Parametrisation of Coloured Petri Nets

Søren Christensen and Kjeld H. Mortensen

University of Aarhus, Computer Science Department,
Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark
{schristensen,khm}@daimi.aau.dk

Abstract

In this paper we propose a conceptual framework for parametrisation of Coloured Petri Nets — a first step towards the formulation and formalisation of *Parametric* Coloured Petri Nets. We identify and characterise three useful kinds of parametrisation, namely value, type, and net structure parameters. While the two former kinds are simple to design the latter kind is more complex, and in this context we describe how net structure parametrisation naturally induces concepts like modules and scope rules. The framework is applied to a non-trivial example from the domain of flexible manufacturing. Finally we discuss implementation issues.

1 Introduction

When we wish to make a computer representation of a large family of entities or objects of interest from the world around us, we can either choose to represent all individual objects or try making more efficient representations. The perspective on a given problem has influence on the kind of efficiency needed. For instance, space efficiency is often a concern. Different approaches exist for making efficient representations — the one concerning us in this paper is that of parametrised representations. The fundamental idea is to represent only a part common for all objects in a family and characteristic holes of interest which later can be filled in. For instance, flexible manufacturing cells in a bottling system could consist of an input buffer, a transportation system, and a machine. We can imagine a parametrised representation of a generic flexible manufacturing cell where, e.g., the machine would be a parameter (the hole). Thus if we wish to have a packaging manufacturing cell we just instantiate the generic manufacturing cell by inserting a packaging machine into the hole. Note, in addition, that the machine itself may be parametrised.

In the process of designing systems it is often convenient to describe a family of systems instead of one specific system. Once we have made a parametrised representation we have a generic and general description which easily can be instantiated or specialised since the locations for substituting concrete entities have already been specified in well-defined locations. Additionally, verification of systems benefits in the case where it is possible to reason about a parametrised representation, i.e., determining a property for a family of systems instead of reasoning about each individual system. For instance we may be able to prove

by induction that a property holds for an infinite family of systems only characterised by, say, one integer parameter.

In this paper we propose a conceptual framework for parametrisation of Coloured Petri Nets (henceforth abbreviated as CP-nets or CPN) [9], and illustrate how the CPN tool, Design/CPN [11], can support parametrised CP-nets. Our aim is to improve the modelling convenience of CP-nets and to improve tool support. We argue that CP-nets can benefit from becoming parametrised which we illustrate with examples. Parametrisation enhances the support for reusable components and is a supplement to the hierarchy concept of CP-nets, and therefore parametrisation also enhances the techniques for designing large scale systems. Additionally, parametrisation provides a flexible and time-saving technique for building models. Once we have a number of basic parametrised building blocks (modules) we can quickly put together a model by specialising modules by supplying specific parameters. Changing parameters is also easy because we avoid the need for re-compilation — only re-instantiation with the new parameters is required. (In this paper a module consists of a hierarchy of CPN pages.)

For CP-nets we have chosen to distinguish between three kinds of parametrisation: value, type, and net structure parameters. Although all parameters are just place-holders, we wish to characterise each level individually because the three of them are different in nature. Additionally, when we look at tool support we are both inspired and restricted by the target language of the Design/CPN simulation engine, namely the language Standard ML (abbreviated as SML) [15, 17]. This language has a construct called functors which provides a module structuring facility with parameters.

The use of parametrised modules as library units introduces the issue of name clashes. Suppose we are building a model and then import some external library module. The external module typically contains new colour set (type) declarations, fusion sets, and many other name declarations. What should happen if a name in our model is in conflict with a name in the external module? Currently all names have global scope, except from fusion sets which do have simple scope rules. In this paper we introduce a general mechanism for resolving name clashes in the form of scope rules. Other computer languages, such as block-structured languages, usually have scope rules of some kind.

The synopsis of the paper is as follows. We begin with motivating the need for parametrised CP-nets in Sect. 2 and declare our essential goals. Supported by this we describe the conceptual framework in Sect. 3 for parametrised CP-nets, which constitutes the main part of this paper. In Sect. 4 we describe our design ideas of scope rules for name declarations, such as colour sets, and in this context generalised the current scope rules for place fusion groups. Then in Sect. 5 we support the usefulness of the conceptual framework by means of a non-trivial example of a manufacturing system. Implementation issues are discussed in Sect. 6. Future work, related research, and the conclusion can be found in Sects. 7, 8, and 9 respectively.

2 Motivation and Problem Analysis

In this section we motivate the use of parametrised representations by informally looking at a specific example with the purpose of investigating the possibilities

of parametrised representations. By means of the example we introduce the concepts needed for our work, and identify and suggest an initial overall set of requirements. The intention is to provide an overview and exhibit the considerations we have made in order to make a framework for parametrised CP-nets.

The example we use in the following is inspired by the domain of flexible manufacturing. We illustrate the usefulness of the three kinds of parametrisation studied here: value, type, and net structure parameters. Manufacturing systems typically consist of the following three classes of entities: materials, machines, and transportation [7]. Material flow through a system by means of a transportation system while the material is manipulated by means of machinery. Our example manufacturing system is depicted in Fig. 1. The figure

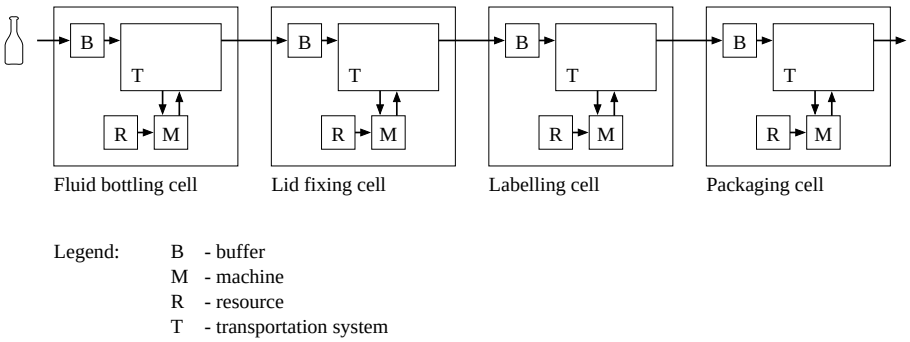

Figure 1: An example of a simple bottling and packaging manufacturing system.

shows a simple bottling (and packaging) manufacturing system. The bottling system is built up in a modular fashion where each module is represented as a rectangle. Materials flowing through the system are bottles, liquids, lids, labels, and packages. The bottles induce the main flow while the rest of the materials are local to each manufacturing cell. Therefore we describe in the following the dynamics of the system from the viewpoint of the bottles. Each bottle enter the bottling manufacturing system in the leftmost cell, the fluid bottling cell, where it is first put into a buffer (B). Then the bottle is transported (T) on a conveyer belt to the fluid bottling machine (M) which takes fluids from its local resource (R), and the bottle is transported out of the cell to the buffer in the next cell, the lid fixing cell. In this cell the bottle is mounted with a lid and is transported via a conveyer belt into the buffer of the next cell, the labelling cell. Here labels are fit to the bottle which then is transported via a conveyer belt into the buffer of the next cell, the packaging cell. The transportation system in the packaging cell is in this module a robot arm which takes bottles one at a time from the buffer and put it into a packaging box mounted in a packaging machine which closes and wraps the box when full of bottles. The boxes are finally transported out of the system by means of the robot.

Obviously the four manufacturing cells above have a lot in common which also is reflected in the figure. Each cell has a buffer, a transportation system, and a machine with a resource. Furthermore, each cell manipulates bottles one way or the other. In this case it may be advantageous to make a generic parametrised manufacturing cell, because we can then use this generic cell as a building block and make various specialised instances as needed. We can even make variations

of the bottling system, e.g., by reordering the cells to investigate alternative assembling sequences, or by adjusting whatever characteristic parameters such as transportation speed or method. To make a generic cell we need to consider which parameters we need in order to be able to describe the current four kinds of cells in the example, but also future possible variants of cells. For bottling manufacturing systems we probably need the following parameters for a generic bottling manufacturing cell:

- Bottle kind
- Buffer
- Transportation system
- Machine
- Resource

The bottle kind is a type parameter while the rest are net structure parameters (using a module). Each of these modules can again be parametrised:

- Buffer
  - Size (value parameter)
  - Bottle kind (type parameter)
  - Functionality (net structure parameter)
- Transportation system
  - Functionality (net structure parameter)
  - Transportation speed (value parameter)
  - Capacity (value parameter)
- Machine
  - Functionality (net structure parameter)
  - Processing speed (value parameter)
- Resource
  - Material (type parameter)
  - Size (value parameter)

We call these *formal parameters*. The items assigned to formal parameters are called *actual parameters*.

The example above is useful for trying out the initial ideas for parameterised representations. The concept of parametrised representations are a useful technique in the support for structuring a system design. It is easy to imagine that parametrised modules can be used both in a top-down and a bottom-up fashion, and that it can be used together with the hierarchy concept of CP-nets. Parametrisation seems in particular to be useful for describing systems with many embedded modules which can have many specialised variants. Systems

such as flexible manufacturing systems often need to be analysed by means of numerous simulation runs, typically where a few simple parameters are perturbed for each run in a series. Thus it is useful to use parametrised representations in connection with scripting where numerous repeated runs are needed, say, for identifying significance in a set of statistical samples. It becomes just a matter of programming the script to choose the parameter series. This implies that we can easily imagine that value parameters often will be used in the initial marking configuration and similar constant value expressions.

Parametrisation is useful for other systems than flexible manufacturing systems. In general, parametrised generic representations can be used to build a reusable model library of standardised modules. With a well-designed library there is support for building models with an advanced vocabulary which is on a level of granularity suitable for the problem domain. For instance, we expect that the domain of hardware design can benefit from parametrised CP-nets. Another class of candidates is layered protocols which profitably can be described as parametrised representations. It would be useful to be able to shift between different variants of a layer by means of a quick, easy, and safe plug-in method.

Currently the formal model of CP-nets does not contain a parametrisation concept. The Design/CPN tool does not currently support parametrised CP-nets, as the tool implementation is influence by the formal model of CP-nets. In spite of this, value parameters can be imitated in an *ad hoc* fashion. However, the current technique is both cumbersome and unsafe to use. It is cumbersome because changing a parameter may require a time consuming re-check of the CPN model, and unsafe because it is easy to make logical mistakes.

Hence our goal is to make support for parametrised representations for CP-nets, the vision being that parametrised CP-nets are a useful technique for designing and reasoning about systems. Our aim is to enhance the modelling convenience of CP-nets and to make tool support for using parametrised CP-nets to build more abstract and generic designs. In the sections following we concretise our ideas by making a conceptual framework of parametrised CP-nets (Sects. 3 and 4).

3 Conceptual Framework

In this section we propose a conceptual framework and design ideas for the parametrisation of CP-nets. We consider a number of key questions: How can CP-nets be parametrised? Which elements of parametrisation can we offer for CP-nets? Can we hope to allow for analysis on the level of parametrised representations? Scope rules are considered separately in Sect. 4.

3.1 Variants of Parametrisation

In general parametrisation is the act of making holes (place-holders) in a representation which then later can be instantiated by filling out the holes with concrete entities. The entities are restricted by the given context of use. We can benefit from characterising parametrisation in sub-categories. For instance if we parametrise with integers then we can immediately make a number of assumptions because integers are a very restricted sub-category. On the other

hand, if we can parametrise without restrictions then it is hard to make assumptions and thus we may lose the possibility to investigate important properties.

Where does it make sense to make holes? That depends on several factors. The main restriction origins in the constituents of the language, and the syntactical categories. For CP-nets we have syntactic elements such as places, transitions, arcs, inscriptions, and declarations. The former three are in the category of net structure while the latter two are textual. For the textual entities we typically find named values and type identifiers. This is also influenced by the specific inscription language which for our case is SML. Thus influenced by CP-nets and the inscription language SML it is natural to investigate parametrisation with values, types, and net structure.

Another issue is more of pragmatic nature: On what level of granularity do we wish to locate a parameter specification? We want to use parametrised representations in practice and it is therefore interesting to investigate parametrisation on a higher level of granularity, e.g., modules. In fact parametrisation of modules is an interesting candidate since the nature of a module is of being predominately self-contained and encapsulated unit, only loosely coupled with the environment. Thus we can expect that a module defines a natural and clear boundary and interface for declaring a parameter specification.

Additionally, there is a trade-off between declaring in advance which names that can be used as parameters or let every name be a potential parameter. We choose the former because the process of parametrising a CPN model also includes identification of exactly where parameters must appear in the net structure. Thus in the user interface there must be support for making a parameter specification of modules. Additionally we could imagine that default values for parameters would be useful such that the user avoids supplying often used values, e.g., empty lists.

Each parametrised module has a *parameter specification* which is the list of parameter names used within the module in question. From a parameter specification we can derive, what we here call, a *module signature* which is essentially the parameter names and their types. This is analogous to SML signatures. Signatures are used to ensure that the use of a module in another is consistent in the sense of type safeness.

In the sections following we treat each of the three kinds of parametrisation separately. Our purpose is to discuss and identify useful properties of the three concepts, where we take advantage of the restrictions that each of the three levels impose.

3.2 Value Parametrisation

Parametrisation with values is the simplest to understand of the three kinds of parametrisation we consider. It is simple because it is only a few well-defined locations in the syntactic categories of CP-nets where values occur. This imposes many restrictions on how and where value parameters can be used in a CPN model. First we give an example.

Example of Value Parametrisation

We focus on the machine module in the flexible manufacturing example from Sect. 2. A CPN model of this module is depicted in Fig. 2. On the arc go-

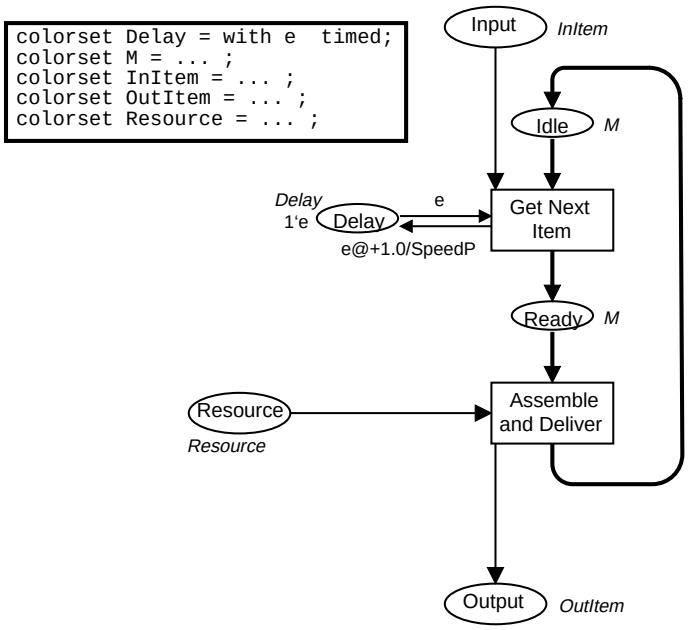

Figure 2: CPN model of a machine which can be used inside our manufacturing cells.

ing to the place *Delay* we have the inscription *(e @+ 1.0/SpeedP)*, implicitly requiring *SpeedP* to be of type real — otherwise we would violate the type system (of SML). (The notation *(exp @+ texp)* means that the multi-set (tokens) generated by expression *exp* has a time delay *texp*.) Our intention is that the name, *SpeedP*, is a formal parameter of the machine module. This means that upon instantiation of this module we need to supply a value to be substituted on the place-holder of *SpeedP*. If we instead had written *SpeedP(x)* then the value parameter *SpeedP* is a function taking one argument x and returning a time value of type real. In the most general case the inferred type of x would be polymorphic. Another useful example of value parametrisation is in initial marking expressions, thus making the initial system configuration more flexible.

Design Ideas for Value Parametrisation

The example suggests that value parametrisation is a simple and useful mechanism. Value parameters can be simple values or functions. The latter is, of course, inspired by the inscription language, SML, of Design/CPN. In this language, functions are first class values. In the example we also saw that the type of a value parameter can be specific or polymorphic. The type is either explicitly annotated, or implicit where the tool then must infer the type.

Based on our findings in the example and discussion above, we summarise our requirements for value parametrisation of modules:

1. A formal value parameter can be assigned any first class value (actual parameter) which can be realised in the inscription language.

2. A formal value parameter has a name and can appear in any inscription as a place-holder. The name must appear where the syntactical category is a value expression.

3. A formal value parameter has a type which is either explicitly given by the user in the module parameter specification, or implicit and inferred by the type checker system.

4. A formal value parameter inside a module must be mentioned explicitly in the parameter specification of the module, including a name, an optional type, and an optional default value.

3.3 Type Parametrisation

Type parameters are, like value parameters, also easy to understand. They appear only a few locations in the syntactic structure of declarations and inscriptions of CPN models. Types can also be polymorphic which implies polymorphic CP-nets.

Example of Type Parametrisation

The example we use for the illustration of type parameters is a generic CPN model of the flexible manufacturing cell, i.e., a model which describes a certain collection (or class) of manufacturing cells as used in Fig. 1. The generic cell is depicted in Fig. 3. In this example we have included a few colour set

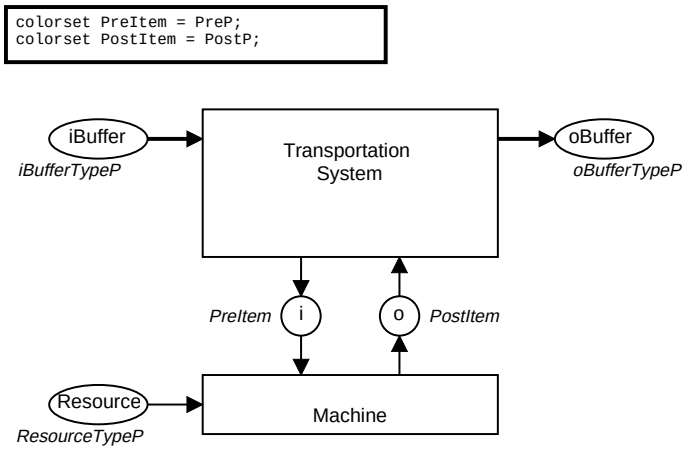

Figure 3: A generic CPN model of the flexible manufacturing cell.

declarations. This is to illustrate that we wish to be able to parametrise here also. It is the intention that the two types for *PreP* and *PostP* are supplied upon instantiation such that *PreItem* and *PostItem* are meaningful. We can also have formal type parameters in colour set inscriptions of places such as with the places *iBuffer*, *oBuffer*, and *Resource*. The formal parameter names here are then *iBufferTypeP*, *oBufferTypeP*, and *ResourceTypeP*. Parametrised type annotations in arc inscriptions should also be possible, e.g., *(x:TypeParam)*

would be a legal inscription where *TypeParam* is a type parameter. Upon instantiation the expression x would be restricted to whatever type supplied by the user. Such an annotation implies that we can speak about a polymorphic CPN module. The polymorphic types are visible in the module signature.

A more advanced use of type parameters may appear in relation with SML pattern matching. Consider the input arc inscription *((p:Packet) as {sender=s, ...})*, where p and s are variables, *Packet* is a formal parameter type, *sender* is a record field, and "..." is part of the inscription syntax ("all the rest"). This means that, we impose the requirement that the place must contain record tokens which includes at least the *sender* field — a very flexible technique because extensions to the *Packet* type does not require modifications to the arc inscription.

So far we have looked at, so called, *parametric polymorphic* types [3]. Below we give an example of exploiting *ad hoc polymorphic* types, more specifically *overloaded* types. Overloading is not a part of SML but is a feature of the specific SML compiler used for implementation. The overloaded types are not used as parameters but we show that they are useful for parametrised CPN models. Assume we have a timed CPN model where we would like a flexible representation in the sense that it should be painless to change between the two time representations integer and real. Let us look at the timed arc inscription from Sect. 3.2: *(e @+ 1.0/SpeedP)*. This form is inflexible if we change the time type to integer because we need to change 1.0 to 1. A more flexible alternative is: *(e @+ Inverse(SpeedP:SpeedTypeP))*, where *Inverse* is an overloaded function, *SpeedP* is a formal value parameter, and *SpeedTypeP* is a formal type parameter. Now we can easily change between integer and real time by only changing module parameters.

Design Ideas for Type Parametrisation

The example above shows that we can use type parameters practically just like value parameters, with the extra feature of type inference. We do, however, not consider sub-typing mechanisms (*inclusion polymorphism* [3]) in this work because the target language SML does not support this. Design/CPN does, however, support a limited version of sub-typing in colour set declarations. The example also indicates that we can take advantage of polymorphic types to express more general polymorphic CPN models.

Based on our findings in the example and discussion above, we summarise our requirements for type parametrisation of modules:

1. A formal type parameter can be assigned any type (actual parameter) within the restrictions of the type inference system.
2. A formal type parameter has a name and can appear in any inscription as a place-holder. The name must appear where the syntactical category is a type expression.
3. A formal type parameter can be polymorphic which implies that the module in question gets a polymorphic signature.
4. A formal type parameter inside a module must be mentioned explicitly in the parameter specification of the module, including a name and an optionally default concrete type.

3.4 Net Structure Parametrisation

We have seen that value and type parametrisation are fairly simple mechanisms to handle. It is a different matter with net structure parameters. To be useful we need to be able to parametrise with chunks of CP-net structure, and for this we need to specify exactly how the *rim* of the chunks should be glued into the hole. There are similarities with substitution transitions where the role of these kinds of transitions are to be net macros. In this work we consider the net structure parameter to represent a chunk which is a CPN hierarchy, i.e., a hierarchy of pages, which can be inserted into a module place-holder. To simplify the discussion and to keep the analogy with substitution transitions we restrict ourselves to place-holders being the syntactical category of transitions only. This means that a transition name can be a formal net structure parameter.

Example of Net Structure Parametrisation

We reuse the example from Fig. 3 in Sect. 3.3. In that figure we see the transition called *Machine* which in the following is a formal net structure parameter. Our intention is to assign a module, such as the machine module in Fig. 4, to the net structure parameter. Just as with value and type parameters, the net structure

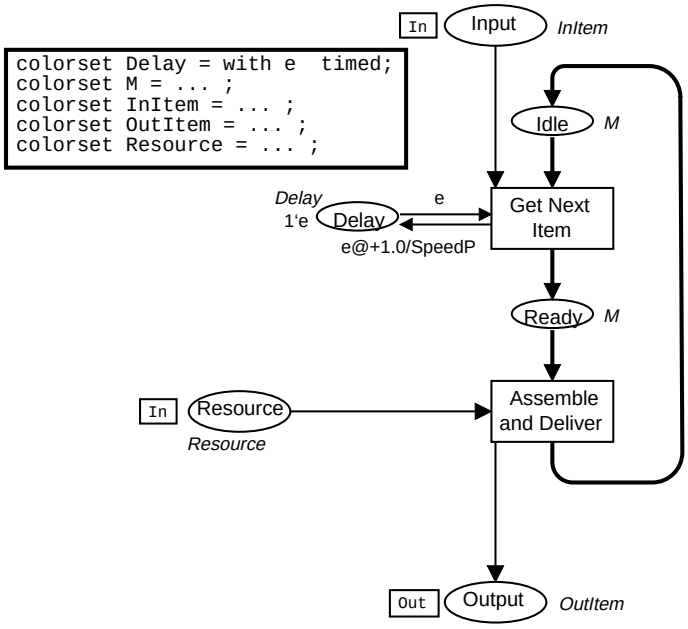

Figure 4: The machine module from Fig. 2 prepared to be used as a formal net structure parameter.

parameter is nothing but a place-holder and we postpone any parameter or interface place assignments until instantiation time. However, in the machine module we need to point out exactly which places that can be used as a module interface in a parametrisation relation, otherwise we would not necessarily know which places to use. Thus we need to explicitly declare the places *Input*, *Output*,

and *Resource* as module interface places. In Fig. 4 we visually use the respective tags *In*, *Out*, and *In*. Finally, in the manufacturing cell module in Fig. 5 we

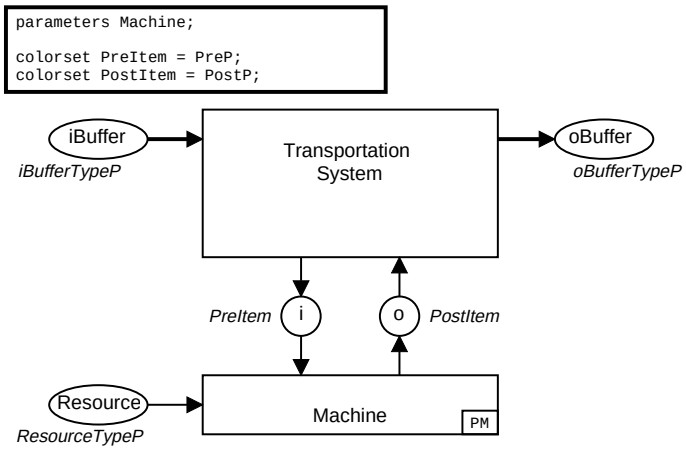

Figure 5: The generic manufacturing cell module from Fig. 5 with the Machine transition as a formal net structure parameter.

need to declare that *Machine* is a formal parameter by adding the name of the parameter in the parameter specification inside the manufacturing cell module itself. As a graphical convention we use the *PM* tag to visualise the Machine net structure parameter.

Design Ideas for Net Structure Parametrisation

To declare that a transition represents a formal net structure parameter is just as simple as with value and type parameters; in a parameter declaration we simply list those names of transitions we wish to be net structure parameters. Additionally we need to specify those places which are interface places of each module. Upon instantiation time the interface places are assigned to the places surrounding the parameter transition, similar to port/socket assignments in hierarchies. As an alternative we could allow any place of a module to be assigned upon instantiation and thus allow any net structure interface relation with the surrounding net. This is indeed possible, however we prefer the net structure interface declaration because in this case we force the user to separate out a well-defined net structure interface to each module. Finally, the interface places in Fig. 3 have types (colour sets). In Sect 5, where we look at a larger example, we suggest that these types can be omitted, where the idea is to let the surrounding net determine, or at least overwrite, the type of the interface places. Thus a type compatibility check is required between matched interface places.

Based on our findings in the example and discussion above, we summarise our requirements for net structure parametrisation of modules:

1. A formal net structure parameter has a name which denotes a syntactical category of a transition.

2. A module which has the purpose of being used as an actual net structure parameter must declare a number of named places as its net structure interface. (In hierarchies these places are called ports.)

3. Assignment of places (actual into formal parameters) should be explicit and unambiguous by means of place names.

4. A formal net structure parameter inside a module must be mentioned explicitly in the parameter specification of the module, including a name and an optionally default module name.

5. Formal net structure parameters implies a relation between modules, thus inducing a hierarchy of parametrised modules. The module relation is considered supplementary to the hierarchical substitution relation.

3.5 Runtime System Parametrisation

In this section we extend the concept of parametrisation to include also the environment in the following sense: The (simulation) runtime system can provide parameters which can be used in the model. We use the term *runtime system parameters* for this purpose.

An example of a runtime system parameter is the function called *inst* as described by Jensen [9] (p. 93). The *inst* function is a parameter which only has meaning during execution, i.e., the function only has a value while the runtime system controls the execution. The function provides the current page instance number. It is the runtime system which provides the value of the *inst* function. In general all parametrised CPN models should have available a number of default parameters which only are supplied with values by the runtime system, i.e., parameter assignments beyond the control of the user.

So far we have only considered value parameters provided by the runtime system. In the following we also investigate type and net structure parameters. As we have seen in Sect. 3.3, type parameters imply polymorphic models — a very useful mechanism for making generic models. Once we provide a concrete type as a parameter we immediately restrict the use of values in the model. Suppose we have a CPN model with time. In this case the runtime system of Design/CPN supplies a concrete type for the type name called *TIME*. This can be either *int* (integer) or *real* (floating point). Thus if the user declares functions in the time domain, then it is advisable to use the type name (formal type parameter) TIME instead of restricting oneself on either integers or reals.

Finally, an example of a net structure runtime parameter could be a platform dependent runtime library of modules. Suppose we have a CPN model where some kind of communication with external components (hardware) takes place. Then when using modules from the runtime library on the Macintosh the system automatically provides the appropriate modules for that platform.

3.6 Putting Modules Together and Instantiation

Until now we have, in this section, considered various kinds of parametrisation. We saw that parametrisation naturally implied modules as the basic building-block. Below we describe the issue of building a model based on parametrised modules and the issue of instantiation.

In Sect. 3.4 we provided design ideas for net structure parameters and thus decided to use parametrised modules for this purpose. Some of the important characteristics of modules are that they are self-contained units with well-defined interfaces, and no or only a few relations and dependencies with other modules. This means that we should have the possibility of using declarations, such as types, variables, functions, etc, locally in each module. Hence we use the term *module declaration* for this purpose. In Sect. 4 we consider scope rules for local declarations among others.

In order to instantiate a CP-net we need a specific module as origin, namely a module containing all the prime pages. We call this special module the *root module* of instantiation. This is the only module which can contain more than one hierarchy of CPN pages, and in the tool this module will be one CPN hierarchy. The root module will, if necessary, refer to other parametrised modules. A module declaration is in particular useful in the root module when making common declarations for all hierarchies with a prime page.

A special section of a module declaration consists of the declaration of the parameters inside the module, i.e., the *formal parameters*. We use the term *parameter specification* for this purpose. From the parameter specification we can derive an overview of the module in the form of a *signature* which is a list of parameter names; for each value parameter name also its inferred type, for each type parameter also its most general inferred type, and for each net structure parameter its interface places. It is the intention that it is the tool itself which derives the signature, unless the user explicitly have supplied additional parameter information in the specification. The signature is useful in connection with instantiation where the tool then quickly can determine whether or not the parameter assignments of the user are valid.

As part of the net structure parametrisation framework we explained that each module has a number of interface places. When specifying how a module is used in another we need to make place assignments. This is simple because this can just happen when making assignments of the formal parameters of the module, i.e., we treat assignment of parameters and interface places on an equal footing. As a tool feature we can make it such that the user involvement part of the place assignment process can be kept to a minimum. Many assignments of interface places can in principle happen automatically. We can simply make a heuristics for place interface assignment. The idea is to take advantage of identical names, types, or in/out tags. This is how it currently works in Design/CPN.

We do not really need to explicitly type the interface places as we can just use the type from the places of the context module where the parameter module is embedded. We say that the types of the context module *overwrites* the types in the parameter module.

Once we are satisfied with the parametrised modules and are ready to link the modules together to form a CPN model, we need a notation for assigning parameters. Suppose we wish to make a small model by means of the modules from the Figs. 3 and 2. In the generic cell module we need to specify the assignments of the parameters relevant for the generic machine module. We thus relate the transition *Machine* in Fig. 3 with the following assignment expression:

```
GenericMachine[
  10 -> SpeedP
```

```
    real -> TimeTypeP
    WinePack[] -> Functionality
    iBuffer -> iPostTrans
    oBuffer -> oPostTrans
    i -> oPreTrans
    o -> iPostTrans
  ] -> Machine
```

where the notation *actual_parameter* → *formal_parameter* means that we assign *actual_parameter* to *formal_parameter*. The last four assignments are assignment of interface places. We must also denote a multi-set of modules to be the starting point of instantiation, and in this context we just make use of the prime page concept from the CPN formalism and Design/CPN tool.

Once the user has given an instantiation relation for a CPN model we can derive a graph which shows the modules and their dependencies. If the user changes net structure parameter assignments then the overview graph will change accordingly. We call such a graph for the *module dependency graph*. See Sect. 5.1 for an example of such a dependency graph. It is important to note that such a graph must be acyclic in order to prevent infinite instantiation.

3.7 Open Runtime Environment

Just as the CP-net model can be parametrised, so can its runtime system. A parametrised runtime system is a kind of an open environment which can be tailored to perform specific tasks. Parameters can be supplied by the user via a (special purpose) user interface. As an example, the tool Design/CPN has a user interface where many different parameters can be changed. For instance, the user can control when a simulation should stop, change the degree of concurrency, and the amount of visual feedback.

In general a runtime system which is parametrised is also a simple kind of tailorable system. Environments which are tailorable have the advantage that they can be adapted to more specific purposes by the users themselves, without modifying the original source code. The CPN tool, Design/CPN, is an open environment which is fairly tailorable, and we have already experienced that users extend or tailor the tool to their purposes. For instance, many users have made their own special purpose graphical animation for simulations, others have made their own special kinds of simulations such as Monte Carlo simulations. Yet others have made a temporal logic plug-in module [4] and equivalence extension [12] to the state space component of Design/CPN.

4 Generalised Scope Rules

In Sect. 3 we saw examples of that scope rules for CP-nets would be helpful, e.g., when using parametrised CPN modules as libraries. In the following we summarise the current scope rules with CP-nets and Design/CPN, and then present our design ideas for scope rules of *name declarations*, such as colour sets, for CP-nets. Furthermore, we generalise the existing scope rules for *place fusion groups*.

For CP-nets we currently have simple scope rules for name declarations and names of place fusion groups. These two need to be characterised and

distinguished as they are used on different levels and for different purposes. In this context we use the concept of *name spaces* as a useful technique for managing and keeping different kinds of names separated. We characterise a name space by a name, its use, domain, and a set of scope rules. Hence names from different name spaces are unrelated, and names in different non-overlapping scopes are mutually invisible. The name spaces currently used with CP-nets and Design/CPN are summarised in Table 1.

Name	Use	Domain	Scope Rules
Declarations	general declarations	colour sets, constants, functions	global
Place fusion groups	fusion of places	place groups	global, page, page instance

Table 1: Current name spaces with CP-nets and Design/CPN.

4.1 Design Ideas for Improving Scope Rules

In the following we describe our design ideas for generalised scope rules for name declarations and place fusions. The presentation is guided by examples.

One interesting question is if it is possible to use the same scope rules for both name declarations and place fusions. We believe that the two domains of name declarations and place fusions are rather similar. The difference is, however, that fusion places complicate the fusion scope rules by the fact that fusion works across the instance tree.

Name Declaration Scope Rules

The scope rules for name declarations are inspired by block-structured languages. Blocks determine a scope and a name declared in a block is visible throughout the block and within nested blocks. However, if the same name is declared again inside a nested block, the inner name shadows the name belonging to the surrounding block. We apply similar principles for CP-nets, where we consider a block to be a CPN page. For this purpose, we introduce the concept of *topological name declarations*, e.g., a colour set declaration, which is analogous to a declaration inside a block. The analogous concept of a nested block in CP-nets is the sub-page, i.e., a page which is related with its super-page by the hierarchical substitution relation (represented with a substitution transition). Note that hierarchical substitution essentially is a macro feature, thus similar to nested scopes. Consider the example in Fig. 6. The figure illustrates that a name declared in a hierarchy declaration is visible downwards in the hierarchy structure, except when shadowed in the page called *Page 2*.

We need to consider a case where there apparently seems to be name conflict due to the fact that the hierarchy structure may have a page with two different super-pages. This may happen because the only restriction to the hierarchy structure is that it is acyclic. Consider the example in Fig. 7. We need to make

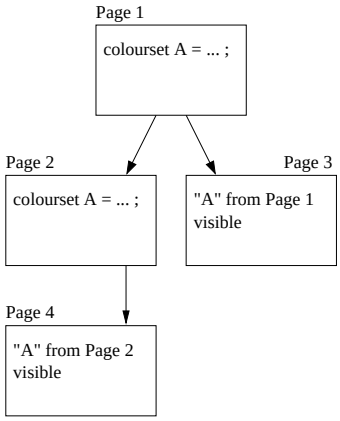

Figure 6: Example of hierarchical name declarations in a substitution hierarchy.

a choice in the bottommost page nodes. Our choice is directed by the instance hierarchy which is always a tree structure. Thus in the instance hierarchy the bottommost page node from before now has two instances. We choose to let each of the pages inherit two different declarations depending on which path is used upwards to find the closest declaration. This means we for some cases need to syntax check a page twice.

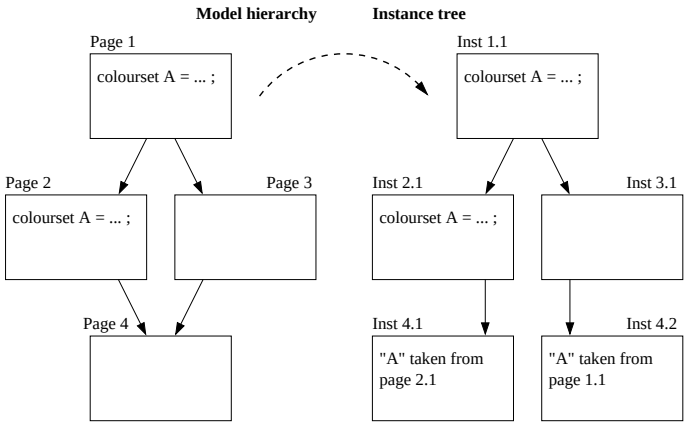

Figure 7: Example of a hierarchy with apparently conflicting declarations.

To avoid some of these shadowing cases we wish to introduce local name declarations, *page declaration*, with a scope limited purely by the page on which it occurs. Consider a variant of the last mentioned figure in Fig. 8. Thus page declarations may help avoiding the extra syntax check which was required in Fig. 7. Alternatively, we could choose to let page declarations shadow names further up in the hierarchy, thus leading to a syntax error, "declaration of A not declared", on *Page 4.1* in Fig. 8. However, we find that it is more important to insert a page declaration scope without affecting other pages in a hierarchy.

We also wish to consider the scope of a module. Recall that we, in this

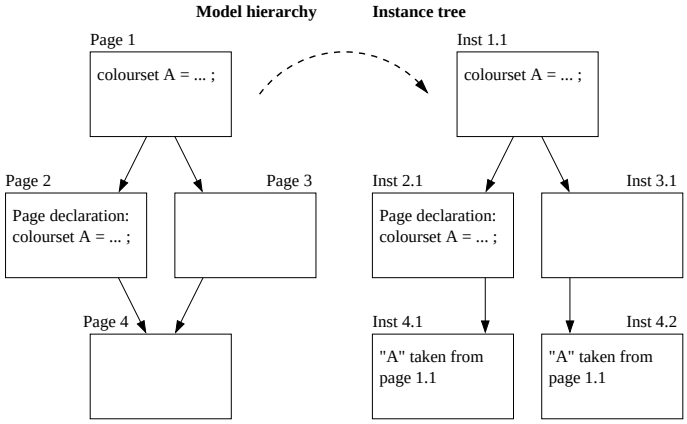

Figure 8: Example of a hierarchy with no conflicting declarations due to a page specific declaration.

paper, consider a module to be a substitution hierarchy of CPN pages. As we consider a module to be a self-contained unit we wish that the scope of a name does not exceed the boundaries of a module. Consider Fig. 9 where we have added an extra module to Fig. 6. The figure shows that the names declared in one module are not visible in an embedded module — unless transferred via a module parameter of course. Thus a module scope is more restricted than a hierarchy scope.

Motivated by the examples we have reached the following scope rules for name declarations used in page hierarchies and modules.

1. A name declaration is visible on the page where defined and all sub-pages in the instance tree.

2. A name declaration may shadow a declaration of a super-page.

3. A *page name declaration* is visible only on the page where it is defined. These declarations do only shadow on the page where defined, and not on sub-pages.

4. *Module declarations* have the scope of the module in which they are defined.

5. Conflicting names are resolved by means of the instance hierarchy structure, which is a tree.

Place Fusion Scope Rules

Current place fusion scope rules consist of three possibilities: global, page, and page instance fusion. A global fusion means that the place is globally visible, thus independent of the instance structure. A page fusion means that the fusion scope is visible only on a specific page but across all instances of the page. A page instance fusion means that the fusion scope is limited to each generated page instance.

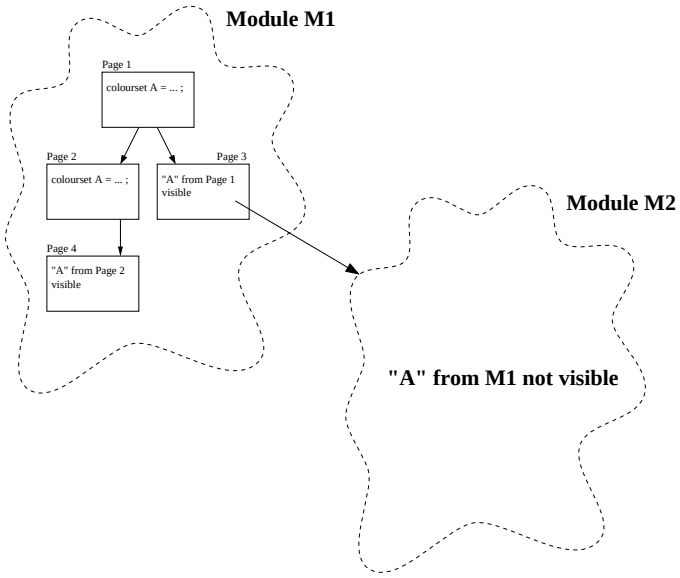

Figure 9: Scoping with CPN modules.

Experience both from our own and industrial CPN models indicate that the current fusion mechanism is not sufficient. In particular there exist several examples where a mechanism oriented towards the hierarchy structure would have resulted in simpler models. We remedy this problem below.

The current fusion scope rules are directed towards pages and instances. Above we saw that scope rules for name declarations, such as colour sets, where directed towards the hierarchical structure. In fact, we use this as motivation for the way we extend the current fusion scope rules with an additional rule related with the hierarchy structure. We refer to this generalised fusion concept as *topological place fusion*.

As we have introduced the notion of modules we wish to reconsider the meaning of a global fusion. We introduce the concept of *module fusion* to mean a fusion place with the scope of all pages in a module and replaces the concept of global fusion. A fusion set declaration on a page hence shadows a module fusion declaration. We do not allow global fusion across modules, thus enforcing the principle that modules are self-contained units with a well-defined interface to their environment.

Analogously to name declarations we can talk about a *place fusion declaration* which defines a fusion scope boundary consisting of the page in question and all sub-pages in the instance tree. (This is the motivation for choosing the name "topological place fusion".) A fusion declaration can be either of the kinds page or instance, and determines, based on the instance tree, how fusion of sub-tree scopes should happen: instance or page wise, respectively. A fusion place will always belong to the same place fusion group as a fusion place located further up in the page hierarchy structure, unless the scope is shadowed with a fusion declaration of the same name.

The general rule for determining the scope works more specifically as follows.

The place fusion groups are determined by inspecting the page instance tree for each fusion place. Given a fusion place on a page, we travel up in the tree until we find a fusion declaration of the same name. The declaration page determines the place fusion group of the fusion place in question. Additionally, if the fusion declaration is of kind page, then we merge the fusion group across all instances of the identified fusion declaration page.

In the example of Fig. 10 we see that the fusion place in page *Inst. 5.1* belongs to the fusion group across all instances of *Page 5*, exactly because the nearest fusion declaration is of kind page. On the other hand, the fusion place

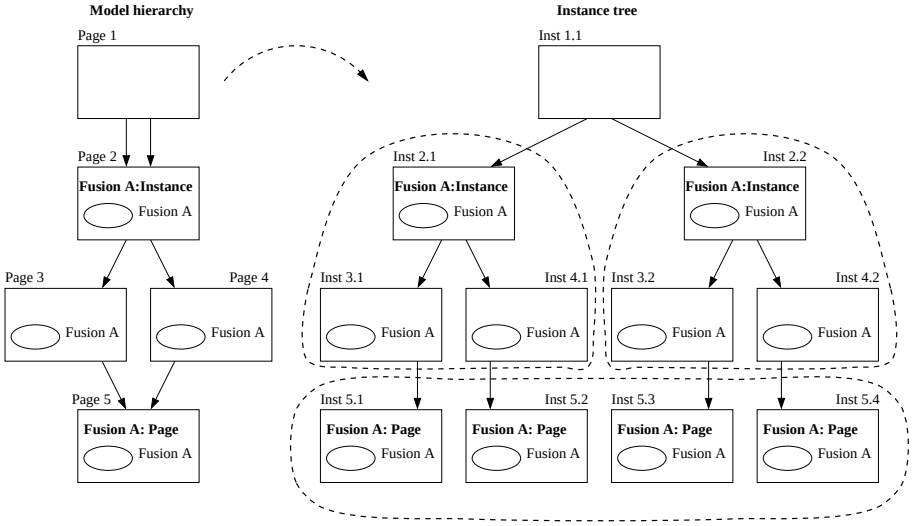

Figure 10: Example of topological instance fusion and topological page fusion.

in page *Inst. 3.1* does not belong to the same place fusion group as the place in *Inst. 3.2* because the nearest fusion declaration (*Inst. 2.1*) is of kind instance. However, the fusion place in page *Inst. 3.1* belongs to the same fusion group as the place in page *Inst. 4.1* because they both are inside the sub-tree of the fusion declaration.

Thus, topological oriented scope rules provides more flexibility oriented towards the page instance tree structure.

5 A Larger Toy Example

In the previous sections we have motivated parametrisation of CP-nets, and made a conceptual framework. In this section we wish to illustrate practical aspects of our work by studying a more elaborate example of the flexible manufacturing system. An example also helps to explore a possible user interface scenario.

5.1 CPN Model of the Bottling Manufacturing System

As our example, we present and describe a CPN model of the bottling manufacturing system of which there is an overview in Fig. 1. We present the CPN

model in a mixture of bottom-up and top-down fashion, and we do it with the granularity of modules. Hence each figure we show is a module, possibly parametrised. We start with a model of the generic manufacturing cell and then, in a top-down fashion, we look at each of its major components, namely the transportation system module and the machine module. Subsequently we glue together variants of the generic cells, in a bottom-up fashion, to form a bottling manufacturing system; where empty bottles enter the system and bottles with fluid, lids, and labels exit the system in packages.

The Generic Manufacturing Cell

We begin with the generic manufacturing cell which is the main building block of our flexible manufacturing system. We model the generic cell as a parametrised module where two of the parameters are formal net structure parameters which are place-holders for a transportation system and a machine module. We only describe the machine module in this section and assume the transportation system for given as an external library module.

The CPN model of the generic manufacturing cell is depicted in Fig. 11. In

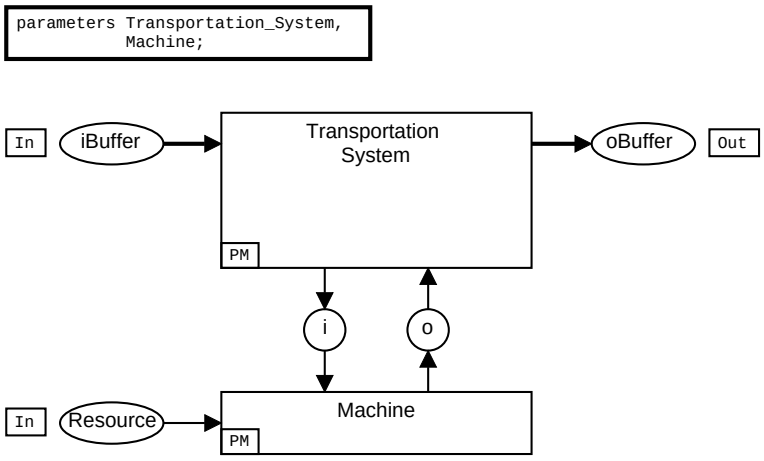

Figure 11: CPN model of the generic manufacturing cell.

Sect. 2 we characterised this module with the following parameters: a transportation system and machine parameter, both of kind net structure. We could have chosen to identify a number of type parameters for the colour sets of all the places. However, we wish this module to be as generic as possible. Thus by leaving them out we assume that the type system infers all the types once we put the module in a context. The role of this module is therefore merely to be a structuring component. Note that we have explicitly declared the two parameters *Transportation_System* and *Machine* in a module declaration box. Additionally the tags *PM* on two of the transitions are a graphical convention, and is a supplementary visual cue to the parameter declarations. We have also explicitly expressed that the three places *iBuffer*, *oBuffer*, and *Resource* are the net structure interface to the surrounding module by using the tagging notation of *In* and *Out*. The role of the in/out-tags is to help the user of this generic

module when building a manufacturing system. (See below where we compose the bottling manufacturing system.)

In the generic cell just described we have the *Machine* parameter. The generic machine module which we wish to use in our example is depicted in Fig. 12. It has the following formal parameters: *Functionality* (net structure

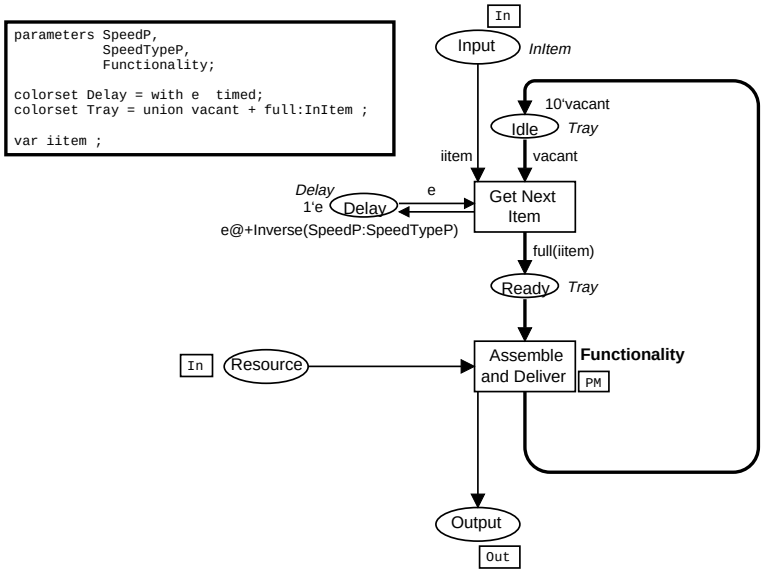

Figure 12: CPN model of the generic machine which is used by the generic manufacturing cell.

parameter), *SpeedTypeP* (type parameter), and *SpeedP* (value parameter). Only two colour set declarations are needed: *Delay* and *Tray*. The rest of the colour sets are inferred by the type checker and given a specific type upon instantiation. A type is inferred by the type checker for the value parameter *SpeedP*. In this case it is determined by the type of the function *Inverse* which again is determined by *SpeedTypeP*. The net structure parameter *Functionality* has the *PM* tag which indicates that it is a place-holder for a module. We do not describe the contents of the *Functionality* module.

Composing the Bottling Manufacturing System

Having made our main building block, the generic manufacturing cell, we can proceed with modelling the bottling manufacturing system itself. We make a number of specialisations of the generic cell and then build a manufacturing line from them. Then we encapsulate this in order to make a manageable parametrisation specification to the complete manufacturing system.

In order to build a bottling manufacturing line in Fig. 13 we need four variants of the generic cell from Fig. 11. Empty bottles enter the system and flow through the four cells with the following functions: first the bottles are filled up with a fluid (*Fluid bottling cell*), then lids are fitted on (*Lid fixing cell*) and labels pasted on (*Labelling cell*), and finally the bottles are packaged (*Packaging cell*) and sent out of the system. Each of the four stages are represented by

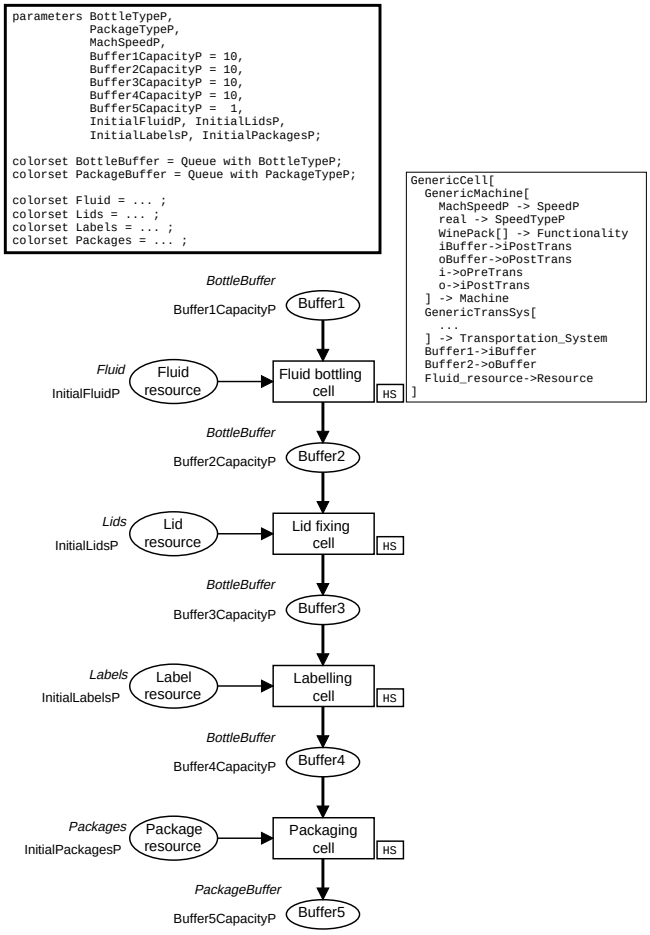

Figure 13: Partial CPN model of a simple bottling manufacturing system.

specialisations of the generic manufacturing cell. For instance, the specialisation to a fluid bottling cell can be seen next to the *HS*-tag of the *Fluid bottling cell* transition. In there we see all the assignments to the formal parameters and assignments of the interface places. Note that in order to assign a module to the machine parameter we need to make assignments to the formal parameters of the machine module:

```
GenericMachine[
  MachSpeedP -> SpeedP
  real -> SpeedTypeP
  WinePack[] -> Functionality
  iBuffer -> iPostTrans
  oBuffer -> oPostTrans
  i -> oPreTrans
  o -> iPostTrans
] -> Machine
```

This means: take the module *GenericMachine* (actual parameter) and assign it to the formal parameter *Machine*, but before doing that a number of parameters of the machine module need to be assigned. In *MachSpeedP* $\rightarrow$ *SpeedP* we take the value of *MachSpeedP* and assign it to the formal value parameter *SpeedP*, where *MachSpeedP* is itself a parameter of the bottling manufacturing system module. In *real* $\rightarrow$ *SpeedTypeP* we assign the type *real* to *SpeedTypeP*. In *WinePack[]* $\rightarrow$ *Functionality* we assign the module *WinePack[]*, which does not have any parameters, to the formal net structure parameter *Functionality*. The last four lines are assignments of the interface places.

The contents of the last three *HS* tags are similar to that of the transition *Fluid bottling cell*. Note the tag notation used for the four transitions are the same as those for substitution transitions in the tool Design/CPN.

Our bottling manufacturing system is almost complete. The final module we need to treat is the top-level module depicted in Fig. 14. The role of this module

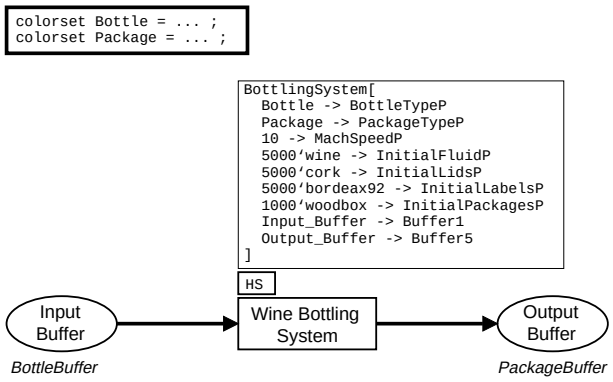

Figure 14: Top-level CPN module of the manufacturing system.

is to be a simple abstraction of the manufacturing system where only the most important formal parameters are visible. Thus this module provides a simple and easy to change interface to the system. Changing an actual parameter here does not require a full type check and compilation, but only a quick re-instantiation of the system.

In Fig. 15 we see the module overview page which is similar to the traditional hierarchy page of CP-nets. Each node represents a module and each arrow represents a relation between modules due to the assignments of net structure parameters. Some of the nodes (and arrows) are dotted. These represent external modules which needs to be imported from module libraries. Thus the dotted nodes represent modules which are not physically part of the main CP-net model which constitutes the solid graphics nodes. The external modules only get a transient physical representation when the system is instantiated for the purpose of execution.

5.2 Evaluation of Applicability

Below we summarise some of the techniques used in the example above and discuss their applicability. The use of parametrised CP-nets seems, as a side effect, to induce a number of other useful modelling techniques.

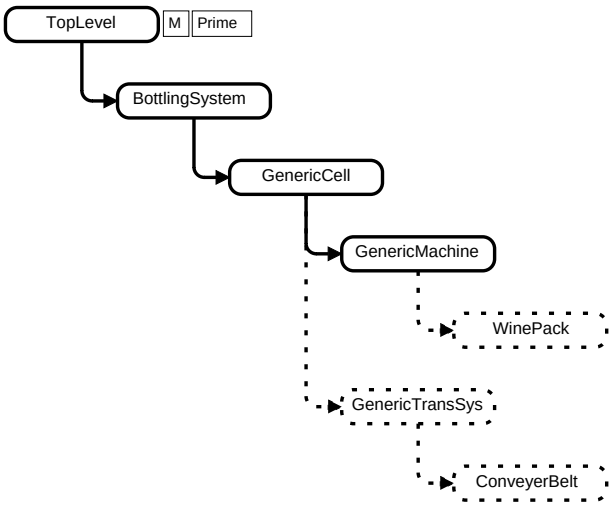

Figure 15: Module overview page. Dotted nodes represent modules which are imported from libraries.

In the example we illustrated the reuse technique where we reused the generic manufacturing cell in a number of specialisations to make the manufacturing system in Fig. 13. Using parametrised modules for this purpose is a flexible technique which would be difficult with hierarchical substitution transitions only. In Fig. 15 we indicated that external library modules also would benefit to the reuse of parametrised modules.

Figure 15 is used to show module dependencies for a specific instantiation. Actually the arrows between *TopLevel*, *BottlingSystem*, and *GenericCell* are essentially the hierarchical substitution relation. The rest of the arrows are a result of the net structure parameter assignments in Fig. 13. If the user edits the formal parameters of the parameter assignments, then the module dependency page may change appearance. We do not need to distinguish (graphically) between the two kinds of relations, hierarchical substitution and parametrised modules, because they are in essence the same.

Even for this relatively small example we observe that there are quite a few formal parameters. As a result we see, e.g., in Fig. 13 that the assignment notation may quickly become rather large and therefore complicated to look at. This indicates that the user interface scenario explored so far may not be adequate for handling larger examples. Thus we suggest that there should be made further investigations in this area to make parameter assignments more scalable.

6 Implementation Issues

In Sect. 3 we have proposed a conceptual framework for parametrised CP-nets. Although we provide sufficient details such that the framework can be used as a rough recipe for implementation, we have not conveyed all useful ideas. The design ideas are somewhat biased with a target tool and implementation lan-

guage in mind. They are respectively Design/CPN and SML. Design/CPN has an interface look-and-feel tradition, and SML does contain a number of useful language constructs such as module structuring features with parametrisation. As the tool and implementation language is fixed in this paper it also makes sense to describe a few practical restrictions imposed by these choices.

6.1 Parametrised Modules

The language SML has a module mechanism which is called *structures*. A structure can be parametrised, and such a construct is called a *functor*. In the following we outline that the modules system of SML, i.e., structures and functors, are sufficient for our purposes of implementation. The SML example below is inspired by the generic machine module from Fig. 12. This module has three formal parameters *SpeedP* (value), *SpeedTypeP* (type), and *Functionality* (net structure). In SML we first declare a couple of useful module interfaces (called signatures):

```
signature FUNCTIONALITY
= sig
  ...
end;

signature GENERICMACHINE
= sig
  ...
end;
```

The purpose of these is to specify more exactly what we allow to be used as net structure parameters for *Machine* and *Functionality*. With these signatures we can now declare the generic machine module with an SML functor:

```
functor GenericMachine
         (type SpeedTypeP
          val SpeedP:SpeedTypeP
          structure Functionality:FUNCTIONALITY):GENERICMACHINE
= struct
  ...
end;
```

Before we can instantiate the generic machine module we need first a module to be assigned to the net structure parameter *Functionality*, which we call the *WinePack* module (a component which can package wine bottles):

```
structure WinePack
= struct
  ...
end;
```

Now that we have the generic machine functor and a module, *WinePack*, we can then instantiate a machine such that an executable machine module can be generated:

```
structure aMachine = GenericMachine
                        (type SpeedTypeP = real
                         val SpeedP = 10.0
                         structure Functionality = WinePack);
```

The SML code above should, of course, be generated automatically by the tool. Throughout the whole process of declaring modules and instantiation we are helped by the strongly typed language of SML. If we make a mistake the SML compiler will report an error. Thus we conclude that SML is a potentially appropriate implementation language for parametrised CP-nets.

7 Future Work

In this section we provide an overview of activities we wish to be a continuation of this work. Below we discuss future work in the area of parametrised CP-nets, implementation work, and related activities. Additionally we propose directions in the important area of validation and verification.

7.1 Parametric CP-nets

In Sect. 3 we have provided a conceptual framework for parametrised CP-nets. The purpose is to provide a preliminary framework for further work. The next step is to apply the current framework on a much larger example. We have presented many design ideas which much be evaluated in the context of realistic case studies.

Another future important step is to make a formal model of parametrisation in CP-nets, which we refer to as *Parametric CP-nets*. It is important because a formal model is a fundamental contribution which can be used as a reference. Such a reference is necessary when ambiguities need to be resolved, and can also be very helpful during implementation of a tool — here the integration into Design/CPN. A formal model is also necessary when studying parametrisation of analysis methods. Our hope is that a formal model for Parametric CP-nets can unify the three kinds of parametrisation we have studied here: value, type, and net structure parameters. Although the tool user does not need to know of this level, a unification may result in a simpler and more general formal model and potentially a simpler implementation.

In this conceptual framework we restricted net structure parameters to be on the level of transitions. Naturally we should also consider the case of letting places be parameters. This is analogous of considering both substitution transitions and places as with the original formal model of CP-nets. We expect that net structure parameters on the level of places is very similar to the case of transitions being parameters, and we do not see any serious problems with having both in the same framework. The two concepts are in essence dual, and they are both useful from a modelling point of view. Additionally, it could also be interesting to investigate if arcs could be used as a syntactical category for the source of net structure parametrisation.

We have been somewhat inspired by the implementation language SML, but we have also made limitations due to SML. Our inspiration has been influenced by the module feature of SML which allows the same three kinds of

parametrisation as those we treat in this paper. SML is a strongly typed functional language which features parametric polymorphism [3]. Other languages, especially object-oriented, features inclusion polymorphism, and virtuals such as BETA [14]. In BETA it is possible to use virtual classes for supporting parametrised representations. As the mechanism of virtuals is very flexible we suggest to investigate how virtuals could be realised, if possible at all, in CP-nets. We do not know of any work in that direction within the research area of Petri Nets.

7.2 Implementation in the Tool Design/CPN

We need tool support for parametrised CP-nets in order to learn more about pragmatic issues on parametrised representations. In Sect. 6 we have shown that the target language, SML, in principle is sufficient for our needs.

We have claimed that parametrised CPN models should facilitate quick and easy instantiation of parametrised modules in order to support an environment for building many variants of models. Therefore we need a useful user interface to building models with modules and instantiation. We can use a scripting language in order to solve this issue. Such a scripting language should support iteration over parameters of all kinds; value, type, and net structure. In fact, the language SML is already suitable for such a purpose. For instance, suppose we wish to study how our manufacturing system performs by varying the machine speed parameter value. Then we just write a script that can make a large number of instantiations with different speed parameter values.

7.3 Enhancing Expressive Convenience

In the examples presented in this paper we have seen that we could leave out many type inscriptions and therefore leave it to the type system to infer the most general type. Furthermore, we have only considered parametric types [3] due to the choice we made in advance about the target implementation language, SML. Although not supported directly by SML we could also consider to include the possibility of inclusion types, i.e., we should consider to introduce concepts from the research area of object-orientation. Many people working with Petri Nets already do research on different kinds of object-oriented Petri Nets [2].

7.4 Validation and Verification

CP-nets have very powerful and general analysis methods, i.e., methods for obtaining answers to questions about the behaviour of CPN models. Usually analysis methods are divided into two groups, namely validation and verification methods. Validation is concerned with convincing ourselves that a CPN model behaves as intended, while verification is concerned with proofs or algorithmic checks that a CPN model has a formally stated property. Simulation is a typical validation technique, and two of the most popular and successful verification methods are state spaces and place invariants [10].

Let us first consider validation techniques on parametrised CP-nets, more specifically simulation. The characteristics of a module is that it is mostly a self-contained unit, which is loosely coupled with its environment. This means that we can expect that a module is primarily developed independently. How

do we validate a parametrised module via simulation? Surely this is what we would like to do in early stages of development. The way we have designed parametrised CP-nets so far does not allow us to simulate a parametrised module without instantiation. In Sect. 8 on related work we refer to other research on parametrisation (although within object-orientation) which allows execution of parametrised modules without assignment of parameters [16]. We suggest to study that digression in order to investigate alternatives.

There are a number of successful verification methods for CP-nets. Below we emphasise two of the most well-known, namely the state space and invariant methods. The state space method relies heavily on the initial marking of the CPN model in question. Thus the method cannot be applied directly on the level of our parametrised CPN models, as they are not executable without instantiation of parameters. However, there are other methods in relation with state spaces where parametrisation has been used. In [18] by Schmidt, a symbolic state space method is applied to marking parametrised algebraic Petri Nets. Although the theoretical results are interesting, the work still lacks an implementation.

While the state space method belongs to the model checking area, the invariant method is more related with the area of theorem proving. Theorem proving is often abstract in a mathematical sense which implies manipulation of formulas on a symbolic level. Therefore we can expect that the invariant method is more compatible with parametrised CP-nets. Although net structure parameters in general violate invariant properties we may have more success with value parameters as they can be used as terms in weight sets and the invariant properties themselves. Type parameters do not influence the invariant properties directly, but rather determine the types of the weight functions as these are derived from the types (colour sets) of the places. We additionally suggest to make investigations on how to build parametrised representations with analysis in mind.

Another analysis approach would be to combine a number of well-known techniques and methods. There are examples of combining methods and techniques in the literature. For instance, Shapiro et al. [8] has combined induction with the state space method in the verification of an arbiter cascade CPN model. The arbiter cascade is a tree structure of hardware components and the verification was conducted with induction in the depth of the tree. Although the model cannot be parametrised within our framework of parametrised CP-nets, we can still in principle use this idea — in particular with (integer) value parametrisation. We may even apply the more general kind of induction called *well-founded* induction. Further investigations in such combinatory approaches would be interesting in the search for new analysis methods and techniques.

As indicated above there are other activities in progress for the development of analysis methods in relation with parametrised Petri Nets. We also indicated that the current framework in this paper of parametrised CP-nets is informal, suggesting that the framework is not necessarily very useful in relation with generalising analysis methods such as the state space and invariant methods. Therefore we suggest to investigate further the issue of restricting the conceptual framework presented here such that the popular analysis methods can be generalised in order to cope with parametrised CP-nets. This should in particular be considered when making a formalisation of parametrised CP-nets. Another approach is to extend the well-known analysis methods. For instance, one could

consider to make semantical annotations on various places in a parametrised CPN model, which then could be used as assumptions in relation with an analysis method such as invariants. These semantical annotations could be given, e.g., to each net structure parameter which then should mean that there would be further restrictions on which parametrised modules that could be used for assignment.

8 Related Work

Chiola et al. [5] define a formal model for Parametric PT-nets. Their formal model is restricted to parametrisation of initial markings, i.e., the parameters are integers. The purpose of their paper is to compare the modelling power of several variants of PT-nets within the framework of Parametric PT-nets. As they formalise Parametric PT-nets we acknowledge that their work is in some sense more rigorous compared with our framework which is informal. However, we cannot compare the results directly as both the purposes and net kinds are different. Chiola et al. identifies that it is in some cases possible to reason about net properties on the level of Parametric PT-nets, i.e., instead of analysing a single system they analyse a family of systems. The family is determined by the parametrised initial markings. One of the more interesting analysis methods they consider is the invariant method.

Another Petri Nets language which supports parametrised representations is the ExSpect framework [21]. This framework is interesting because it is related with the CPN formalism and the framework supports the same three kinds of parametrisation as in this work — in their terminology; functions, types, and processors/subnets. However, the parametrisation concept is not built into the ExSpect formalism, only in their tool. Like our work with parametrisation of CP-nets, the ExSpect framework needs to formalise parametrisation in order to get an unambiguous semantics. However, they already have the advantage of having implemented parametrisation in their ExSpect tool — which we have not. Additionally they have not made any work on analysing parametrised ExSpect representations.

Some object-oriented languages have parametrisation capabilities. One of these is BETA [14]. This language indirectly supports parametrisation with a language construct called virtual classes. It is a very general construct which is also used for expressing other mechanisms than parametrisation. The authors of [16] introduce an interesting idea of type substitution (a kind of genericity) in object-oriented languages which then works as parametrisation. A parametrised class can in this case be instantiated without the need to supply parameters. The type names are already parameters and are thus already legal types. We do not use this approach in parametrisation of CP-nets, although it is possible in principle. One reason is that we are from the beginning influenced by the target implementation language SML which does not support type substitution (or object-orientation for that matter).

The SDL language [19], which is a recommended telecommunications standard, share some characteristics with Petri Nets. The language is graphical and has some kind of state/transition concept. The original standard, SDL'88, was extended with object-oriented concepts [13] and later the SDL'92 [20] was proposed, also called OSDL. We take interest in OSDL because the language sup-

ports concepts such as virtuals and parameters, and is furthermore interesting for people working with object-oriented Petri Nets. The parameters supported are values, types, and processes which are similar to the net structure parameters in this paper. Parametrised SDL modules cannot be executed without supplying parameters, however SDL modules with virtuals can. In spite that OSDL is executable and tools have been made to support code generation from OSDL representations, there are currently no advanced and general purpose verification methods such as the invariant method for CP-nets. Some SDL tools do, however, use state spaces in limited fashions.

In this paper we do not treat the issue of object-orientation or virtuals together with Petri Nets. There are many other people working with introducing object-oriented concepts into Petri Nets [2, 1]. None of them, however, consider parametrisation in their own variants of Petri Nets. The research on object-oriented Petri Nets is very active, but no common directions or agreement on object-oriented Petri Nets have been concluded yet.

9 Conclusion

In this paper we have investigated the possibility for the parametrisation of CP-nets. To support this idea we have provided a conceptual framework for parametrisation which resulted in the investigation of three concepts for parametrisation; namely value, type, and net structure parametrisation. By means of examples we have indicated that value and type parametrisation is straightforward while net structure parametrisation is more complicated. The latter induced the need for net structure parametrisation by means of modules, which implied a relation between modules — a concept supplementary to the hierarchical substitution relation of CP-nets.

We saw that the introduction of modules into CP-nets lead naturally to the need for scope rules for declarations such as colour sets. In the same context we generalised the existing scope rules for place fusion groups, where we introduced the concept of topological scope rules.

This conceptual framework is being considered a preliminary stage in the parametrisation of CP-nets. We propose first to build tool support for parametrised CP-nets, and as a later stage when the design ideas have mature, to realise the formalisation of *Parametric* CP-nets.

Acknowledgements

Kurt Jensen has commented on earlier versions of the present paper. This work has been supported by a grant from the Danish Research Council SNF.

References

[1] G. Agha, F. de Cindio, and A. Yonezawa, editors. *Workshop on Object-oriented Programming and Models of Concurrency of the 17th International Conference on Application and Theory of Petri Nets*, Osaka, Japan, 1996.

[2] G.A. Agha and F. de Cindio, editors. *Proceedings of Workshop on Object-Oriented Programming and Models of Concurrency*, Torino, Italy, 1995. <URL: http://wrcm.dsi.unimi.it/PetriLab/ws95/home.html>.

[3] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[4] Allan Cheng, Søren Christensen, and Kjeld H. Mortensen. Model Checking Coloured Petri Nets Exploiting Strongly Connected Components. In M.P. Spathopoulos, R. Smedinga, and P. Kozák, editors, *International Workshop on Discrete Event Systems, WODES96*, pages 169–177, Edinburgh, Scotland, UK, August 1996. Computing and Control Division, Institution of Electrical Engineers.

[5] G. Chiola, S. Donatelli, and G. Franceschinis. On Parametric P/T nets and their Modelling Power. In *Application and Theory of Petri Nets, 12th International Conference*, pages 206–227. IBM Deutschland, 1991.

[6] Design/CPN Online. WWW. <URL: http://www.daimi.aau.dk/design-CPN/>.

[7] F. DiCesare, G. Harhalakis, J.M. Proth, M. Silva, and F.B. Vernadat. *Practice of Petri Nets in Manufacturing.* Chapman and Hall, 1993.

[8] H.J. Genrich and R.M. Shapiro. Formal Verification of an Arbiter Cascade. In K. Jensen, editor, *Proceedings of the 13th International Conference on Application and Theory of Petri Nets, Sheffield, UK*, volume 616 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[9] K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, 1992.

[10] K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science. Springer-Verlag, 1994.

[11] K. Jensen, S. Christensen, P. Huber, and M. Holla. *Design/CPN Reference Manual.* Computer Science Department, University of Aarhus, Denmark, 1996. Available from [6].

[12] J.B. Jørgensen and L.M. Kristensen. *Design/CPN OE/OS Graph Manual.* University of Aarhus, Computer Science Department, Denmark, 1996. Also available via [6].

[13] J.L. Knudsen, M. Löfgren, O.L. Madsen, and B. Magnusson. *Object-Oriented Environments, the Mjølner Approach*, chapter 8: Rationale on object-oriented SDL. Prentice Hall, 1994.

[14] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison Wesley, 1993. World-Wide Web: <URL: http://www.mjolner.com/>.

[15] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[16] J. Palsberg and M.I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley and Sons, 1994.

[17] L.C. Paulson. *ML for the Working Programmer (2nd Edition).* Cambridge University Press, 1996.

[18] K. Schmidt. Parameterized Reachability Trees for Algebraic Petri Nets. In Giorgio De Michelis and Michel Diaz, editors, *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*, pages 392–411, Turin, Italy, June 1995. Springer-Verlag. LNCS 935.

[19] CCITT, Specification and Description Language SDL, Recommendation Z100–Z104, ITU, 1988.

[20] CCITT, Specification and Description Language SDL, Recommendation Z100–Z104, ITU, 1992.

[21] K.M. van Hee, L.J. Somers, and M. Voorhoeve. Executable Specifications for Distributed Information Systems. In E.D. Falkenberg and P. Lindgreen, editors, *Proceedings of the IFIP TC 8 / WG 8.1 Working Conference on Information System Concepts: An In-depth Analysis*, pages 139–156, Namur, Belgium, 1989. Elsevier Science Publishers, Amsterdam.