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

Kurt Jensen
Department of Computer Science
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 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 give a historical perspective on the research contributions that led to the development of the CPN technology.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Petri nets*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Design, Verification, Reliability, Theory

Keywords

Coloured Petri Nets, Formal Modelling and Validation

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 becoming a major paradigm for future computing systems. The goal of the CPN modeling language was to develop a formally founded modeling language for concurrent systems 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 to their complex behavior which may result in subtle bugs if not carefully designed. 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 origin, CPNs builds on Petri nets (see sidebar on Petri nets) that were introduced by Carl Adam Petri in his doctoral thesis published in 1962 [10] as a formalism for concurrency and synchronization. This was far ahead of the time where distributed systems were invented and computers started to have parallel processes. At that time, programs and processing were considered to be sequential and deterministic. Hence, it was extremely visionary of Carl Adam Petri to predict the importance of being able to understand and characterize the basic concepts of concurrency. In Petri nets, concurrency is a fundamental concept in that Petri nets is inherently based on the idea that behaviour 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.

Sidebar: Petri Nets. A Petri net is a directed bi-partite graph with nodes consisting of places (drawn as ellipses) and transitions (drawn as rectangles). The state of a Petri called a marking consists of a distribution of tokens (drawn as black

[*]The Coloured Petri Nets model presented in this paper is available via `http://X.Y.Z/cpncommitmodel` and CPN Tools is available for download via `www.cpntools.org`

dots) positioned on the places and the execution of a Petri net consists of occurrences of enabled transitions removed tokens from input places and adding tokens to output places as described by integer arc weights thereby changing the current state (marking) of the Petri net. □

In the decade following the introduction of Petri nets by C. A. Petri, Petri nets were widely accepted as one of the most well-founded theories to describe important behavioral concepts such as concurrency, conflict, synchronization and resource sharing. Petri nets were also used to model and analyze smaller concurrent systems. However, the practical use soon revealed a serious shortcoming. Petri nets (in their basic form) 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 well-suited for modeling systems in which data and manipulation of data plays a crucial role. Furthermore, Petri nets did not provide concepts that made 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. This implied that the use of Petri nets for practical modelling were staggering. To remedy this situation many researchers proposed different ad-hoc extensions to Petri nets. This created a large zoo of different Petri net modeling languages. Many ad-hoc extensions were not well-defined, and even when they were, they often had fundamental theoretical shortcomings. Whenever a new ad-hoc extension was introduced, all the basic concepts and analysis methods had to be redefined - to apply for the extended Petri net language (with the ad-hoc extension). With the invention of the first (text-based) computer tools to support the analysis of Petri net models, the situation became acute. Whenever a new ad-hoc extension was introduced (to handle a modeling shortcoming) all existing computer tools became void, and could only be used after time-consuming and error-prone reprogramming. Hence, there was an urgent need to develop a class of Petri nets that were general enough to handle a large variety of different application areas without the need of making ad-hoc extensions.

The first successful step towards a common more powerful class of Petri nets were taken by Genrich and Lautenbach in 1979 with the introduction of Predicate/Transition Nets (PrT nets) [3]. Their work was inspired by earlier work on transition nets with *colored tokens* by Schiffers and Wedde [?] and transition nets with *complex conditions* [?] by Shapiro. The basic idea behind PrT nets was to introduce a set of colored tokens which can be distinguished from each other - in contrast to the indistinguishable black tokens in basic Petri nets. In this way it became possible to model different processes in a single subnet. PrT nets used arc expressions to define how transitions can occur in different ways (occurrence modes) depending of the colors of the involved input and output tokens. The invention of colored distinguishable tokens in PrT nets was a gigantic step forward – but it still had some limitations. PrT nets only had one set of token colors, and all places have to use this set (or Cartesian products based on this set).

The second step towards a more general class of Petri nets was taken by Jensen in his PhD thesis in 1980 [5] with the introduction of the first kind of Colored Petri Nets (CPNs). This Petri net model allowed the modeler to use a number of different color sets This made it possible to represent data values in a more intuitive way instead of having to encode all data into a single shared set. It later turned out to be convenient to define the color sets by means of data types known from programming languages, such as products, records, lists, and enumerations. The use of types had three implications: Token colors became structured (and hence much more powerful); type checking became possible (making it much easier to locate modeling errors); and color sets, arc expressions and guards could be specified by the well-known and powerful syntax and semantics known from programming languages. This gave the modeler a convenient way to handle complex data and specify the often complex interaction between data and system behavior. A third step forward was taken by Huber, Jensen, and Shapiro in 1990 with the introduction of Hierarchical CPNs [4]. Their work was heavily inspired by the hierarchy concepts in the Structured Analysis and Design Technique (SADT) developed by Marca and McGowan [?]. It was Shapiro who got the idea to port the SADT hierarchy concepts to CPN. The introduction of hierarchical CPNs allowed the modeler to structure a large CPN model into a number of interacting and re-usable modules - in a similar way as known from programming languages. This implied that Petri net models of large systems become much more tractable, since they can be split into modules of a reasonable size and the model can be viewed at different levels of abstraction.

2. COLOURED PETRI NETS

To give an informal introduction to the concepts of CPNs, we use a CPN model of a distributed two-phase commit transaction system. The reader interested in the formal definition of CPNs is referred to [7]. The CPN model of the two-phase commit system is comprised of four *modules* hierarchically organized into three levels. 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 transitions has an associated *submodule* that model the detailed behavior of the coordinator and the workers, respectively. The name of the submodule is written in the rectangular tag positioned at the bottom of each substitution transition.

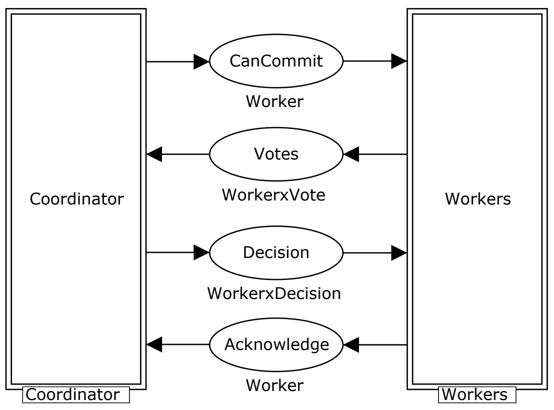

Figure 1: The top-level module of the CPN model.

```
val W = 2;

colset Worker = index wrk with  1..W;
colset Vote = with Yes | No;
colset WorkerxVote = product Worker * Vote;

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

var w : Worker;
var vote : Vote;
```

Figure 2: Definition of color sets used in Fig. 1.

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 *type* of the place (written in text below the place). For historical reasons the types of places are called *color sets*. Figure 2 lists the definitions of the color sets used for the four places in Fig. 1. These color sets are defined using the CPN ML programming language which is based on the functional language Standard ML [?].

The `Worker` color set is an indexed type 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 type containing the values `Yes` and `No`, and is used to model that a worker may vote `Yes` or `No` to commit the transaction. The color set `WorkerxVote` is a product type containing pairs consisting of a worker and its vote. The color set `Decision` is an enumeration type 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.

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 substitution transition in Fig. 1. The places CanCommit, Votes, Decision, and

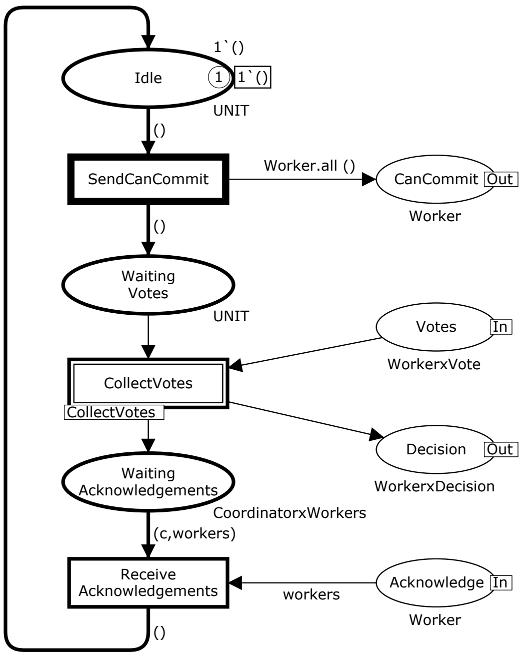

Figure 3: The Coordinator module.

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 added (removed) in the marking of the associated socket place in the top-level module. The places Votes and Acknowledge are *input port places* which means that the Coordinator module will only consume tokens from these places. The places CanCommit and Decision are *output port places* which means that the Coordinator module will only produce tokens on these places. It is also possible for a place to be an *input-output port place* which means that the module may both consume and produce tokens from (on) this place.

The places Idle, WaitingVotes, and WaitingAcknowledgement are used to model the states of the coordinator 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 Idle which consists 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 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 ReceiveAcknowledgements model the events/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 (substitution transition CollectVotes) and send a decision to the workers that voted yes indicating whether the transaction is to be committed or not. Finally, the coordinator will receive an acknowledgment from each worker that voted yes, confirming 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 how use of ordinary and substitution transitions can be mixed within a module. We omit the details of CollectVotes in this paper.

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 places connected to output arcs as determined 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 call to the function `Worker.all` that takes a unit value (`()`) as parameter and returns all colors of the color set `Worker`. This illustrates how complex calculations (e.g., of multi-sets of tokens or involving tokens with complex data values) can be encapsulated within function calls, and how several tokens can be added/removed in a single step (transition occurrence) without introducing intermediate states (markings).

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 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 WaitingDecision model 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 models 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.

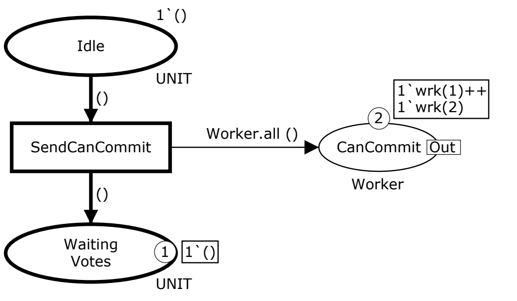

Figure 4: Current marking after SendCanCommit.

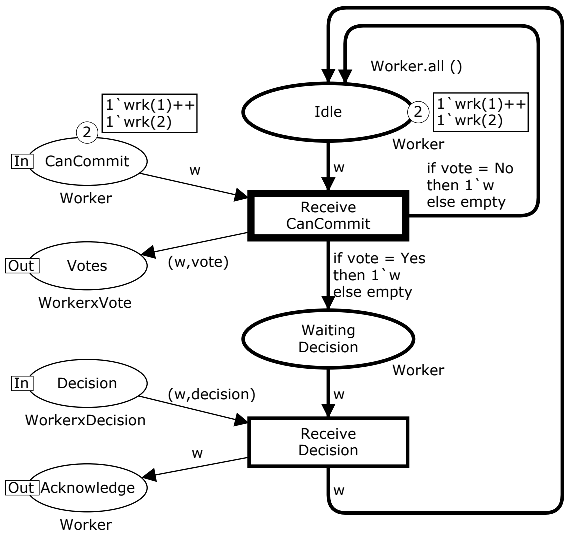

Figure 5: The Worker module.

The current marking of place CanCommit in Fig. 5 is `1‘wrk(1) ++ 1‘wrk(2)` modelling 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 bind (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 four bindings listed above is enabled in the marking shown in Fig. 5.

The tokens produced on output places when a transition occurs 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)` evaluates 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 WaitingDecision is an if-then-else expression which in the binding b_{1Y} evaluates to the multi-set `1‘wrk(1)` which will be added to the tokens on place WaitingDecision. The if-then-else expression on the arc from ReceiveCanCommit to Idle 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.

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, adding a `(wrk(1),No)`-token to Votes, adding no tokens to place WaitingDecision and adding a `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.

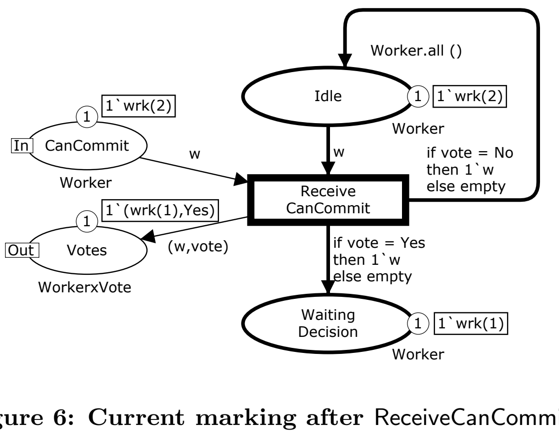

Figure 6: Current marking after ReceiveCanCommit.

All four bindings listed for transition ReceiveCanCommit were enabled in the marking shown in Fig. 5. 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 concurrently enabled since each binding can gets its token from the input places without sharing 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 concurrently enabled. These two bindings are in *conflict* because they each need 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 bindings of different transitions. A fundamental property of a set of concurrently enabled bindings that CPNs inherit from Petri nets is that these can be executed in any interleaved order and the resulting marking will be the same independently of the interleaved execution considered.

Above relatively simple arc expressions were used. In general, arc expressions can be any expression that can be written in Standard ML as long as they have types that matches the corresponding places. In particular, arc expressions may apply functions including higher-order functions. CPNs also includes a notion of time inspired by the work of van der Aalst [?] that makes it possible to model the time taken by different activities. It is based on the introduction of a *global model clock* that represents the current model time and on attaching *time stamps* to tokens in addition to the token colors. The time stamp of a token specifies the earliest model time at which the token can be removed by the occurrence of a transition, and delay inscriptions on the transitions and arcs are used to determined the timestamps on tokens produced by transitions. During model execution, the global model clock is always advanced to the earliest next time at which a transition becomes enabled, and the model stays at the current model time until no more transitions are enabled. The time concept provides the foundation for conducting simulation-based performance analysis of CPN models.

3. UNFOLDING AND FOLDING OF CPNS

The extensions of Petri nets that CPNs bring in the form of data types, a programming language, and modules add practical modeling power to Petri nets by making it possible to create models of complex real systems. The extensions do not add expressive power from a theoretical perspective as any hierarchical CPN model can be unfolded to a non-hierarchical CPN model which in turn can be *unfolded* to a behaviorally equivalent (and possibly infinite) Place/Transition net (PTN). The unfolding of a hierarchical CPN to a non-hierarchical CPN consists of recursively replacing each substitution transition with its associated submodule such that related port and socket places are merged into a single place. The unfolding of a non-hierarchical CPN to a PTN consists of unfolding each CPN place to a PTN place for each color in the color set of the CPN place, and unfolding each CPN transition to a PTN transition for each possible binding of the CPN transition. In the other direction, any PTN can be *folded* into a CPN with a single module consisting of a single place and a single transition. In practice such a folding is not interesting, because the arc expressions will be extremely complex and non-interpretable for a human being. However, the fact that the unfolding and folding exists shows that hierarchical CPNs has the same theoretical properties as basic Petri nets – in particular that CPNs constitute a solid model for concurrency, conflict, synchronization and resource sharing.

To illustrate the unfolding consider the fragment shown in Fig. 4. To represent this fragment as an ordinary Petri net, the CPN place CanCommit needs to be unfolded to PTN places corresponding to `wrk(1)` and `wrk(2)`. The place Idle and WaitingVotes does not need to be unfolded as the `Unit` color set contains just a single value. Similarly, no unfolding of the SendCanCommit transition is required in this case since the transition does not have any variables. The equivalent PTN is shown in Fig, 7. It can be seen that the arc expressions are replaced by arc weights, and that the initial marking is replaced by the specification of a single token initially in Idle. To show the unfolding of a CPN transition consider the CPN model fragment shown in Fig. 6. To represent this as a PTN, we need to unfold the places as explained in the previous paragraph and in addition unfold the transition ReceiveCanCommit to a PTN transition for each of the four binding elements listed in Fig. **??**. Figure 8 shows the corresponding PTN for worker one. The PTN will have an identical copy also for worker two.

The unfolding above also demonstrates that Petri nets do not provide a way to easily scale the model according to some system parameter (in this case the number of workers). With ordinary Petri nets is is necessary to have a subnet for each worker (even though they behave in exactly the same way). With PrT nets and CPNs we can use tokens with color `wrk(1)` to model the state of the first worker, tokens with the color `wrk(2)` to model the state of the second worker, and so on. This means that we for `W` workers can have a single Idle place, which may contain tokens of `W` different colors – instead of having a separate `Idle` place for each of the `W` workers. In particular, in the CPN model where we just need to change the symbolic constant `W` (see Fig. 2) to configure the model to handle, e.g., five workers whereas with basic Petri nets we need to add places, transitions, and arcs and hence change the net structure in order to increase 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

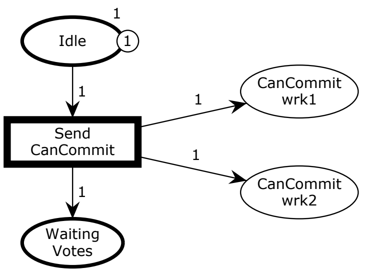

Figure 7: PTN representation of the CPN in Fig. 4.

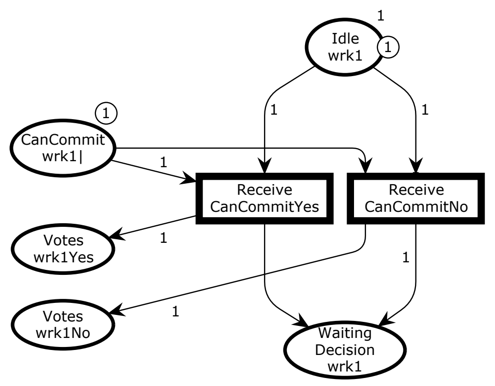

Figure 8: PTN representation corresponding to worker one for the CPN in Fig. 6.

single instance of the CanCommit place in order to accommodate any finite number of workers. Comparing CPN and PrT nets. With CPN it is possible to have many color sets and hence we can use a number of different color sets (e.g. one color set for the coordinator, a second for the workers, a third for Yes/no votes and a fourth for abort/commit decisions). With PrT nets only one set of token colors are allowed (or Cartesian product thereof) and hence with PrT nets we could have had to represent the identity workers, Yes/No votes and abort/commit decisions by colors based on this single set of token colors.

4. TOOLS AND APPLICATIONS

The construction and analysis of CPN models have been supported by two generations of graphical computer tools: Design/CPN and CPN Tools. These tools have been instrumental to the success of CPNs as they have enabled the practical use in a broad range of domains such as distributed software systems, communication protocols, embedded systems, and process- and workflow modeling. A comprehensive list containing more than hundred papers describing practical applications and domains is available [9].

The first generation was the Design/CPN tool [**?**] created at Meta Software, Cambridge, Massachusetts, USA starting in 1988. The main architects behind the tool were Jensen, Shapiro and Huber [8] and the implementation was made together with an international group of people. The first version of Design/CPN supported modeling, syntax cheek and interactive simulation. The introduction of hierarchical CPNs supported by the Design/CPN tool made a dramatic change to the practical used of Petri nets. The new modeling language and its tool support were general and powerful enough to eliminate the need of making ad-hoc extensions as discussed in the introduction of this paper. A common platform for practical modeling had been established and this was used by most Petri net practitioners. The use of the platform was supported by a three volume monograph on Colored Petri Nets published by Kurt Jensen in 1992-1997 [6]. The CPN research group at Aarhus University was responsible for the development of the Design/CPN tool in the period 1996-2002. Starting from year 2000 a second generation of tool support, called CPN Tools, was designed and implemented at Aarhus University, Denmark. The main architects behind the new tool were Jensen, Christensen, and Westergaard [?]. It was based on empirical studies of the use of Design/CPN and much easier and efficient to use when constructing CPN models. In 2010 CPN Tools had 10.000 licenses in 150 countries. At that time the development and maintenance of the tool set were transferred to the group of Wil van der Aalst at the Technical University of Eindhoven, The Netherlands [2]. New updates with improved functionality are made at a regular basis.

CPN Tools supports the editing and construction of CPN models, interactive and automatic simulation, state space-based model checking (see sidebar), and simulation-based performance analysis (see sidebar). CPN Tools is based on a much faster simulation engine compared to the one in Design/CPN developed by Haagh and Hansen [?]. With this simulation engine, models run more than a factor thousand times faster compared to Design/CPN allowing complex automatic simulations to be executed within seconds instead of hours. The suite of state space methods supported by CPN Tools has been developed by Kristensen, Westergaard, Evangelist, and Mailund [?]. The support for simulation-based performance analysis developed is based on the work of Wells and Lindstrøm [?].

Figure 9 provides a screen-shot of CPN Tools with the CPN model considered in this paper. The user of CPN Tools works directly with the graphical representation of the CPN model. The graphical user interface of CPN Tools has no conventional menu bars and pull-down menus, but is based on interaction techniques, such as *tool palettes* and *marking menus*. The rectangular area to the left is an *index*. It includes the Tool box, which is available for the user to manipulate the declarations and modules that constitute the CPN model. The Tool box includes tools for creating, copying, and cloning the basic elements of CPNs. It also contains a wide selection of tools to manipulate the graphical layout and the appearance of the objects in the CPN model. The latter set of tools is very important in order to be able to create readable and graphically appealing CPN models. The remaining part of the screen is the *workspace*, which in this case contains two *binders* (the rectangular windows) and a circular pop-up menu.Each binder may hold a number of items which can be accessed by clicking the tabs at the top of the binder (only one item is visible at a time). In the example shown, there are two binders. One binder (left) containing the Commit module and one binder (right) contained the Coordinator module. In addition, two tool palettes are shown: one (Sim) containing the tools that can be used for simulation of the model, and one tool palette (Create) containing the tools for creating CPN model elements. Items can be dragged from the index to the binders, and from one binder to another binder. A circular marking menu has been popped up on top of the right binder. Marking menus are contextual menus that make it possible to select among the operations possible on a given object. In the case of Fig. 9, the marking menu gives the operations that can be performed on transition.

CPN Tools performs syntax and type checking, and error messages are provided to the user in a contextual manner next to the object causing the error. The syntax check and code generation are incremental and are performed in parallel with editing. This means that it is possible to execute parts of a CPN model even if the model is not complete, and that when parts of a CPN model are modified, a syntax check and code generation are performed only on the elements that depend on the parts that were modified. CPN Tools supports two types of simulation: interactive and automatic. In an interactive simulation, the user is in complete control and determines the individual steps in the simulation, by selecting between the enabled events in the current state. CPN Tools shows the effect of executing a selected step in the graphical representation of the CPN model. In an automatic simulation the user specifies the number of steps that are to be executed and/or sets a number of stop criteria and breakpoints. The simulator then automatically executes the model without user interaction by making random choices between the enabled events in the states encountered. Only the resulting state is shown in the GUI. CPN Tools also includes support for getting domain-specific graphical feedback from ongoing simulations developed by Westergaard [?].

Sidebar: State Spaces and Model Checking. State space exploration is the main technique used for verifying behavioral properties of CPN models. In its basic form, a state space is a directed graph consisting of a node for each reachable marking (state) of the CPN model and edges corresponding to occurrences of enabled binding elements. From a constructed state space it is possible to automatically verify a wide range of standard behavioral properties of Petri nets (such as boundedness-, home-, and liveness properties). In addition, state spaces of CPN models can be used to perform model checking of behavioral properties expressed using temporal logics such as CTL and LTL. The main limitation of state space methods is the inherint state exploration problem which implies that state spaces in their basic form are often too large to be with the available computing power. Many advanced techqniues exists to combat the state explosion problem, and most of these can be applied also in the context of CPN models. □

Sidebar: Performance Analysis. Simulation-based performance analysis is the main technique available for conducting quantative analysis of timed CPN models. The basic idea of simulation-based performance analysis is to conduct a number of lenghty simulations of the CPN model under consideration. During the lengthy simulations data on, e.g., on queue length and delays, are collected. Data collection is based on the concept of monitors that observes simulations and write the extracted date into log files for post processing or directly compute key performance figures such as averages, standard deviation, and confidence intervals. In addition, batch simulations can used to explore the parameter space of the model and conduct multiple simulation for each parameter in order to obtain statistically reliable results. □

5. CONCLUSIONS AND PERSPECTIVES

The development of the CPN technology has been driven by a research agenda with simultaneous focus on theoretical development, design and implementation of software tool support, and practical application and case studies. These three aspects has had a clear mutual influence in that as a theoretical foundation is needed in order to develop seman-

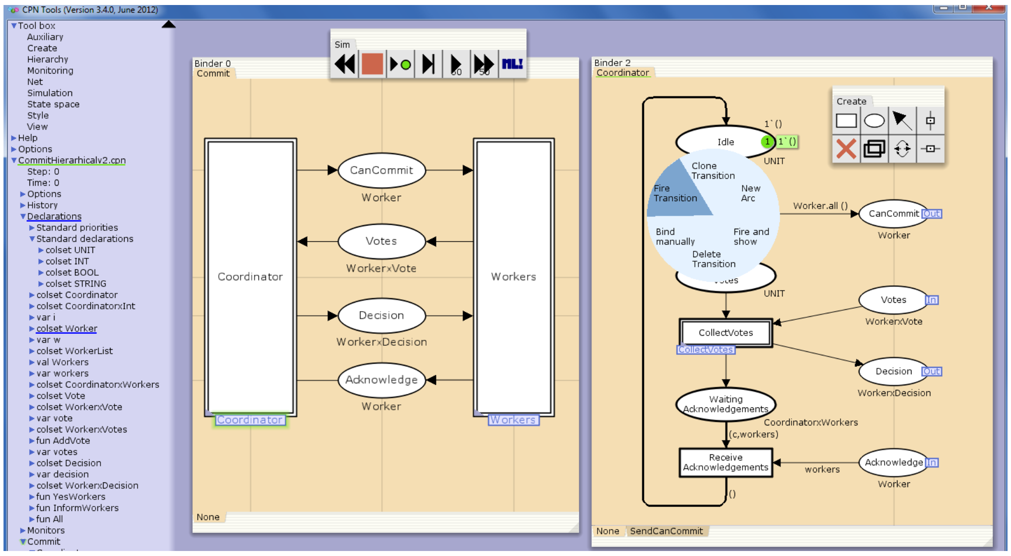

Figure 9: Two-phase Commit Protocol in CPN Tools

tically sound software tools which in turn is needed in order to get evidence on the practical applications and limitations of the theoretical foundation.

The development of the CPN modeling languages can be divided into four steps which has many similarities with the developments in programming languages. The first step from the black tokens of basic Petri nets to colored tokens in PrT-nets corresponds to the step from bit representation to simple data types. The second step from PrT-nets to CPNs corresponds to the invention of structured data types and type checking. The third step with the introduction of hierarchical CPNs corresponds to the introduction of structuring concepts like modules, procedures, functions and subroutines. The fourth step represented by the development of Design/CPN and CPN Tools corresponds to the implementation of compilers and run-time systems which are essential for programming languages in a similar way as a Petri net language is of no practical use without tool support. In addition to the historical development covered in this paper, an international standard for high-level Petri nets was developed headed by Billington [1] and approved in 2004. The high-level Petri net standard is heavily based on CPNs which adhere to the standard.

In this paper we have provided an overview and an introduction to the CPN language and its computer tool support and we have discussed the research development that led to the development of the CPN modelling language as it exists today. The most recent monograph on Colored Petri Nets has been published by Jensen and Kristensen in 2009 [7]. It provides an in-depth, but yet compact introduction to modeling and validation of concurrent systems by means of CPNs. The book introduces the constructs of the CPN modeling language, presents its analysis methods, and provides a comprehensive road map to the practical use of CPNs. Furthermore, the book presents some selected industrial case studies illustrating the practical use of CPN modeling and validation for design, specification, simulation, and verification in a variety of application domains. The book is aimed at use both in university courses and for self-study.

Acknowledgements. Jawahar Malhotra got the idea to use the Standard ML language as a basis for type definitions and net inscriptions in Design/CPN, Ole Bach Andersen implemented the graphical interface, and Søren Christensen implemented the complex inference algorithms for automatic binding of free variables during simulations. Hartmann Genrich constributed to the Design/CPNq with knowledge and experience from PrT nets. The graphical user interface of CPN Tools was designed together with Michel Beaudouin-Lafon and Wendy McKay.

6. REFERENCES

[1] J. Billington. ISO/IEC 15909-1:2004, Software and System Engineering - High-level Petri Nets - Part 1: Concepts, Definitions and Graphical notation.

[2] CPN Tools. Online documentation. http://cpntools.org/.

[3] H. Genrich and K. Lautenbach. System modelling with high level petri nets. *Theoretical Computer Science*, 13:109–136, 1981. North-Holland.

[4] P. Huber, K. Jensen, and R. Shapiro. Hierarchies in coloured petri nets. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 313–341. Springer, 1991.

[5] K. Jensen. Coloured petri nets and the invariant method. *Theoretical Computer Science*, 14:317–336, 1981. North-Holland.

[6] K. Jensen. *Coloured Petri Nets. Vol. 1: Basic Concepts, Vol. 2: Analysis Methods, Vol. 3: Practical Use.* Springer, 1992-1997. Monographs in Theoretical Computer Science.

[7] K. Jensen and L. Kristensen. *Coloured Petri Nets. Modelling and Validation of Concurrent Systems.* Springer, 2009. Web-page: www.cs.au.dk/CPnets/cpnbook/.

[8] K. Jensen et. al. *Design/CPN Reference Manual.*

Meta Software and Computer Science Department, Aarhus University, Denmark. On-line version available at: www.daimi.au.dk/designCPN/.

[9] I. U. of Coloured Petri Nets. www.cs.au.dk/CPnets/industrialex/.

[10] C. Petri. Kommunikation mit automaten. Technical report, Institut für Instrumentelle Mathematik, Bonn, 1962.