

Requirements of a Module System for Coloured Petri Nets

Thomas Mailund

11th May 2000

Today, no one would even consider modelling industrial sized systems as a single component. Such a task would be too cumbersome, too error prone, and the result would be nearly unmaintainable. The lesson learned from high level programming languages is that complex programs should be constructed by composing simpler modules, which in turn could be constructed from even simpler modules, until the modules finally have a manageable complexity. Working with coloured Petri nets should be no different. In this report we examine the requirements of a module system for CP-nets. We list what we find to be the minimal requirements of a module system and illustrates these requirements through examples.

1 Requirements of a CPN Module System

The exact requirements of a module system will vary with the specific definitions of the nets and with the different uses for nets and module system. It is not our intention to discuss every aspect of module systems here, but simple to list what we find to be general requirements which should be satisfied by any module system for CP-nets. As a minimum, the following requirements should be satisfied:

- Modules should be autonomous and should distinguish between interface and implementation
- A module system should allow compositional module construction
- A module system should be intuitive
- Modules should be highly reusable
- Modules and their composition should have a well-defined semantics allowing formal analysis

1.1 Modules Should Be Autonomous

We build modules to be able to work on parts of models independently. When working on one module, we want to ignore all other modules, or at least their detailed behavior. The behaviour of one module should not depend on subtle details in other modules, and editing internal details one module should not cause

changes in other modules. To quote Christensen and Mortensen [2]: “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.”

The behaviour of a module should be apparent from the module’s specification. For a module system to be useful, it is necessary for modules to communicate. Thus, it is unavoidable that the behaviour of the individual modules will depend on the environment in which the module is placed. As long as this communication takes place through a well-defined interface this is not contrary to the demand of autonomous modules. The point is not to isolate modules, but to make sure that modules can be understood independently, and that their isolated behaviour to a large extent matches their behaviour in any environment they are put.

As an example, consider a module system where all modules in a composition share the same set of declarations. This is similar to hierarchical as defined by Jensen [5] and as implemented in the tool Design/CPN [1], where all modules share a single set of declarations. In such a setting the individual modules are not autonomous. Not only the behaviour, but also the correctness of a module depend on the declarations outside the module. Moving a module from one model to another can cause problems due to changing declarations.

Consider the net in 1. Here constraints in the definition of CP-nets require that a is a variable over the set A , b is a variable over the set B , and that $f : A \times B \rightarrow C_{MS}$ is a function from the product of A and B to the multi sets over C . Thus with the declarations in 2 the net is a legal net while with the declarations in 3 the net is not. For modules to be autonomous, all variables, values, and types used in the modules should be declared local to the module, or be parameters of the module.

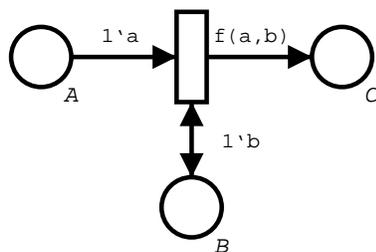


Figure 1: Simple net without declarations. The declarations are assumed to be given elsewhere.

A module system should allow for both bottom-up and top-down development. To achieve this goal a module system should clearly differentiate between the inner workings of a module and the interface between a module and its environment. Composition of modules alone allows only bottom-up development. We can create modules and combine them into other modules. But without putting some restrictions on the composition of modules, modifying one module can require extensive changes in other modules. By clearly distinguishing between

```

colourset A = int ;
var a : A ;
colourset B = real ;
var b : B ;
colourset C = product int * real ;
fun f (x:int, y:real) = 1'(x,y)

```

Figure 2: Legal declarations for the net in 1.

```

colourset A = int ;
var a : A ;
colourset B = real ;
var b : B ;
colourset C = int ;
fun f (x:int, y:real) = x+y ;

```

Figure 3: Illegal declarations for the net in 1.

the interface and the inner workings of a module we open up for incremental refinement of modules. It is then possible to create abstract modules, fix their interface to their environment, and compose these modules. Later on, the abstract modules can be replaced with refined versions, and as long as the new modules respects the interface changing one module should not require changes to other modules.

The net in 4 models a lossy network. Packets delivered to the module are either transferred across the network, with a probability taken from the place **SuccessRate**, or lost. Say we no longer want the success rate to depend on a token value, but instead be a constant. We can achieve this by changing the net to that shown in 5. However, another module might access the place **SuccessRate** to update the success rate depending on network load or crashing servers. When we remove the place, this module breaks.

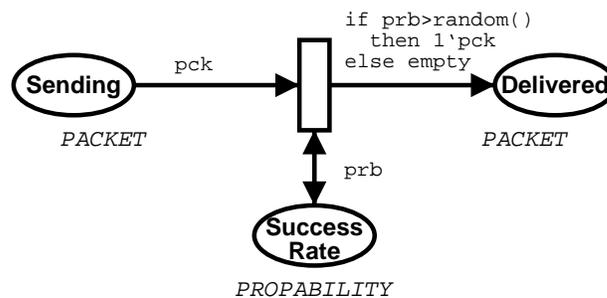


Figure 4: Packets arrive on the left and with a certain probability, read from the place **SuccessRate**, are transferred to the left.

If the entire module can be accessed by any other module, we cannot change modules without risking breaking other modules. To make sure we can change parts of a module independent of other modules, we must encapsulate these

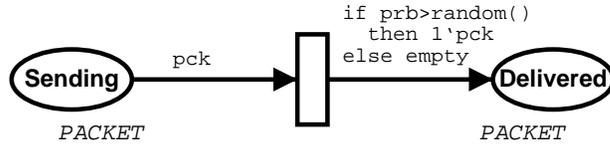


Figure 5: Packets arrive on the left and with a certain probability, declared as the constant `prb`, are transferred to the right.

parts. The net in 6 hides the transition and the place `SuccessRate` from all other modules, and only exposes the places `Sending` and `Delivered`.

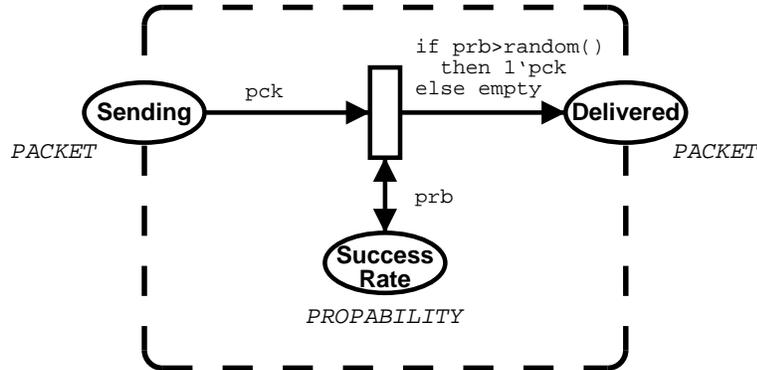


Figure 6: A network module with the interface consisting of the two places `Sending` and `Delivered`.

Of course, distinguishing between internal and external nodes only ensures a structural consistency between modules. There are no guarantees about behavioral properties. To achieve results in that direction one could extend the interface to contain behavioural information.

1.2 Compositional Module Construction

Composition of modules is really the essence of a module system. The difference between a set of individual nets, and a set of modules, is that the modules can be composed to create new modules. It is important that the composition of modules result in a module. Otherwise module reuse is limited to simple components while complex constructions need to be redone each time they are needed, which would limit the usability of the module system.

This differs from what is found in hierarchical CPN. In hierarchical CPN a number of nets (pages) are combined into a hierarchical net, but two hierarchical nets cannot in turn be combined into another hierarchical net. There is a distinction between non-hierarchical nets, the pages, and the composed net, the hierarchical net.

Assume that we wish to build a model of a stop-and-wait protocol. We can build this out of three components: the underlying protocol layer, which can

be a lossy network as in 7; a sender as in 8; and a receiver as in 9. A model constructed in this way is shown in 10. Clearly, a stop-and-wait protocol can be a useful component in a model of a layered protocol. It should be possible to use the constructed stop-and-wait protocol directly in such a model. It is not acceptable to be forced to re-create the composition from the sub-modules each time the stop-and-wait protocol is used. Though it does not look that intimidating with only three sub-modules, consider that in turn a layered protocol can be a useful component in a model of a distributed application such as a file server. So in modelling a file server one would first need to re-construct the stop-and-wait protocol and then reconstruct the layered protocol, which in all likelihood will consist of more modules than just the stop-and-wait module. And a file server is just one of many distributed application which could make use of a layered protocol module.

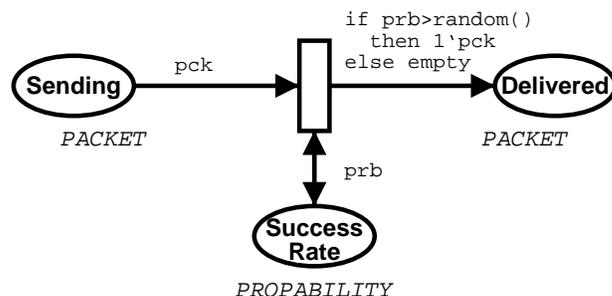


Figure 7: Packets arrive on the left and with a certain probability are transferred to the left.

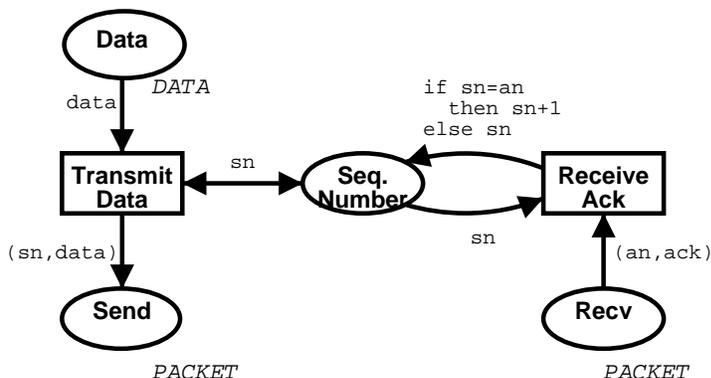


Figure 8: Data arrive from a higher layer (the place **Data**) and is transmitted with a sequence number. When an acknowledgement arrive the sequence number is incremented and data is send with the next number.

1.3 A Module System Should Be Intuitive

With a module system, we try to achieve a compositional approach to modelling. We split up a complex model in smaller, less complex chunks, the modules,

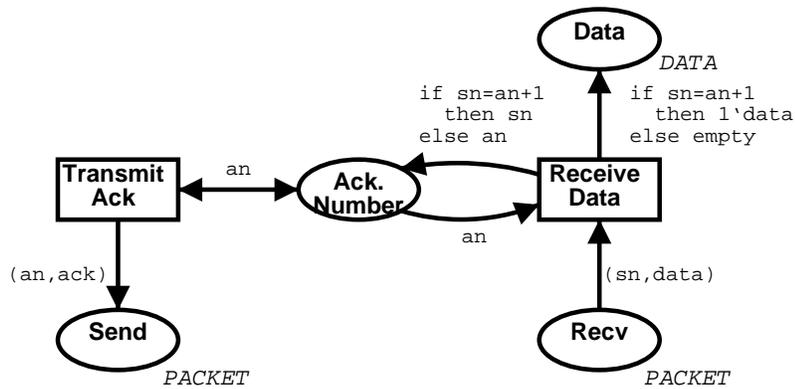


Figure 9: Data packets arrive and, if the sequence number is right, transferred to a higher layer (the place **Data**). Acknowledgments are then returned to the sender.

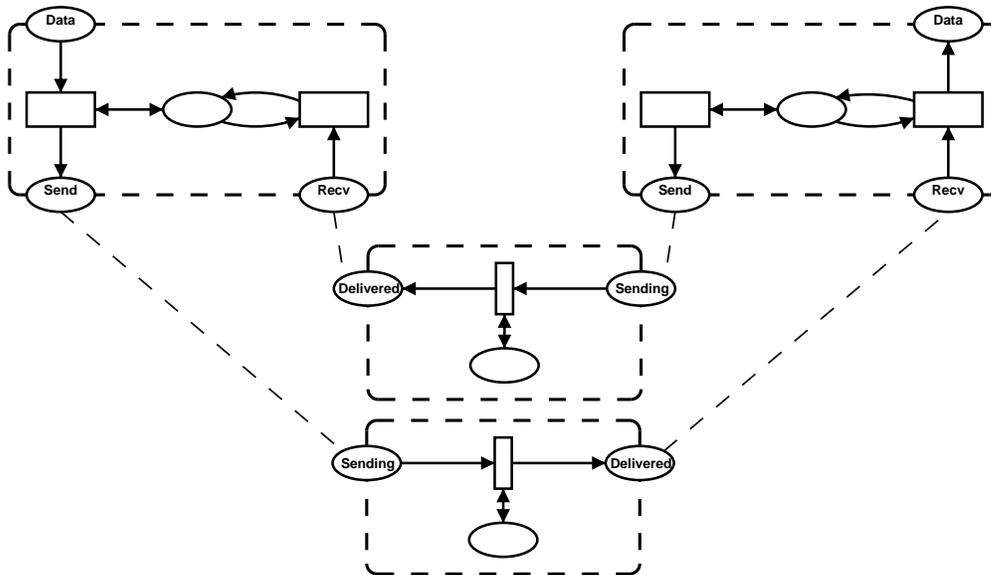


Figure 10: A stop-and-wait model composed by the sender module; the receiver module; and two copies of the network module, one for each direction of packet transmission. The place **Send** in the sender module is fused with the place **Sending** in one of the network modules, while the place **Recv** in the sender is fused with the place **Deliver** in the other network module. Similarly for the receiver module.

recursively, until we can handle the complexity. The way a module is split into sub-modules, however, should not be arbitrary. The split should be done in a way that reflects our understanding of the modelled system. If not, this fragmentation will do more harm than good.

A module system should not force us to create counter-intuitive models. Neither should they force us into cluttering our models with “technical hacks” nor force us into modelling details not relevant to our intended level of abstraction.

Consider the net in 11. The net contains an entire producer/consumer system in a single component. On the left is shown the producer subnet. The subnet models a number of producers, folded in the colour set PROD producing a number of different items, folded in the colour set ITEM . The producers produce items, modelled by the transition \mathbf{P} , and write them, transition \mathbf{W} , into a buffer, the place \mathbf{B} . On the right the consumer subnet is shown. A number of consumers, folded in the colour set \mathbf{C} removes items from the buffer, modelled by transition \mathbf{R} and consume them, modelled by transition \mathbf{C} .

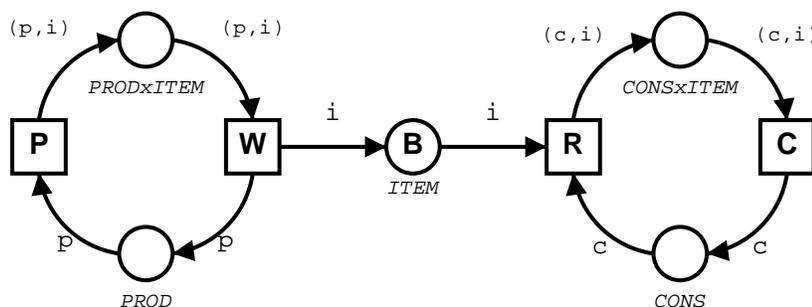


Figure 11: On the left, a number of producers produce items and store them in the buffer in the middle. On the right a number of consumers remove the items from the buffer and consumes them.

An obvious modularisation of the producer/consumer net in 11 is splitting the net into a producer module and a consumer module. Now, assume that the way we can connect modules are through the familiar arcs, i.e. edges connecting a place and a transition. Since the producer module should contain all information about the producers, and since the consumer module should contain all information about the consumers, the split would have to go through one of the two arcs going to or from the buffer node \mathbf{B} .

One of the splits is shown in 12 and 13. In this particular split the buffer was placed in the producer module. The buffer might as well have been placed in the consumer module. The decision to put it in the producer module was arbitrary. An arbitrary decision we should not be forced to make. The buffer is as much a part of the consumers as it is a part of the producers. To leave it out of one or both of the modules is counter-intuitive. To allow the buffer to be present in both modules we need to resolve to technical “hacks” as in 14.

A better approach, in this example, would be to allow place-sharing between modules, e.g. in the sense of place fusion. This is illustrated in 15. Many

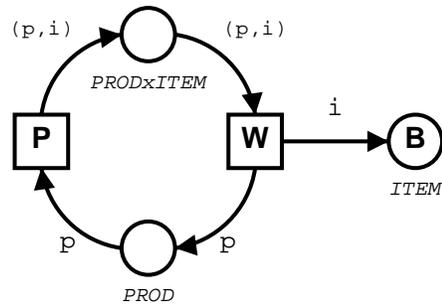


Figure 12: The module contains the left hand side of the net in 11 together with the place **B** modelling the buffer.

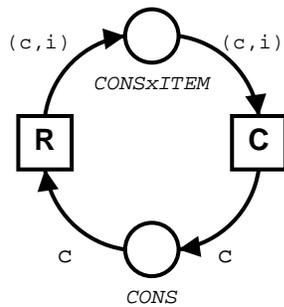


Figure 13: The module contains the right hand side of the net in 11.

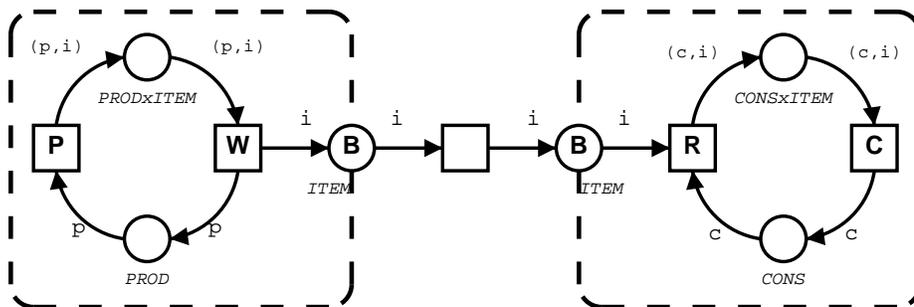


Figure 14: On the left is shown the producer module from 12. On the right is shown the consumer module from 13 extended with a place modelling the buffer. In the middle a transition has been added to allow the buffer to be present in both modules.

analysis techniques require either exclusive synchronous or asynchronous communication between modules. Restricting modelling in this way, however, is counter-intuitive, and should therefore be avoided.

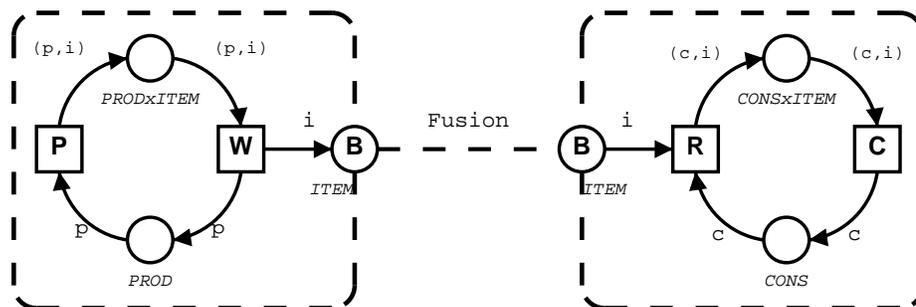


Figure 15: On the left is shown the producer module from 12. On the right is shown the consumer module from 13 extended with a place modelling the buffer. The two port places are fused.

1.4 Modules Should Be Highly Reusable

A very important gain from a modular approach to modelling is reusability. A large degree of reuse significantly decreases the development time and reduces maintenance cost. Without modules the only way to reuse a certain net construction is to make several copies of the construction. With modules, commonly used constructions only need to be created once, and can then be reused throughout the model. Furthermore, modules can be compiled into libraries of commonly used constructions that can be used in several modelling projects.

A module system should promote reusability. As mentioned above in 1.3 modules should be built to reflect the modelled system. The module system, then, should be flexible enough to allow different uses of the modules.

Say we have created a module modelling an event triggered function, for example the visual feedback of a GUI button when the button is “pressed”. The interface of this module is the transition modelling the triggering event. Such a module is shown in 16. This module is very general and we should be able to use it in various different settings without modifications. 17 shows the module used in a GUI framework where user actions are immediately dispatched to the relevant components while 18 shows the module in a framework based on an event queue.

Another way a module system could achieve higher levels of reusability is through parameterisation. It is often the case that models contain a number of similar constructions that only differ in minor ways. A module system aimed at reusability would allow the modeller to describe families of modules rather than specific instances. With parameterised representations of modules, that can be instantiated and specialised into concrete modules through the module

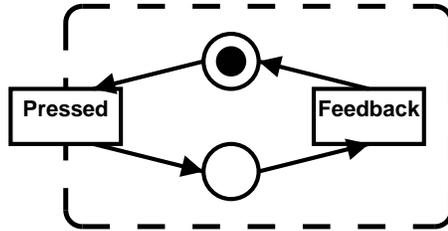


Figure 16: A model of the visual feedback triggered when a GUI button is pressed. The interface of the module is the transition **Pressed** modelling the triggering event. Other components can trigger the action by firing the transition.

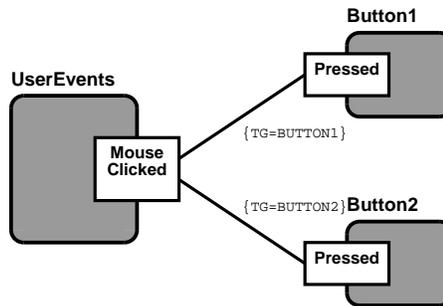


Figure 17: User-actions are immediately translated into GUI events, and propagated to the right widget. Here pressing GUI buttons is modelled by the synchronisation of the transition **Mouse Clicked** with the transition **Pressed** in either of the two buttons, depending on transition group. (Here we assume a language construction similar to the transition groups described by Christensen and Petrucci [3, 4].)

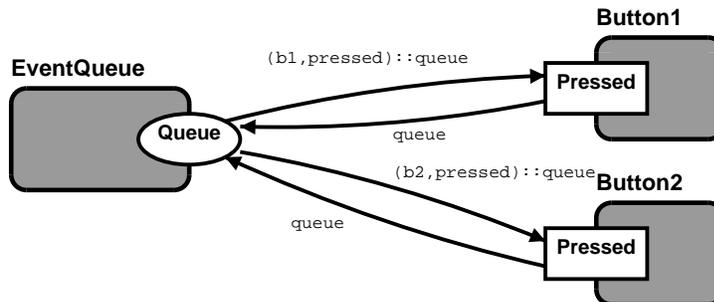


Figure 18: As user-actions take place, events are queued up. The events are then taken off the queue one at a time and triggers the listening widget.

system, the need for several nearly identical copies vanishes and a single module can be used.

We return to the network module we have seen earlier. The behaviour of a network usually does not depend on the data transferred. The behaviour is decided by headers attached to packets, but not the data itself. In the model of the network we do not examine the value of the variable `pck` containing the data. Nevertheless, to simulate the module we need to have a type for the data, and this type should match the type used by the higher layers in the protocol or the endpoints of the protocol. If the two endpoints of the network communicate strings, then the packet type must be string (see 19).

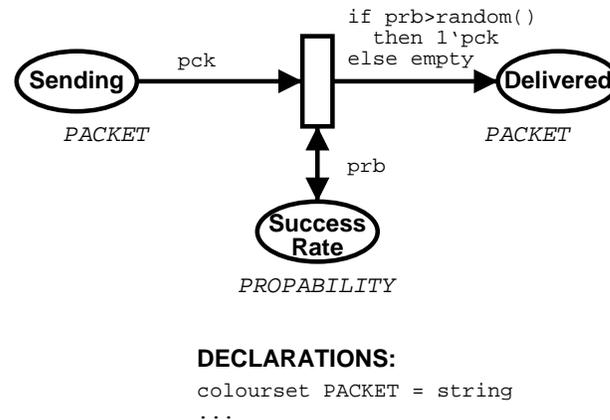


Figure 19: A network module that transmit string packages.

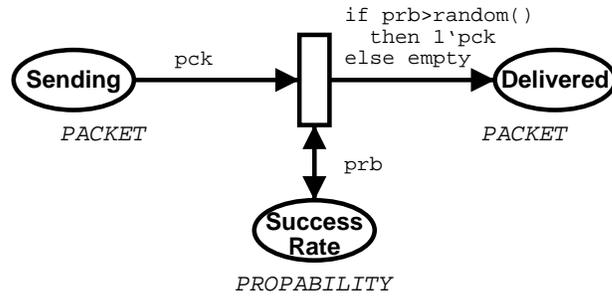
In a model which uses different networks to transmit different types of data, e.g. strings, integers, database records, this forces us to have several copies of the network module; one for each packet type. This is clearly not satisfactory. It would be far better having the packet type as a parameter to the module, as shown in 20. In that case there is only one copy of the module, that get instantiated with different packet types as needed.

For further discussion of parameterisation of coloured Petri net modules we refer to [2, 6].

1.5 Well-defined Semantics

A well-defined semantics is hard to argue against. Without the mathematical basis giving a net its meaning, its behaviour, a net is just a drawing. The strength of Petri nets comes from the underlying mathematical basis which makes precise the meaning of models, which permits us to create executable models for simulation, and which makes possible the number of verification methods on nets.

A module system of nets should have a clear unambiguous semantics as well. When we combine modules, we need to clearly specify the behaviour of the new



PARAMETERS:
 colourset PACKET
 ...

Figure 20: A network module that transmit packages of a parameterised type. To use the module within a model the module should be instantiated with a specific type.

module created. Combined modules should result in executable models, just as their individual components, and it should be possible to analyse modular models mathematically. Preferably this semantics should permit compositional reasoning about modules.

References

- [1] *Design/CPN*, <http://www.daimi.au.dk/designCPN>.
- [2] S. Christensen and K.H. Mortensen, *Parametrisation of Coloured Petri Nets*, DAIMI-PB 521, Department of Computer Science, University of Aarhus, 1997.
- [3] S. Christensen and L. Petrucci, *Towards a Modular Analysis of Coloured Petri Nets*, Proceedings of ICATPN 1992, LNCS, no. 616, Springer-Verlag, 1992, pp. 113–133.
- [4] S. Christensen and L. Petrucci, *Modular State Space Analysis of Coloured Petri Nets*, Proceedings of ICATPN'95, LNCS, vol. 935, Springer-Verlag, 1995, pp. 201–217.
- [5] K. Jensen, *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. - Volume 1: Basic Concepts*, Springer-Verlag, 1992.
- [6] T. Mailund, *Parameterised Coloured Petri Nets*, Proceedings of CPN 1999, DAIMI-PB, no. 451, 1999, pp. 133–151.