Design/CPN — A Computer Tool for Coloured Petri Nets

Søren Christensen, Jens Bæk Jørgensen, and Lars Michael Kristensen

Computer Science Department, University of Aarhus
Ny Munkegade, Bldg. 540, DK–8000 Aarhus C, Denmark
E-mail: {schristensen, jbj, kris}@daimi.aau.dk

Abstract. In this paper, we describe the computer tool Design/CPN supporting editing, simulation, and state space analysis of Coloured Petri Nets. So far, approximately 40 man-years have been invested in the development of Design/CPN. It is used world-wide by more than 200 companies and research institutions. For the presentation, we draw from the experiences gained in a recent industrial application using Coloured Petri Nets in the design, validation, and verification of communication protocols for audio/video systems.

1 Introduction

Coloured Petri Nets (CP-nets or CPN) [11,12] is a powerful graphical language for design, specification, validation, and verification of systems. CP-nets have a wide range of application areas and many projects have been carried out in the industry and documented in the literature, e.g., in the areas of communication protocols [6], operating systems [3], hardware designs [7], embedded systems [17], software system designs [18], and business process re-engineering [16].

The first graphical computer tool supporting CP-nets emerged in 1989, the editor and simulator Design/CPN [10]. The tool is developed in close cooperation between Meta Software Corporation, Cambridge, Massachusetts, and the CPN group at University of Aarhus, Denmark. Design/CPN is under ongoing development, with participation of the authors of this paper. Our aim here is to describe Design/CPN as it appears in 1997, focussing more on the involved concepts than on the user interface. In particular, we describe the recently developed support for formal analysis using state spaces, and the way in which it is integrated with simulation.

The paper is organised as follows: Section 2 sums up the industrial application used as running example throughout this paper. In Sect. 3, important concepts of the CPN formalism are informally introduced. Section 4 provides an overview of the architecture of Design/CPN. Sections 5, 6, and 7 describe the support for editing, simulation, and state space analysis, respectively. Section 8 concludes the paper by discussing related work and future plans for Design/CPN.

2 Example of Industrial Use — The B&O Project

As the main part of a two man-years CPN project [4], the renowned Danish manufacturer of audio/video systems Bang & Olufsen A/S (B&O) used Design/CPN for validation and verification of an important communication protocol. The protocol is part of B&O's BeoLink® system that connects the audio/video devices of a home in a network. This allows sharing of resources such that, e.g., a person can remotely use a CD player located in another room. In this paper, we refer to the CPN model of the protocol under consideration as the *B&O model.* The protocol is a mutual exclusion protocol ensuring exclusive access to various services. A device must possess a *key* in order to be allowed to perform critical actions, e.g., change of track on a CD. The purpose of the protocol is to prevent disorder, e.g., that track 11 is selected on a CD if two users request track 1 simultaneously. There is exactly one key in the system. The key is being passed between the devices upon request. Figure 1 depicts a typical communication.

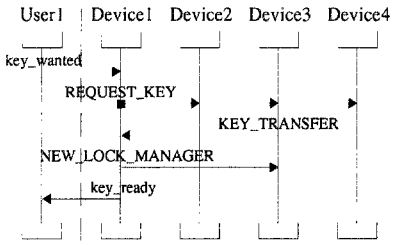

Fig. 1. Communication between devices.

Communications in the protocol are initiated by users. When a user presses the button on a remote control, internal actions inside a device are prompted, which results in a request for the key. In Fig. 1, assume that `Device1` is a CD player, operated by `User1` who wants to change track. The key is requested by `User1` with the event `key_wanted` sent to `Device1`. `Device1` does not have the key. Therefore, the key is requested on the network by broadcasting a `REQUEST_KEY` *telegram.* `Device1` is granted the key when the `KEY_TRANSFER` telegram is received. Now, a `NEW_LOCK_MANAGER` telegram is sent to the former key holder `Device3` as an acknowledgement. Finally, `User1` gets the event `key_ready`, and the change of track can take place.

The protocol was thoroughly validated using simulation, and important parts were formally verified by state space analysis. Moreover, Design/CPN was used for design, validation, and verification of a possible future version of the protocol under consideration.

3 Coloured Petri Nets

In this section, we give an informal introduction to CP-nets. The intention is to give an idea of the involved concepts, not to provide a complete description. The reader who is interested in a full and formal definition is referred to [11] or [12].

A CP-net is always created as a graphical drawing, a *CPN diagram*. An example can be seen in Fig. 2, which shows an extract from the B&O model. The extract models the actions taken in the mutual exclusion protocol, when a device receives a REQUEST_KEY telegram. The example is more complex than usual introductory examples in order to give a flavour of CPN models of real-world, industrial systems.

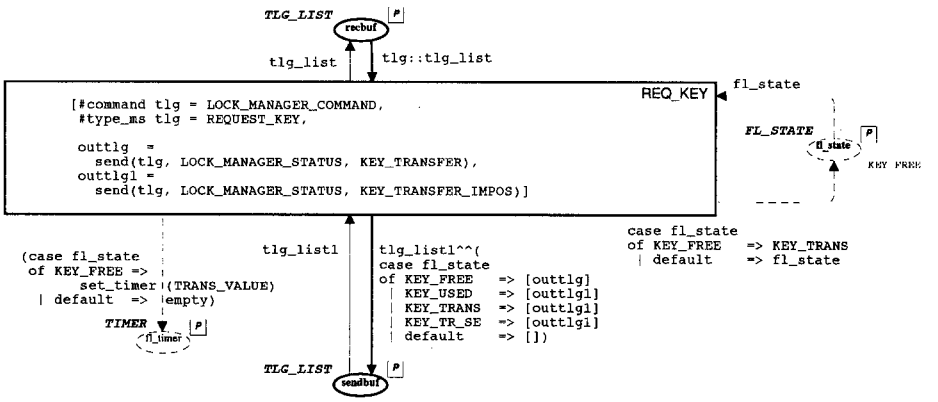

Fig. 2. Extract of CPN diagram.

A CP-net describes both the states and the actions of a system. Below we explain how. Moreover, we outline the operational semantics of CP-nets, and we introduce the concepts of time and modules.

Modelling of states. The state of a CP-net is represented using *places*, drawn as ellipses with a name positioned inside. A *marking* (or state) of a CP-net is a distribution of *tokens* on the places. The tokens carry data values (colours), and each place has a *type*[1] (colour set) which determines the kind of tokens which the place may contain. The type of a place is written in bold and italics at the top left of the place.

In Fig. 2, the places `recbuf` and `sendbuf` both have the type `TLG_LIST` denoting lists of telegrams ("TLG" abbreviates "telegram"). The two places model the receive and send buffers used to temporarily store incoming and outgoing telegrams respectively within a device. A list type is chosen to model that telegrams are handled in the order of reception. The place `fl_state` ("fl" abbreviates "function lock", which is B&O's name for the modelled protocol) has the type `FL_STATE`, used to model the state of a device with respect to the protocol, e.g.,

[1] An alternative and perhaps better name for Coloured Petri Nets might be "Typed Petri Nets". However, the term "Coloured" has a historical explanation, and it has stayed the most commonly used term.

to signal whether the device has the key or not. The place `fl_timer` is used for time-outs, as explained later.

Modelling of actions. The actions of a CP-net are represented using *transitions*, drawn as rectangles. Transitions and places are connected by *arcs*. The actions of a CP-net consist of transitions removing tokens from the places connected to incoming arcs (input places) and adding tokens to the places connected to outgoing arcs (output places). This is often referred to as the *token game*. The tokens removed and added are determined by *arc expressions*, which are positioned next to the arcs. E.g., the arc expression on the bold arc from `recbuf` to the transition (named `REQ_KEY`) is `tlg::tlg_list` and the arc expression on the thin outgoing arc to `recbuf` is `tlg_list`.[2] `tlg` is a *variable* of type `TLG`, i.e., a telegram, and `tlg_list` is a variable of type `TLG_LIST`.

Operational semantics. A transition, which is ready to remove and add tokens, is said to be *enabled* and may *occur*. There are two kinds of conditions that must be fulfilled for a transition to be enabled. The first kind is that appropriate tokens are present on the input places. More precisely, it must be possible to assign (bind) data values to the variables appearing on input arcs such that the arc expressions evaluate to tokens available on the input places. In Fig. 2, this means that one condition for enabling of `REQ_KEY` is that `recbuf` contains a non-empty list. In that case, the variable `tlg` can be bound to the head of the list, and the variable `tlg_list` to the tail. The expressions on the two input arcs from the places `sendbuf` and `fl_state` respectively are variables. A variable may be bound to any token (of the right type). Thus the only condition on enabling of `REQ_KEY` from these two places are that they are non-empty, i.e., contain at least one token each.

The second kind of condition comes from the *guard*, which is a boolean expression assigned to the transition. The guard must evaluate to true in order for the transition to be enabled. In Fig. 2, the guard is positioned in square brackets inside the transition. The commas between the constituents of the guard are interpreted as logical conjunctions. The first two equations of the guard check that a telegram, which is ready at `recbuf`, is of a type to be handled here. It must be a `LOCK_MANAGER_COMMAND` (i.e., pertain to this mutual exclusion protocol — there are many other telegrams on the network used for other purposes), and moreover a `REQUEST_KEY` telegram. The two remaining equations are used to construct an appropriate response in the variables `outtlg` and `outtlg1`.

An occurrence of the transition `REQ_KEY` models reception of a `REQUEST_KEY` telegram by a device. The result is an appropriate response. There are three possibilities: 1) The receiving device has the key and is willing to give it away: The marking of `fl_state` is `KEY_FREE`, and is changed to `KEY_TRANS` to signal that a key transfer is started — see the `case` expression on the dashed arc to `fl_state`. Also, a `KEY_TRANSFER` telegram is put on `sendbuf` — see the guard

[2] The operator `::` is the basic list constructor.

and the case expression on the bold arc to sendbuf.[3] 2) The receiving device has the key but is not willing to give it away: A KEY_TRANSFER_IMPOS telegram is put on sendbuf — can be seen as in the previous case. 3) The receiving device does not have the key: There is no response — the case expression on the bold arc to sendbuf evaluates to the empty list, and the marking of sendbuf remains unchanged.

Time. In Fig. 2, when a key transfer is started, a timer is set in order to be able to time out if an acknowledgement from the recipient does not arrive in due time. To capture this aspect of the protocol, the model is timed. Time is part of the CPN formalism [13]. In a *timed CP-net*, there is a *global clock*, *delays* are assigned to some arcs and transitions, and some tokens have *time stamps*. A token can only participate in an occurrence of a transition if it has a time stamp smaller than the global clock. After each step, the global clock is incremented such that at least one transition becomes enabled (if possible).

From the arc expression on the arc to the place fl_timer, it can be seen that, when a key transfer is started, a token is put on this place with a time stamp indicating that the token cannot participate in any occurrence before a time period of length TRANS_VALUE has elapsed. TRANS_VALUE is a globally defined symbolic constant specifying the length of the delay.

Modules. The structuring and reusability offered by *modules* are part of the CPN formalism. The B&O model consists of 13 modules, and we only consider a small fraction in Fig. 2. Here, the small box with a P inside near each place indicates that the place acts as an interface to the other modules of the full model: The places are in a well-defined way merged with places on other modules, thus allowing exchange of tokens between modules.

4 Tool Architecture

In this section, we give an overview of Design/CPN. The overall architecture is shown in Fig. 3.

The two main components are the *Graphical User Interface* (*GUI*) and the *Abstract Machine*. They communicate in the sense that graphical representations are transformed into abstract/internal representations and vice versa.

When a CPN diagram has been created in the *Editor*, the *Syntax Checker* of the *Compiler* is invoked to ensure that the model constitutes a legal CP-net. When this is the case, the *Simulation Code* can be generated by the *Simulation Code Generator*. Enabling and occurrence of transitions are calculated by the *Simulator* of the Abstract Machine, and the simulation is controlled and viewed by the user on the CPN diagram in the *Simulator* of the GUI. The Simulator of the Abstract Machine also contains the *State Space Code Generator* which is able to build the functions and data structures used for state space analysis. The

[3] The operator # extracts a field from a record and ^^ concatenates two lists.

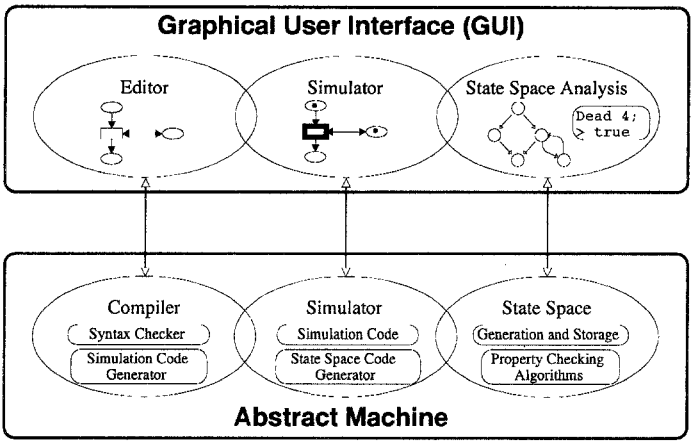

Fig. 3. Architecture of Design/CPN.

State Space part of the Abstract Machine provides functionalities for *Generation and Storage* of state spaces, and it contains the *Property Checking Algorithms* which are available for the user through the *State Space Analysis* part of the GUI.

The GUI is built on top of the general graphical package Design/OA [5] and the Abstract Machine is built on top of the Standard ML (SML) New Jersey compiler [1]. The GUI and the Abstract Machine run as two separate communicating processes.

A language called CPN ML is used for declarations (of types, etc.) and net inscriptions (arc expressions, guards, etc.) in CP-nets. CPN ML is SML [15] extended with some syntactical sugar to ease declarations of types, variables, etc. The fact that the inscription language is based on SML has several virtues. Firstly, the expressiveness of SML is inherited by Design/CPN. Secondly, SML is strongly typed allowing many modelling errors to be caught early. Thirdly, SML is a functional language — evaluation of SML expressions has no side effects, which is consistent with the operational semantics of CP-nets. It would not make sense if, e.g., evaluation of a guard for a transition might have an impact on the marking of some places. A forth virtue is that polymorphism and definition of infix operators in SML allow net inscriptions to be written in a natural, mathematical syntax. Finally, SML is well documented (see, e.g., [19]), tested, and maintained. The choice of SML has turned out to be one of the most successful design decisions for Design/CPN. The main drawback is that the generality, of course, has a negative impact on the size and speed of the Abstract Machine.

It is an advantage to build Design/CPN upon Design/OA and SML, which are both available on different platforms, since the platform dependency is isolated in these building blocks and not in the tool itself.

5 Editor

The Design/CPN editor supports construction, modification, and syntax check of CPN diagrams. In typical industrial applications, a CPN diagram consists of 10-100 modules with varying complexities. A modeller must find a suitable way to divide the model into modules. Moreover, the modeller must find a suitable balance between net structure (i.e., places, transitions, and arcs), declarations, and net inscriptions. We list and discuss important features of the editor below.

Overview of modules. Design/CPN provides an overview of the modules of a CP-net and their interrelation by automatically creating and maintaining a so-called *hierarchy window*, which is similar to project managers known from conventional programming environments. The hierarchy window for the B&O model is shown in Fig. 4.

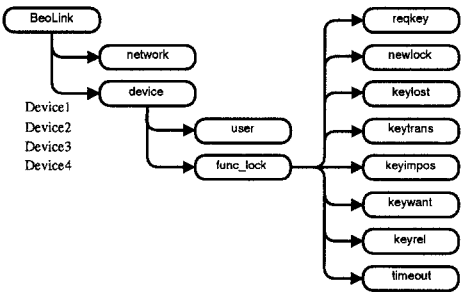

Fig. 4. Hierarchy window.

The hierarchy window represents each of the 13 modules as a node. There is an arc from one module to another, if the first has parts that are described in more detail in the second. E.g., the topmost module `BeoLink` consists of a network and a number of devices, where the details are modelled by other appropriately named modules. From the hierarchy window, the modules can be opened, i.e., the user can browse the modules constituting the CPN model.

Flexible graphics. In order to be able to create easily readable CP-nets, Design/CPN supports a wide variety of graphical parameters such as shapes, shading, borders, etc. The underlying formal CPN model (in the Abstract Machine of Fig. 3) is unaffected by the graphical appearance. E.g., an object created as a place remains a place forever, independent of graphical changes. Flexible graphics is in Design/CPN accompanied by sensible defaults, e.g., the default shape of a place is an ellipse. Figure 2 is an example on the use of flexible graphics. The main flow goes from the topmost place `recbuf` to the bottommost place `sendbuf`, as indicated by the thick borders of the places and the bold arcs.

Syntax checking. The editor enforces a number of built-in syntax restrictions, and thereby prevents the user from making certain syntax errors during the construction of a model. It is, e.g., not possible to draw an arc between two transitions or between two places. However, it is impossible to catch all syntax errors efficiently that way. Hence, there is a syntax checker, which can be invoked when the user wants to ensure that the created model constitutes a legal CP-net.

Reporting of syntax errors is based on a hypertext facility. We illustrate this by explaining how an error in the extract from Fig. 2 may be reported. Assume that we made an error during the editing of the module `reqkey` that includes the considered extract. In the hierarchy window (Fig. 4), an *error box* will appear. The error box will contain a text saying that there is an error in the module `reqkey`, and there will be a *hypertext link* pointing to the erroneous module. Following this link will open the module `reqkey` and select another error box with a further description of the problem. An example of an error may be that an arc expression has a type which is different from the type of the place of the arc. Several errors can be reported at the same time during a syntax check.

In many cases, correcting syntax errors only involve local changes. For efficiency reasons, the syntax check is incremental. This means that only the modified part of the model is syntax checked again — not the entire model. E.g., assume that all 13 modules of the B&O model depicted in Fig. 4 have been syntax checked. When a syntax error regarding the transition `REQ_KEY` has been fixed, only that transition and its surrounding arcs are rechecked, not all 13 modules.

6 Simulator

The Design/CPN simulator supports execution of CPN models. Simulation of CPN models has many similarities with debugging of programs written in high-level languages such as Pascal or C. Design/CPN provides different modes of simulation suitable for different purposes. We list and discuss important features of the simulator below.

Simulation control. In the early phases of a modelling process, the user typically wants to make a detailed investigation of the behaviour of the individual transitions. The simulator here plays the role of a single-step debugger. For this purpose, Design/CPN offers an *interactive mode*: The user is in full control, sets breakpoints, chooses between enabled transitions, possibly changes markings, and studies the token game in detail. Typically, a few steps per minute are executed. Interactive simulations do not require the model to be complete, i.e., the user can start investigating the behaviour of parts of a model and directly apply the gained insight in the ongoing design activities. Often, a model is gradually refined — from a crude description towards a more detailed one.

Later on in a modelling process, the focus shifts from the individual transitions to the overall behaviour of the full model. The *automatic mode* of Design/CPN is suitable here: The simulator itself makes random choices between

enabled transitions, and the token game is not displayed; feedback has a different form, e.g., graphical animation or write to a file. Many steps are executed in a short time. This is achieved even for large models, because the enabling and occurrence rules of Petri Nets are local. This means that, when a transition has occurred, only enabling of the nearest transitions need to be recalculated, i.e., the number of steps per second is independent of the size of the model. The speed in automatic mode is high, typically more than 1,000 steps per second.

Viewing interactive simulations. In Fig. 5, a snapshot from an interactive simulation is depicted.

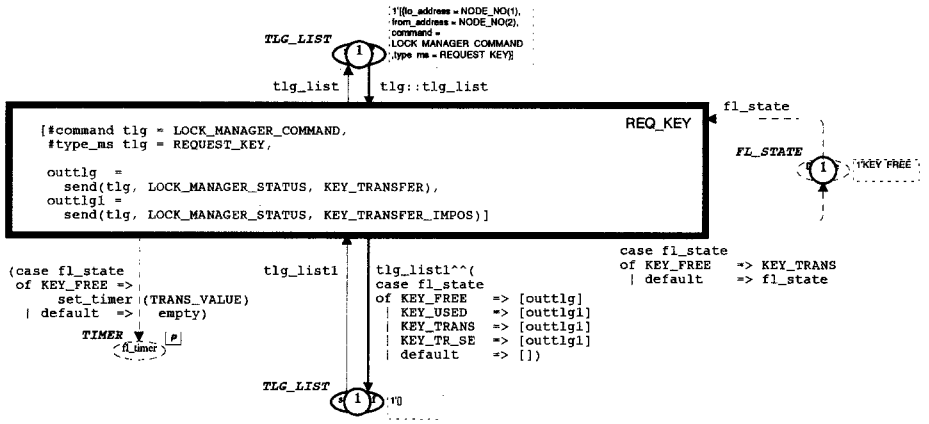

Fig. 5. Visualisation of interactive simulation.

The current marking is indicated: The number of tokens on a place is contained in the circle on top of the place. The absence of a circle corresponds to zero tokens. The data values of the tokens are shown in the box with a dashed border positioned next to the place. If desired, the user can make a box invisible, e.g., if the data values are irrelevant or too big to print. The transition is shown with a thick border to indicate that it is enabled in the current marking.

Viewing automatic simulations. Before and after an automatic simulation, the current marking and the enabled transitions are indicated as described above. Of course, this is typically less information than desired. The user wants to know what happened during an automatic simulation. Design/CPN supports that a log of the transitions executed are saved in a text file. This is full information, but the representation may be inappropriate.

Often, the user wants to define his own abstract view focussing on a particular aspect of the model. In the B&O model, the goal was to study the telegrams exchanged between devices. Hence, the model was instrumented to give feedback in terms of a *message sequence chart* as the one already shown in Fig. 1. It was

much easier and more efficient for the involved B&O engineers to discuss results of simulations in terms of the familiar message sequence charts than in terms of the token game. It was also important for presentation purposes, since simulation results could easily be discussed with colleagues not familiar with CP-nets.

Message sequence charts in Design/CPN are supported by a library, which was easily built due to the generality, expressiveness, and flexibility provided by Design/OA and SML.

Integration with the editor. Often, a simulation results in the desire to modify the model. Some of these modifications can be made immediately: It is possible to make minor changes while remaining in the simulator, e.g., to edit an arc expression. Other modifications require more involved rechecking/regeneration of the simulator code, which is only supported in the editor, e.g., it is not possible to add or change a type in the simulator. In Design/CPN, the editor and simulator are closely integrated. Therefore, it is easy and fast to switch from the simulator back to editor, fix a problem, re-enter the simulator, and resume simulation.

7 State Space Analysis

The *state space* of a CP-net is a directed graph with a node for each reachable state and an arc for each possible state change. If the state space is finite, it can be used to prove an abundance of properties, e.g., reachability, boundedness, liveness, and fairness.

Design/CPN supports generation, analysis, and drawing of state spaces for CPN models (timed as well as un-timed). The well-known state explosion problem is, of course, a practical obstacle. Whether a model can be analysed is determined by the amount of memory of the computer and the complexity of the model. Design/CPN has been used to handle state spaces with up to 400,000 nodes and 1,000,000 arcs.

We list and discuss important features of support for state space analysis in Design/CPN below.

Generation control. Often, a state space is so huge that it cannot be fully generated. Thus, the user is forced to focus on certain aspects of the model, corresponding to generating only subsets of the state space. For this purpose, Design/CPN provides *stop options* and *branching options*. Stop options are used to terminate the generation, e.g., when a certain number of nodes has been generated. Branching options enable the user to specify that, for some states, no successors should be generated. An example of a use of branching options comes from the B&O model, where the investigated protocol governs a key. When the system starts from scratch, no key exists. The protocol must ensure that a key is initially generated. Thus, a branching option was used to specify that, for states where a key is present, no successors should be generated. In

this way, a partial state space was generated, and it was used to formally prove that the protocol does in fact ensure that a key is always generated. For a system with four audio/video devices, this partial state space had 13,420 nodes and 41,962 arcs. On a Sun Ultra Sparc Enterprise 3000 computer with 512 MB RAM, the generation took about three minutes, and the desired analysis results were subsequently achieved within a few seconds.

It is also possible to generate (parts of) a state space interactively. Here the user specifies a state, and Design/CPN then calculates all direct successors. This is typically used in conjunction with drawing of parts of state spaces (see below).

Queries. The aim of generating a state space is to check whether the considered model has certain properties. Some *standard queries* are relevant for many models, e.g., to give generation statistics (number of nodes and arcs), list of dead states, and information on liveness of transitions. Design/CPN supports that the results of the standard queries are automatically saved in a textual report. Negative answers to standard queries are constructive, i.e., they help the user investigate why an expected property does not hold. E.g., if an unexpected dead state is found, a shortest path from the initial state to the dead state is provided as helpful information.

Other queries depend on the model being investigated. Design/CPN provides a general query language implemented in SML for that purpose. An example of a model-dependent query from the B&O project is to find all states in which a given device has the key.

In addition to the standard SML-based query language, the Design/CPN library ASK-CTL [2] allows analysis of state spaces by means of a CTL-like temporal logic. It is not only possible to formulate queries about states, but also queries about state changes.

Drawing. Since state spaces often get large, it rarely makes sense to draw them in full. However, the result of a query is often a set of nodes and/or arcs possessing certain interesting properties, e.g., a path in the state space leading from one state to another. A good and quick way in which to get detailed information on a small number of nodes and arcs is to draw the necessary part of the state space.

In Design/CPN, part of the state space can be drawn either manually or automatically. An example of a drawing of a selected part of the state space from the B&O model can be seen in Fig. 6. Node 37 corresponds to the state which was shown in Fig. 5, where the transition `REQ_KEY` is enabled. The occurrence of this transition corresponds to the arc leading to node 43. Design/CPN provides *descriptors* for nodes and arcs. For the nodes, the descriptors typically show the marking of certain places; in Fig. 6, the places `recbuf` and `sendbuf`. For the arcs, the descriptors typically show the occurring transition and the binding of some of its variables. The descriptors have sensible defaults but may be customised by the user, thus offering a flexible way in which to define a view on the state space.

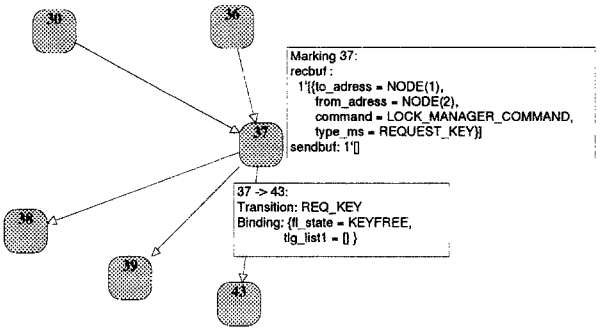

Fig. 6. Drawing of a state space.

Integration with the simulator. During a modelling process, the user often switches between state space analysis and simulation. Design/CPN supports transfer of a state from the generated state space to the simulator. The new state of the simulator is displayed graphically as usual on the CPN diagram. The user may now start a simulation from this state. E.g., transferring the state corresponding to node 37 in Fig. 6 into the simulator results in Fig. 5. In this way, the simulator can be used as a quick way of viewing a state fetched from the state space. Transferring the simulator state into the state space is supported as well. If the simulator state is not already included in the state space, the state will be added. Otherwise, the number of the node corresponding to the simulator state is returned to the user. A typical use of this feature is to investigate all possible states reachable within a few steps from the current simulator state. Here the user transfers the simulator state into the state space. Now, all successor states can easily be found and drawn as explained above. In contrast, the simulator itself can, of course, only be used to investigate one execution at a time.

Time and space efficiency. The major time-consuming task when generating a state space is to check if a given state is already included. Design/CPN uses hash coding to speed up this check. The hash coding maps all states with the same number of tokens on all places to the same key.

For many states, only the markings of a few places differ. Because of that, Design/CPN uses sharing when storing a state space. Information is shared on two levels: Firstly, the states are separated into a part for each module of the model — which means that a transition often only changes one of these parts. The other parts can be shared immediately. Secondly, the representations of the markings of places are stored only once. All places, having the same marking, will refer to this one representative.

Condensation methods. Even with the considerations above taken into account, the sizes of the state spaces remain a problem, and improvements are

needed. Several methods for construction of smaller state spaces have been proposed. One approach relies on the observation that many models have states that are very similar — they are in a well-defined way equivalent [13]. Design/CPN has recently been extended to support this method through the library the Design/CPN OE/OS Tool [10].

8 Conclusions

We now conclude the paper by considering related tools for construction and analysis of systems, and future plans for Design/CPN. We discuss one good Petri Net tool and two other well-known and ingenious tools. The two latter are not based in Petri Nets. The three tools are described, first by listing important virtues, then by listing drawbacks compared to Design/CPN.

PEP [8] is another Petri Net based tool. Compared to Design/CPN, PEP has a more modern, graphical user interface, and it allows process algebraic specification of systems as well. The main drawback of PEP is that, before verification, a system specification is always translated (unfolded) into an ordinary (low-level) Petri Net. This approach entails a serious complexity problem with respect to analysis of many real-world systems. Design/CPN does all analysis directly on the given CP-net without unfolding.

SPIN [9] is a widely used tool for design and analysis of systems. Like Design/CPN, SPIN supports editing, simulation, and state space analysis of models. Input to SPIN is given in the C-like textual language PROMELA. With respect to formal verification, SPIN is currently more sophisticated than Design/CPN. SPIN includes, e.g., the bit state hashing technique and partial order reduction as means to alleviate the state explosion problem. We believe that the approach to modelling by drawing is one of the main virtues of Design/CPN. It is as easy and convenient to structure a large model as a set of graphical modules with well-defined relations between them in Design/CPN, as it is to create modules of text in SPIN. Moreover, although it certainly requires expertise to create models in Design/CPN, they can often be easily understood also by non-experts because of the appealing graphics. With respect to simulation, SPIN primarily presents the results in terms of message sequence charts. Design/CPN is more open, and allows users to customise the feedback, e.g., [17] describes use of advanced graphical feedback in Design/CPN to design an alarm system.

SMV [14] is a tool using binary decision diagrams (BDDs) to obtain space-efficient storage of state spaces. Like in SPIN, input to SMV is given in a textual language. SMV is capable of analysing systems with indeed very many states, and has proven highly useful for verification of systems in which the states have a simple (in some sense) description. SMV is tuned for design and analysis of hardware and hence not as general as Design/CPN. The type concept of SMV is restricted to simple types like booleans and enumeration types. It is unknown whether the BDD technique can be effectively generalised to CP-nets with their very elaborated notion of state. If it is possible, it may induce a dramatic im-

provement of Design/CPN. The support for simulation in SMV is limited. SMV only supports what corresponds to the interactive mode of Design/CPN.

Many improvements and extensions are interesting for future versions of Design/CPN. Here we mention two important ones. A new version of the simulator is being built. Several data structures and internal algorithms have been redesigned and reimplemented. Experimental measures show that, for many models, the new simulator runs about a thousand times faster than the old. So far, three man-years have been invested in the design and implementation of the new simulator, but one man-year is still needed. The second extension is to support other formal analysis methods in addition to state space analysis. Design/CPN will be extended with a part for place invariant analysis. An early prototype exists [10]. It is able to check the validity of (many) proposed place invariants without generating all reachable states. Instead, the check is done locally — for each transition, it is checked that no occurrence can change the proposed invariant. Only the arc expressions and guards need to be considered. The check is in general undecidable, but indeed possible for many CPN models appearing in practical applications.

Design/CPN is a complex and comprehensive tool, and in this paper we have merely given an overview. More information on CP-nets and Design/CPN is available on the World Wide Web at `http://www.daimi.aau.dk/designCPN/`, e.g., a tutorial, a user's manual, many examples, and more on future plans. Also, here it is described how to get a free-of-charge copy of the tool.

Acknowledgements. We thank the numerous people involved in the development of Design/CPN, in particular Peter Huber, Kurt Jensen, and Robert Shapiro. We thank Niels Toft Sørensen, B&O, who was responsible for developing the B&O model used as example. Finally, we thank Allan Cheng, Kjeld Høyer Mortensen, and Kim Sunesen for comments and proof-reading.

The work on this paper has been supported by grants from University of Aarhus Research Foundation and the Faculty of Science at University of Aarhus.

References

1. A.W. Appel and D.B. MacQueen. Standard ML of New Jersey. In J. Maluszyński and M. Wirsing, editors, *Third International Symposium on Programming Languages Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.
2. A. Cheng, S. Christensen, and K.H. Mortensen. Model Checking Coloured Petri Nets Exploiting Strongly Connected Components. In M.P. Spathopoulos, R. Smedinga, and P. Kozák, editors, *Proceedings of the International Workshop on Discrete Event Systems, WODES96*. Institution of Electrical Engineers, Computing and Control Division, Edinburgh, UK, 1996.
3. L. Cherkasova, V. Kotov, and T. Rokicki. On Scalable Net Modeling of OLTP. In *Proceedings of the 5th International Workshop on Petri Nets and Performance Models, Toulouse, France*. IEEE Computer Society Press, 1993.
4. S. Christensen and J.B. Jørgensen. Analysing Bang & Olufsen's BeoLink® Audio/Video System Using Coloured Petri Nets. Technical report, Computer Science Department, University of Aarhus, Denmark, 1996.

5. Meta Software Corporation. *Design/OA*. Meta Software Corporation, 150 CambridgePark Drive, Cambridge MA 02140, USA.
6. D.J. Floreani, J. Billington, and A. Dadej. Designing and Verifying a Communications Gateway Using Colored Petri Nets and Design/CPN. In J. Billington and W. Reisig, editors, *Proceedings of the 17th International Conference on Application and Theory of Petri Nets, Osaka, Japan*, volume 1091 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
7. H.J. Genrich and R.M. Shapiro. Formal Verification of an Arbiter Cascade. In K. Jensen, editor, *Proceedings of the 13th International Conference on Application and Theory of Petri Nets, Sheffield, UK*, volume 616 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
8. B. Grahlmann and E. Best. — PEP — More than a Petri Net Tool. In T. Margaria and B. Steffen, editors, *Proceedings of TACAS96, the Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Passau, Germany*, volume 1055 of *Lecture Notes in Computer Science*, 1996.
9. G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International Editions, 1991.
10. K. Jensen. Design/CPN Online, Computer Science Department, University of Aarhus, Denmark. Online: `http://www.daimi.aau.dk/designCPN/`.
11. K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.
12. 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, 1993.
13. K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Vol. 2, Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, 1994.
14. K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
15. R. Milner, R. Harper, and M. Tofte. *The Definition of Standard ML*. MIT Press, 1990.
16. K.H. Mortensen and V. Pinci. Modelling the Work Flow of a Nuclear Waste Management Program. In R. Valette, editor, *Proceedings of the 15th International Conference on Application and Theory of Petri Nets, Zaragoza, Spain*, volume 815 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
17. J.L. Rasmussen and M. Singh. Designing a Security System by Means of Coloured Petri Nets. In J. Billington and W. Reisig, editors, *Proceedings of the 17th International Conference on Application and Theory of Petri Nets, Osaka, Japan*, volume 1091 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
18. G. Scheschonk and M. Timpe. Simulation and Analysis of a Document Storage System. In R. Valette, editor, *Proceedings of the 15th International Conference on Application and Theory of Petri Nets, Zaragoza, Spain*, volume 815 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
19. J.D. Ullman. *Elements of ML Programming*. Prentice-Hall, 1993.