Coloured Petri Nets: A Graphical Language for Formal Modeling and Validation of Concurrent Systems

Kurt Jensen
Computer Science Department
Aarhus University, Denmark
kjensen@cs.au.dk

Lars M. Kristensen
Department of Computing
Bergen University College, Norway
lmkr@hib.no

ABSTRACT

Coloured Petri Nets (CPNs) combine Petri nets with a programming language to obtain a scalable formal modeling language for concurrent systems. Petri nets provide the formal foundation for modeling concurrency and synchronization, and a programming language provides the primitives for modeling data manipulation and creating compact and parameterizable models. We provide an example driven introduction to the core syntactical and semantical constructs of the CPN modeling language, and briefly surveys how quantitative and qualitative behavioral properties of CPN models can be validated using simulation-based performance analysis and explicit state space exploration. In addition, we give a brief overview of CPN Tools which provide tool support for the practical use of CPNs, and provide pointers to some significant examples where the CPN technology has been put into practical use in an industrial setting. As we proceed, we provide a historical perspective on the research that led to the development of the CPN language.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

TODO

1. INTRODUCTION

The vast majority of IT systems today can be characterized as concurrent and distributed systems in that their operation inherently relies on communication, synchronization, and resource sharing between concurrently executing software components and applications. This is a development that has been accelerated first with the pervasive presence of the Internet as a communication infrastructure, and in recent years by, e.g, cloud- and web-based services, mobile applications, and multi-core computing architectures.

The development of Coloured Petri Nets (CPNs) was initiated in the early 80'es when distributed system was considered to become a major paradigm for future computing systems. The goal of the CPN modeling language was to develop a formally founded modeling language for concurrent system that would make it possible to formally analyze and validate concurrent systems, and which from a modeling perspective would scale to industrial systems. A main motivation behind the research into CPNs (and many other formal modeling languages) was that the engineering of correct concurrent systems is a challenging task due of their complex behavior which may result in subtle bugs. As concurrent systems are becoming still more pervasive and critical to society, formal techniques for concurrent system was – and still is – a highly relevant technology to support the engineering of reliable concurrent systems.

At its very base, CPNs builds on the visionary work of C. A Petri [**?**] who already in the 60'es introduced Petri Nets as a formalism for concurrency an synchronization. In Petri Nets, concurrency is a fundamental concept in that Petri Nets is inherently based on the idea that everything is (implicitly) concurrent unless explicitly synchronized. This is in contrast to many other modeling formalisms where concurrency must be explicitly introduced using parallel composition operators. A further advantage of Petri nets is that they rely on very few basic concepts, and is still able to model a wide range of communication and synchronization concepts and patterns. A disadvantage of Petri nets in their basic form is, however, that they do not scale to large systems unless one models the systems at a very high level of abstraction. The primary reasons for this is that Petri nets are not suited for modeling sequential computation and data manipulation and they do not provide concepts that make it easy to scale models according to some system parameter, e.g,. increase the number of servers in a modeled system without having to make major changes to the model.

The shortcoming of ordinary Petri nets outlined above prompted a research direction into the development of high-level Petri nets which ... KURT TO ADD HISTORICAL PERSPECTIVE ON THE DEVELOPMENT OF HIGH-LEVEL NETS. WE NEED TO TALK ABOUT STANDARD ML SOMEWHERE AND SAY THAT CPN ML IS BASED ON SML.

2. COLOURED PETRI NETS

To present the concepts of CPNs, we use a CPN model of a distributed two-phase commit transaction system. We use the example to give an informal introduction to CPNs. The reader interested in the formal definition of CPNs is referred to [**?**].

The CPN model of the two-phase commit system is comprised of 4 *modules* hierarchically organized into three level. Figure 1 shows the top-level module which consists of two *substitution transitions* (drawn as rectangles with double-lined borders) representing the Coordinator and the *Workers* in the system. Each of the substitution transition has an associated *submodule* that model the behavior of the coordinator and the workers, respectively. The name of an associated submodule is written in the rectangular tag positioned at the bottom of each substitution transition.

The two substitution transitions are connected via directed *arcs* to the four places CanCommit, Votes, Decision, and Acknowledge (drawn as ellipses). Places connected to substitution transitions are called *socket places* and are linked to *port* places on the associated submodules (to be presented shortly). The coordinator and the workers interact by producing and consuming *tokens* on the places. These tokens carry data values and the type of tokens that may reside on a place is determined by the *color set* (type) of the place (written in text below the place). Figure 2 lists the definitions of the color sets used as color set of the four places in Fig. 1. These color sets are defined using the CPN ML programming language.

The `Worker` color set is an indexed color set consisting of the values `wrk(1),wrk(2),...,wrk(W)` where the symbolic constant `W` is used to specify the number of worker processes considered. This color set is used to model the identity of the worker processes. The color set `Vote` is an enumeration color set containing the values `Yes` and `No`, and is used to model that a worker by vote yes or no to commit the transaction. The color set `WorkerxVote` is a product type containing pairs consisting of a worker and a vote. The color set `Decision` is an enumeration color set used to model whether the coordinator decides to `abort` or `commit` the transaction (only if all workers vote yes will the transaction be committed). It should be noted that in addition to the type constructors introduced above, CPN ML supports union, lists, and record types. The variables `w` and `vote` declared in Fig. 2 will be introduced later.

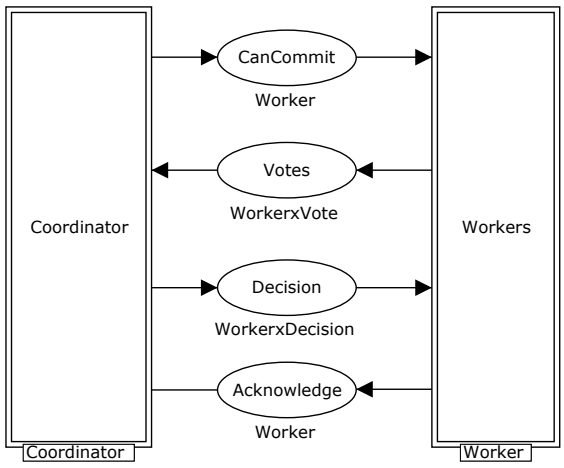

Figure 1: The top-level module of the CPN model.

```
val W = 2;
colset Worker = index wrk with  1..W;
var w : Worker;

colset Vote        = with Yes | No;
var vote : Vote;
colset WorkerxVote = product Worker * Vote;

colset Decision        = with abort | commit;
colset WorkerxDecision = product Worker * Decision;
```

Figure 2: Color sets definition used in Fig. 1.

The state of a CPN model is called a *marking*), and consists of a distribution of *tokens* on the places of the model. Each place may hold a (possibly empty) *multi-set* of tokens with data values (colors) from the color set of the place. The *initial marking* of a place is specified above each place (and omitted if the initial marking is the empty multi-set). For the places in Fig. 1 all places are empty in the initial marking. Initially, the *current marking* of a CPN model equals the initial marking. When a CPN model is executed occurrences of enabled transitions consume and produce tokens on the places which will change the *current marking* of the CPN model.

The Coordinator module is shown in Fig. 3. This is the submodule associated with the Coordinator submodule in Fig. 1. The places CanCommit, Votes, Decision, and Acknowledge are *port places* as indicated by the rectangular In and Out tags positioned next to them. These places are linked to the accordingly named places in the top-level module (Fig. 1) via a *port-socket association* which implies that any tokens added/removed from a port place by transitions in the *Coordinator* module will also be reflected in the marking of the associated socket place in the top-level module. The places CanCommit and Decision are *output port places* which means that the Coordinator module will only produce tokens on these places. The places Votes and Acknowledge are *input port places* which means that the module will only consume tokens from these places. It is also possible for a place to be an input-output port place which means that the module may both produce and consume tokens on (from) this place.

The places Idle, WaitingVotes, and WaitingAcknowledgement are used to model the states that the coordinator goes through when executing the two-phase commit protocol. The places Idle and WaitingVotes have the color set `UNIT` containing just a single value `()` (denoted unit). Initially, the coordinator is in an idle state as modeled by the initial marking of place Coordinator which consist of a single token with the color unit. In CPN ML this multi-set is written `1‘()` specifying one (`1`) occurrence of (`‘`) the unit color (`()`). The number of tokens on a place in the current marking is indicated with a small circle positioned next to a place, and the detail of the

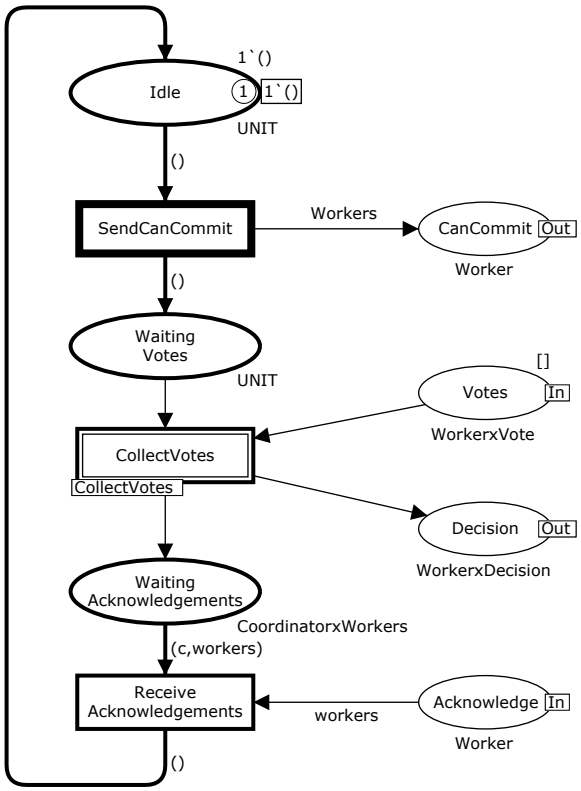

Figure 3: The Coordinator module.

color of the tokens are provided in an associated text box. The indication of the current marking of a place is omitted if currently the place contains no tokens.

The transitions SendCanCommit, CollectVotes, and ReceiveAcknowledge model the event/actions that cause the coordinator to change state. The coordinator will first send a can commit message (transition SendCanCommit) to each worker asking whether they can commit the transaction; then the coordinator will collect the votes from all workers (transition CollectVotes) and then send a decision to each worker indicating whether the transaction is to be committed or not. Finally, the coordinator will receive an acknowledgment from each worker that they have received the decision (transition ReceiveAcknowledgements). It should be noted that CollectVotes is a substitution transition which means that the details of how the coordinator collects votes is modeled by the associated CollectVotes submodule. This illustrates that it is possible to mix the use of ordinary and substitution transitions within a module.

In the current marking shown in Fig. 3 only the transition SendCanCommit is enabled as indicated by the thick border of that transition. The requirement for a transition to be *enabled* is determined from the *arc expressions* associated with the incoming arcs of the transition. In this case, there is only a single incoming arc from place Idle containing the expression `()`. This expression specifies that for SendCanCommit to be enabled, there must be at least one `()`-token present on `Idle`. When the `SendCanCommit` transition *occurs*, it will consume a `()`-token from place `Idle` and it will produce tokens on place connected to output arcs as determining by evaluating the arc expressions on output arcs. In this case, the expression `()` on the arc to `WaitingVotes` evaluates to a single `()`-token. The expression `Worker.all ()` is a function call when calls a function `Worker.all` that takes a unit value as argument and returns all colors of the color set `Worker`. This illustrates that arc expression may also make use of function to calculate the tokens to be added (removed) from places. In particular this means that complex data manipulation can be performed without intruding intermediate steps in the model itself.

Figure 4 shows the marking of the surrounding places of transition SendCanCommit after the occurrence of SendCanCommit. It can be seen that the place `CanCommit` contains two tokens - one of each worker in the system - representing messages going to the two worker processes. The coordinator has now entered a state in which it is waiting to collect the votes from the worker processes.

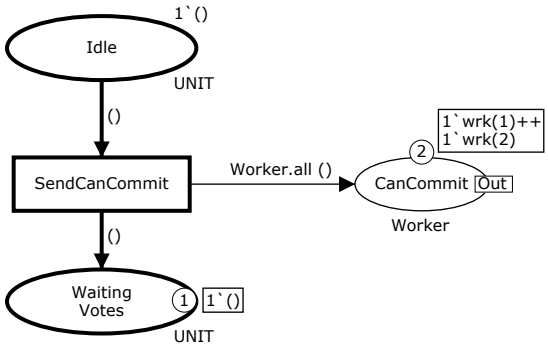

Figure 4: Current marking after SendCanCommit.

By setting the symbolic constant `W` (see Fig. 2) we can easily configure to model to handle, e.g., five workers. With ordinary Petri nets we would have had to create a copy of the CanCommit place for each worker. In particular, we would have to make changes to the net structure (places, transitions, arcs) when changing the number of workers. This shows that CPNs provides a means for easily creating parameterizable models and also that it enables more compact modeling as we only need a single instance of the CanCommit place in order to accommodate any finite number of workers.

- WE COULD EABORATE MORE ON THIS BY DRAWING THE CORRESPONDING PT-NET FOR THE CANCOMMIT TRANSITION. THAT WOULD ILLUSTRATE UNFOLDING OF PLACES (see comment at the end of this section that proposes illustration of unfolding transition. Perhaps it is a good approach to split the introduction of unfolding in two steps. Then at the end of the section we can mention that any CPNs can be unfolder to a possibly infinite behaviorally equivalent PT-nets. The two example that we have in this section would then nicely illustrate how. It would also be possible to factor this into a seperate section in order to avoid that this section becomes overlong in comparison with the other section.

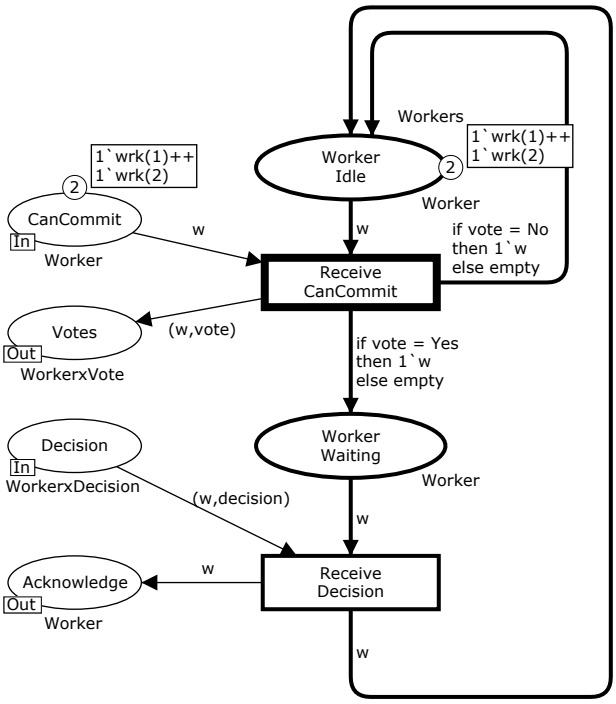

Figure 5: The Worker module.

Figure 5 shows the Worker module which is the submodule of the Workers substitution transition in Fig. 1. The places CanCommit, Votes, Decision, and Acknowledge constitute the port places of this module and are linked to the accordingly named socket places in Fig. 1. The places Idle and Waiting models the two main states of worker processes. Each of these places have the color set `Worker` and the idea is that when there is a token with color `wrk(i)` on, e.g., the place Idle, then this represent that the i'th worker is in state idle. This makes it possible to model the state of all workers in a compact manner within a single module without having to have a place for each worker or a module instance for each worker. Initially, all workers are in the idle state as represented by corresponding tokens on place Idle in the initial marking. The transition ReceiveCanCommit models the reception of can commit messages from the coordinator and the sending of a vote. The transition ReceiveDecision models the reception of a decision message from the coordinator and the sending of an acknowledgment.

The current marking of place CanCommit is `1‘wrk(1) ++ 1‘wrk(2)` corresponding a marking where the coordinator has sent a can commit message to each worker. The thick border of transition ReceiveCanCommit indicates that this transition is enabled in the current marking. The arc expressions on the surrounding arcs of the ReceiveCanCommit transition are more complex than the arc expressions of the SendCanCommit transition in the Coordinator module considered earlier in that they contain the `free variables` `w` and `vote` defined in Fig. 2. This means that in order to talk about the enabling and occurrence of transition `ReceiveCanCommit`, we need to assign values to these variables in order to evaluate the input and output arc expressions. This is done by creating a *binding* which associates a value to each of the free variables occurring in the arc expression of the transition. Bindings can be considered different modes in which a transition may occur. As `w` is of type `Worker` and `vote` is of type `Decision`, this gives the following four possible bindings reflecting that each of the two workers may vote yes or no to committing the transactions:

$$
\begin{aligned}
b_{1Y} &= \langle \texttt{w} = \texttt{wrk(1)}, \texttt{vote} = \texttt{Yes} \rangle \\
b_{1N} &= \langle \texttt{w} = \texttt{wrk(1)}, \texttt{vote} = \texttt{No} \rangle \\
b_{2Y} &= \langle \texttt{w} = \texttt{wrk(2)}, \texttt{vote} = \texttt{Yes} \rangle \\
b_{2N} &= \langle \texttt{w} = \texttt{wrk(2)}, \texttt{vote} = \texttt{No} \rangle
\end{aligned}
$$

A binding of a transition is enabled if evaluating each input arc expression in the binding results in a multi-set of tokens which is a subset of the multi-set of tokens present on the corresponding input place. For an example, consider the binding b_{1Y}. Evaluating the input arc expression `w` on the input arc from `Idle` results in the multi-set containing a single token with the color `wrk(1)` which is contained in the multi-set of tokens present on place Idle in the marking depicted in Fig. 5. Similarly for the input arc expression on the arc from place CanCommit. This means that binding b_{1Y} is enabled and may occur. In fact, all the four bindings listed above is enabled in the marking shown in Fig. 1.

The tokens produced on output places when an transition occur in an enabled binding is determined by evaluating the output arc expressions of the transition in the given binding. Consider again the binding element b_{1Y}. The output arc expression `(w,vote)` will evaluate to `wrk(1),Yes` and this token will be added to place `Votes` to inform the coordinator that worker 1 votes yes to committing the transaction. The arc expression on the arc from ReceiveCanCommit to Waiting is an if-then-else expression which in the binding b_{1Y} will evaluate to the multi-set $1‘wrk(1)$ which will then be added to the tokens on place Waiting. The if-then-else expression on the arc from ReceiveCanCommit evaluates to the `empty` multi-set and hence no tokens will be added to place Idle in this case. Figure 6 shows the marking resulting from an occurrence of the b_{1Y} binding.

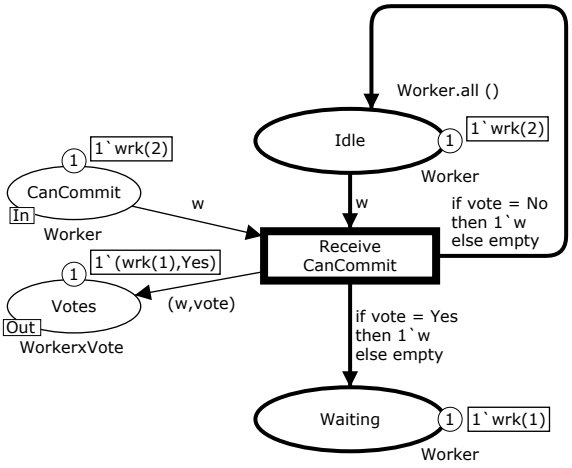

Figure 6: Current marking after ReceiveCanCommit.

The occurrence of the binding b_{1N} representing that worker one votes no would have the effect of removing a `wrk(1)`-token from Idle, and adding no tokens to place Waiting and adding one `wrk(1)`-token to place Idle. This models the fact that if a worker votes no to committing the transaction, then it goes back to idle; whereas if it votes yes, then it will go to waiting to be informed about whether the transaction is to be committed or not. Recall that this is a distributed system and hence a worker cannot (without exchanging messages) know what other workers have voted. Above we have considered relatively simple arc expression but the arc expression of a transition can be any expression that can be written in Standard ML as long as it has a type that matches the corresponding place. In particular, arc expressions may apply functions including higher-order functions.

All the four bindings listed for transition ReceiveCanCommit were enabled in the marking shown in Fig. 5. Furthermore, enabled bindings may be *concurrently enabled* if each binding can get its required multi-set of tokens from each input place independently of the other enabled bindings in the set. As an example, the two bindings b_{1Y} and b_{2Y} are concurrent enabled since each binding can gets its token from the input places without competing with each other. This reflects that the workers are executing concurrently and may simultaneously send a vote back to the coordinator. In contrast, the two bindings b_{1Y} and b_{1N} are not concurrent. These two bindings are in *conflict* because they each need, e.g., the single `wrk(1)`-token on Idle (and in fact also the single `wrk(1)`-token on place CanCommit. The notion of concurrency and conflict of binding extends to also span to bindings of different transitions. A fundamental property of a set of enabled bindings that CPNs inherit from Petri nets is that theses can be executed in any interleaved order and the resulting marking will be the same independently of the interleaved execution considered.

- HERE WE COULD MAKE A FURTHER LINK TO PT-NETS BY SHOWING THE FRAGMENT CORRESPONDING TO RECEIVECANCOMMIT. THAT WOULD ILLUSTRATE UNFOLDING OF TRANSITIONS.
- HERE WE COULD BRIEFLY MENTION GUARDS.
- WE COULD BRIEFLY TALK ABOUT MODELING TIME IF WE SKIP HAVING A SECTION ON THIS.

3. TIMED CPNS

4. ANALYSIS AND VALIDATION

5. CPN TOOLS AND APPLICATIONS

The construction and analysis of CPN models have been supported by two generations of graphical computer tools. The first generation was the Design/CPN tool [**?**] which was developed starting in the mid 80'es at Meta Software Corp. and later by the CPN Group at the University of Aarhus. This was followed by CPN Tools [**?**] that has been developed since 2000 first by the CPN Group at the University of Aarhus and since 2009? by the XX group at the Technical University of Eindhoven. CPN Tools supports the editing and construction of CPN models, interactive and automatic simulation, state space-based model checking, and simulation-based performance analysis. Both Design/CPN and CPN Tools has been widely distributed tools and they have been applied for modelling and validation in a broad range of application domains. Below we provide some pointer to some selected applications within typical application domains. A more comprehensive list of example applications and domains can be found via [**?**].

Embedded Systems. Dalcotech or B and O

Process Scheduling. COAST

Internet Protocols. ERDP

Mobile Phone Software. NOKIA

Capacity Planning. HP

6. REFERENCES