A Brief Introduction to Coloured Petri Nets

Kurt Jensen

Computer Science Department, University of Aarhus
Ny Munkegade, Bldg. 540, DK–8000 Aarhus C, Denmark
E-mail: kjensen@daimi.aau.dk
WWW: http://www.daimi.aau.dk/~kjensen/

Abstract. Coloured Petri Nets (CP-nets or CPN) is a graphical oriented language for design, specification, simulation and verification of systems. It is in particular well-suited for systems in which communication, synchronisation and resource sharing are important. Typical examples of application areas are communication protocols, distributed systems, imbedded systems, automated production systems, work flow analysis and VLSI chips.
The development of CP-nets has been driven by the desire to develop a modelling language — at the same time theoretically well-founded and versatile enough to be used in practice for systems of the size and complexity we find in typical industrial projects. To achieve this, we have combined the strength of Petri nets with the strength of programming languages. Petri nets provide the primitives for the description of the synchronisation of concurrent processes, while programming languages provide the primitives for the definition of data types and the manipulation of data values.

1 Introduction to Coloured Petri Nets

A small example of a CP-net is shown in Fig. 1. It describes a simple transport protocol transferring a number of packets over a unreliable network from a sender to a receiver. The ellipses and circles are called **places**. They describe the states of the system. The rectangles are called **transitions**. They describe the actions. The arrows are called **arcs**. The **arc expressions** describe how the state of the CP-net changes when the transitions occur.

Each place contains a set of markers called **tokens**. In contrast to low-level Petri nets (such as Place/Transition Nets), each of these tokens carries a data value, which belongs to a given **type**. As an example, place *Send* (in the upper left corner of Fig. 1) has seven tokens in the initial state. All the token values belong to the type *INTxDATA* and they represent seven packets which are ready to be sent. The first element in the pair is the packet number, while the second is the data contents of the packet. All the 1's in front of the small backslashes indicate that there is exactly one of each of the seven packets (in general a place may have several tokens with the same data value). Each of the places *NextSend* and *NextRec* starts with a single token with value 1 (belonging to type *INT*). These places represent two counters, holding the number of the next packet to be

sent/received. Place *Received* starts with a token which contains the empty string "" (belonging to type *DATA*). It represents the contents of those messages which have been successfully received. The remaining four places *A*-*D* have no tokens in the initial state. They represent input and output buffers of the transmission network.

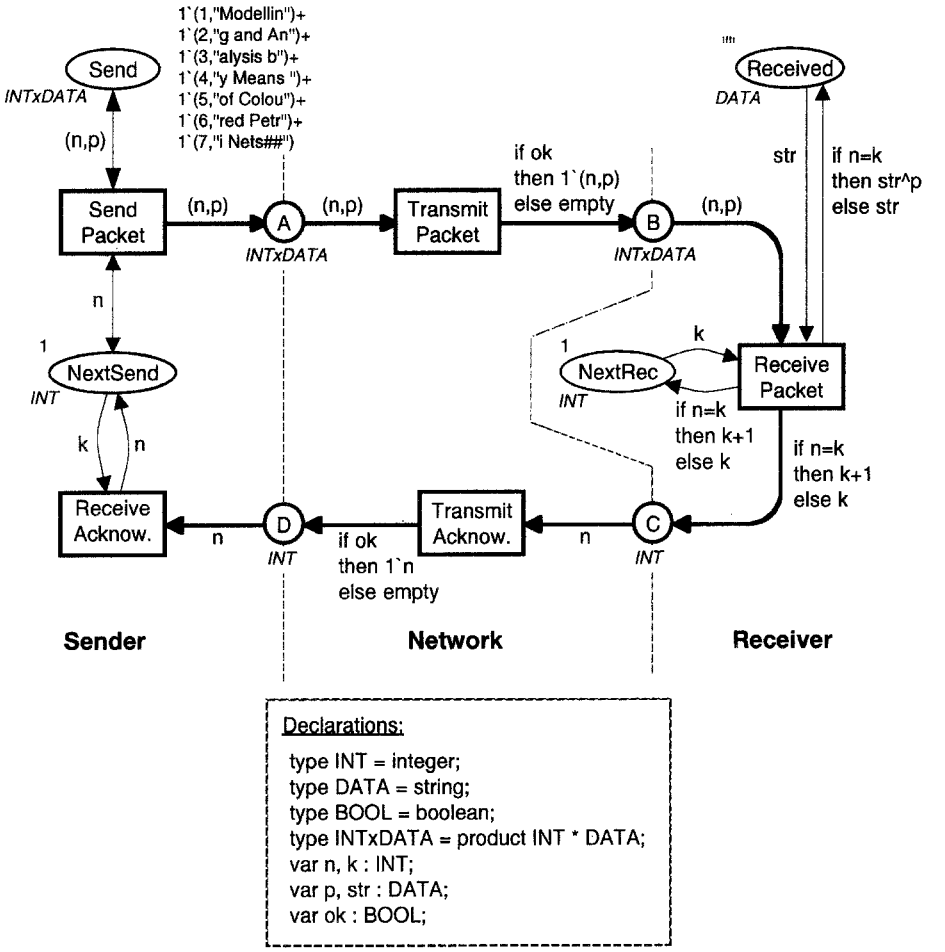

Fig. 1. Simple example of Coloured Petri Net

Coloured Petri Nets have got their name because they allow the use of tokens that carry data values and can hence be distinguished from each other — in contrast to the tokens of low-level Petri nets, which by convention are drawn as black, "uncoloured" dots. In the beginning, only small, unstructured sets of colours were used, e.g., enumerating a fixed set of processes. Later it was realised

that it was possible to generalise the theory and the tools, in such a way that arbitrarily complex data types can be used as colour sets. Today it is not at all atypical to have tokens that carry a complex data value, e.g., a list of many thousand records, involving fields of many different types. There is no longer any difference between a colour set and a type, and no difference between a token colour and a token value.

To be able to occur, a transition must have sufficient tokens on its input places, and these tokens must have token values that **match** the arc expressions. As an example, let us consider transition *SendPacket*. It has three surrounding arcs of which two are double arcs (which is a shorthand for two opposite directed arcs, sharing the same arc expression). The three arc expressions involve the variable n of type *INT* and the variable p of type *DATA*. For the transition to occur, we must **bind** these two variables to values in their types — in such a way that the arc expression of each incoming arc evaluates to a token value that is present on the corresponding input place. Since *NextSend* only contains one token with value *1*, it is obvious that n must be bound to *1*. Next we see that p must be bound to "*Modellin*", since *Send* only has one token in which the first element of the pair is *1*. With the binding <*n=1, p="Modellin"*> transition *SendPacket* is **enabled**, because there is a *1* token on place *NextSend* and a *(1,"Modellin")* token on place *Send*. When the transition **occurs** it removes the two specified tokens from the input places, but it immediately puts them back, due to the two double arcs. Simultaneously it produces a copy of the *(1,"Modellin")* token on place *A*. Intuitively, this means that we have sent the first packet by adding it to the input buffer of the network. We do not delete the packet from *Send* or update the *NextSend* counter. The reason is that the packet may be lost on the network. Hence, we may have to retransmit the packet. Our protocol is pessimistic, in the sense that it keeps transmitting the same packet, until it receives an acknowledgement requesting the next package.

When the *(1,"Modellin")* token is put on place *A*, transition *TransmitPacket* becomes enabled with two different bindings: <*n=1, p="Modellin", ok=true*> and <*n=1, p="Modellin", ok=false*>. If the first binding is chosen, the packet is transferred from place *A* to place *B*. If the second binding is chosen, the packet is lost on the network (no token is added to place *B*, since the arc expression evaluates to the *empty* multi-set). The choice between the two bindings is non-deterministic (in an interactive simulation the choice may be made by the user; in an automatic simulation it is made by a random number generator). In our example, there is a fifty-fifty chance of success/failure. It is easy to modify the CP-net to use a different probability.

Transition *SendPacket* remains enabled, and hence a retransmission may occur at any time. For the moment, we do not describe timing issues, such a duration of actions and delays before retransmissions. Instead we describe the *possibility* of having retransmissions, with a non-deterministic race between the occurrences of *SendPacket* and the other transitions. The tokens on a given place can be used in any order, independent of the order in which they arrived. This means that packets may overtake each other, both at place *A* and at place *B*.

When a token arrives at place B, transition *ReceivePacket* becomes enabled. It compares the number n in the incoming packet to the number k in the *NextRec* counter. If the two values are identical, the packet is the expected one. Then the data contents p of the packet is added to the previous contents *str* of the token in place *Received* (using the string concatenation operator ^), the *NextRec* counter is increased by one, and an acknowledgement is sent via place C (containing the number of the next packet to be received). If the two values are different, the packet is ignored. The token values on *Received* and *NextRec* are unchanged, and an acknowledgement is sent via place C (containing the number of the next packet to be received).

Transition *TransmitAcknow* works in a similar way as *TransmitPacket*. It transmits acknowledgements over the network by moving them from place C to place D, unless *ok=false*, in which case the acknowledgement is lost.

Transition *ReceiveAcknow* handles those acknowledgements that reach the *Sender*. It updates the *NextSend* counter, so that the sender begins to transmit a new packet.

2 More Complex Coloured Petri Nets

The CP-net presented above is a small toy example, used to introduce the CPN language. Industrial applications of CP-nets usually have 10-200 subnets with a total of 50-1,000 places/transitions and 10-200 types. The individual subnets are related to each other via a set of well-defined interfaces. Subnets may be reused from model to model, and it is possible to include multiple copies of a subnet in a CPN model. The syntax and semantics of CP-nets have a formal definition, which is the basis for syntax check, simulation and verification of CPN models. For more details see [1], [2] or [3].

The relationship between CP-nets and ordinary Petri Nets is analogous to the relationship between high-level programming languages and assembly code. In theory, the two levels have exactly the same computational power. However, in practice the high-level languages offer much more modelling power, because they have better structuring facilities, e.g., types and modules.

3 Analysis of Coloured Petri Nets

During the construction of a CP-net, simulations are used to validate the CPN model, i.e., to check that it has the expected behaviour. It is customary to work in an iterative way. In the early phases, the CPN model is simple, covering only selected parts of the system and ignoring many aspects of the final system. Later the scope of the model is extended and more details are added. By making simulations during the entire design process, and not just at the very end, the modellers learn about the system — in a similar way as when prototyping is used. This means that design errors can be removed at an early stage and it also means that the designers acquire new knowledge about the system — knowledge that can be used in the remaining parts of the design process.

When a CPN model has been debugged, it can be analysed in different ways. First of all, it is possible to use automatic simulations. They are similar to program executions and can be very fast with several hundreds/thousands occurring transitions per second. It is possible to specify time delays that describe the duration of the different actions in the modelled system. In this way we can make simulations that investigate the performance of the system. As an example, we may investigate how the delay between retransmissions in a protocol influences the throughput and the use of bandwidth, or we may compare the efficiency of two different protocols.

A CPN model can be verified, to prove that it behaves as desired. The most straightforward verification method builds on model checking by means of state spaces, i.e., directed graphs with a node for each reachable system state and an arc for each possible transition from one system state to another. State spaces often become huge. Hence, it is necessary to have efficient tools for construction and analysis of state spaces. We often work with partial state spaces or state spaces that are condensed in some way. Another verification method builds on place invariants, which in many respects are similar to the invariants used in program verification. This method is more demanding, with respect to the required skills of the involved personnel, and hence invariants are scarcely used in industrial projects. However, it is possible to build some of the skills into the tool support, and in this way make the place invariant method more assessable for practitioners.

As for all other formal languages, the practical use of CP-nets is highly dependent on the existence of adequate tool support. For this purpose, we have developed the Design/CPN tool package supporting the construction, editing, syntax check, and simulation of large, modular CP-nets, with or without time delays. Design/CPN also supports construction and analysis of state spaces, allowing the user to verify a large variety of different behavioural properties. The tool package is distributed free of charge to all kinds of users (including commercial companies). At the end of 1996 it was used by more than 200 institutions in 30 different countries (more than 50 of these are commercial companies). The tool package is one of the most elaborate Petri net tools available, and more than 40 man-years have been used for its design and implementation.

4 More Information on Coloured Petri Nets

A detailed introduction to CP-nets can be found in the text book [1] which consists of three volumes.

The first volume introduces the basic concepts of the CPN language, i.e., the syntax and semantics. It also defines a number of standard dynamic properties, which can be used to characterise the behaviour of a given CP-net, e.g., reachability, boundedness, home states, liveness and fairness. Finally, it describes the CPN editor and the CPN simulator, and it gives a brief introduction to the analysis methods and to four examples of industrial projects in which CP-nets have been used.

The second volume contains a detailed introduction to the theory behind the formal analysis methods. It covers state spaces, condensed state spaces, place invariants, transition invariants and timed CP-nets. Some parts of this volume are rather theoretical while other parts are application oriented.

The third volume deals with the practical use of CP-nets. It contains a detailed presentation of 19 applications of CP-nets, covering a broad range of application areas. Most of the projects have been carried out in an industrial setting. The volume presents the most important ideas and experiences from the projects, in a way which is useful, also for readers without personal experience with the construction and analysis of large CPN models. The volume demonstrates the feasibility of using CP-nets and the CPN tools for industrial projects. The presentation of the projects are based upon material provided by persons who have participated in the individual projects.

Other introductions to CP-nets can be found in [2], [3] and [4]. The first two of these are fairly general. The third focuses on the theory of condensed state spaces, but it also contains a brief introduction to CP-nets and their dynamic properties.

Finally, a lot of different material can be found on the WWW pages http://www.daimi.aau.dk/CPnets/. Here you can find the entire technical documentation of the Design/CPN tool. You can also find small examples of CPN models and a list of publications describing industrial projects in which CP-nets have been used.

References

1. K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Vol. 1: Basic Concepts, 1992, Vol. 2: Analysis Methods, 1994, Vol. 3: Practical Use, 1997.* Monographs in Theoretical Computer Science. Springer-Verlag.
2. K. Jensen. Colored Petri Nets: A High-level Language for System Design and Analysis. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
3. K. Jensen. An Introduction to the Theoretical Aspects of Coloured Petri Nets. In J.W. de Bakker and W.-P. de Roever, editors, *A Decade of Concurrency, Reflections and Perspectives*, volume 803 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
4. K. Jensen. Condensed State Spaces for Symmetrical Coloured Petri Nets. *Formal Methods in System Design 9, Kluwer Academic Publishers*, 1996.