively simple behaviour have a considerable impact on the performance (and the applicability of analysis tools, especially due to the growing state space). Many transitions have to fire before the, from the view point of object-orientation, simple action has been performed. This has lead us to look for shorthand notations. UML can be considered to be one possible solution. However, we can not directly provide an abstraction recommendation which can be applied within the Coloured Petri nets themselves.

One of the goals is to get a better understanding of the semantics of e.g. class diagrams. This can be achieved when providing an operational semantics for them. Tool support helps on this way and so does the GPS tool set.

A Class diagram description language

```
diagram               ::= <classdescription>*

classdescription       ::= 'class' <classname> '{'
                            <relations>
                            <attributes>
                            <methods>  '}'


relations             ::= 'relationship' <relationset>
                          | ()

attributes            ::= 'attributes' <attributeset>
                          | ()

methods               ::= 'methods' <methodset>
                          | ()


relationset           ::= (
                           <relatedClass>
                           <relationtype>
                           <relationname> ';'
                          )*

attributeset          ::= ( <type> <attributename> ';' )*

methodset             ::= ( <type> <methodname> <arguments> ';' )*


relatedClass,
relationname,
attributename,
methodname            ::= identifier

relationtype          ::= '1-1' | '1-n' | 'n-1' | 'n-m'

arguments             ::= '(' <typeset> ')'

typeset               ::= <type> ',' <typeset> | <type>

type                  ::= 'void' | 'int' | 'string' | 'real'
```

References

[BG91] Didier Buchs and Nicolas Guelfi. CO-OPN: A Concurrent Object Oriented Petri Net Approach. In *Application and Theory of Petri Nets, 12th International Conference, Gjern, Denmark*, pages 432–454. University of Aarhus, IBM Deutschland, June 1991.

[BM93] Ulrich Becker and Daniel Moldt. Objektorientierte Konzepte für gefärbte Petrinetze. In Gert Scheschonk and Wolfgang Reisig, editors, *Petri-Netze im Einsatz für Entwurf und Entwicklung von Informationssystemen*, Informatik Aktuell, pages 140–151, Berlin Heidelberg New York, 1993. Gesellschaft für Informatik, Springer-Verlag.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The unified modeling language user guide: The ultimate tutorial to the UML from the original designers.* Addison-Wesley object technology series. Addison-Wesley, Reading, Mass., 1999.

[Des93] Meta Software Corporation, Cambridge, MA, USA. *Design/CPN Handbook Version 2.0*, 1993.

[EMNW99] Adriana Engelhardt, Daniel Moldt, Marc Netzebandt, and Frank Wienberg. Erweiterung objektorientierter gefärbter Petrinetze um Typisierung und Schnittstellen. Fachbereichsmitteilung planned, FBI-HH-M-xxx/99, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, October 1999.

[Fow97] Martin Fowler. *UML Distilled.* Addison-Wesley Longman, Inc., 1st edition, 1997.

[Hei99] Heiko Rölke. Transformation von Klassendiagrammen in Objekt-Orientierte Petrinetze unter besonderer Berücksichtigung von Assoziationen. Studienarbeit, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, April 1999.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process: UML; The complete guide to the Unified Process from the original designers.* Addison-Wesley object technology series. Addison-Wesley, Reading, Mass., 1999.

[Jen92] Kurt Jensen. *Coloured Petri Nets: Volume 1; Basic Concepts, Analysis Methods and Practical Use.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin Heidelberg New York, 1992.

[Kum00] Olaf Kummer. *Referenznetze.* Dissertation, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, 2000.

[Lak95] C.A. Lakos. From Coloured Petri Nets to Object Petri Nets. In *16th International Conference on the Application and Theory of Petri Nets*, number 935 in Lecture Notes in Computer Science, pages 278–297, Torino, Italy, 1995. Springer.

[LM98] R.B. Lyngsø and T.M Mailund. Textual Interchange Format for High-Level Petri Nets. In *Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, pages 47–64, Ny Munkegade, Building 540, DK-8000 Aarhus C, Dänemark, 1998. Computer Science Department, Aarhus University.

[Lou93] Kenneth C. Louden. *Programming languages: principles and practice.* PWS-Kent, Boston, 1993.

[Mai96] Christoph Maier. Darstellung von Konzepten der objektorientierten Modellierung und Programmierung mit Petrinetzen. Studienarbeit, University

of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, May 1996.

[MM99] Christoph Maier and Daniel Moldt. Object Coloured Petri Nets – a Formal Technique for Object Oriented Modelling. In G. Agha, F. De Cindio, and G. Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, number yet unknown in Lecture Notes in Computer Science, Berlin, 1999. Springer-Verlag.

[MMR98] Christoph Maier, Daniel Moldt, and Heiko Rölke. SNIFF – An Input/Output Library for Design/CPN. In *Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, pages 65–82, Ny Munkegade, Building 540, DK-8000 Aarhus C, Dänemark, 1998. Computer Science Department, Aarhus University.

[Mol96] Daniel Moldt. *Höhere Petrinetze als Grundlage für Systemspezifikationen*. Dissertation, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, August 1996.

[Net99] Marc Netzebandt. Untersuchung der Einsatzmöglichkeiten von Petrinetz-Konzepten in der objektorientierten Analyse am Fallbeispiel eines Reiseunternehmens. diploma thesis, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, Januar 1999.

[Pau92] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1992.

[Pet62] Carl Adam Petri. Kommunikation mit Automaten. Dissertation, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, Bonn, 1962.

[Rei92] Wolfgang Reisig. *A Primer in Petri Net Design*. Springer Compass International. Springer-Verlag, Berlin Heidelberg New York, 1992.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The unified modeling language reference manual: The definitive reference to the UML from the original designers*. Addison-Wesley object technology series. Addison-Wesley, Reading, Mass., 1999.

[SB94] C. Sibertin-Blanc. Cooperative nets. In Robert Valette, editor, *15th International Conference on the Application and Theory of Petri Nets*, number 815 in Lecture Notes in Computer Science 815, pages 471–490, Berlin Heidelberg New York, 1994. Springer-Verlag.

[Val91] Rüdiger Valk. Modelling Concurrency by Task/Flow EN Systems. In *Proceedings 3rd Workshop on Concurrency and Compositionality*, number 191 in GMD-Studien, St. Augustin, Bonn, Germany, 1991. Gesellschaft für Mathematik und Datenverarbeitung.

[Val98] Rüdiger Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In Jörg Desel, editor, *19th International Conference on Application and Theory of Petri nets*, number 1420 in LNCS, Berlin, 1998. Springer-Verlag.

[Weg87] Peter Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, Cambridge, 1987. Cambridge University Press.

[You89] Edward Yourdon. *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs New Jersey, 1989.