Game Coloured Petri Nets

M. Westergaard

Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark,
Email: mw@daimi.au.dk

Abstract. This paper introduces the notion of game coloured Petri nets. This allows the modeler to explicitly model what parts of the model comprise the modeled system and what parts are the environment of the modeled system. We give the formal definition of game coloured Petri nets, a means of reachability analysis of this net class, and an application of game coloured Petri nets to automatically generate easy-to-understand visualizations of the model by exploiting the knowledge that some parts of the model are not interesting from a visualization perspective (i.e. they are part of the environment, and not controllable by the system itself, or they are part of the system itself and therefore we need not worry about them).

1 Introduction

The coloured Petri nets (CPNs or CP-nets) [14] formalism has proven itself useful for modeling concurrent systems such as network protocols [11,12,17,18,21] and workflows [4,15]. One problem that is often encountered is how to distinguish between the system itself and its environment [30], as we would often like to make the assumptions about the environment explicit even though they are not directly part of the system we want to model. Normally we would model the environment simply as a part of the model and at best accompany the model by an informal textual description of what parts of the model comprise the environment or, at worst, let this information be implicit. If the modeler is a bit more thorough, he will split the model up into separate modules and put the environment in certain modules and the modeled system in other modules. The problem with the first of approach is that the information about which parts of the model are the actual system is not part of the modeled system, thereby making it impossible to use this information during analysis or other treatments of the model. If, on the other hand, we put the model of the environment in separate modules, we may obtain an unnatural model in which the natural flow of the system is not readily visible if the flow consists of frequent interleavings of actions between the modeled system and the environment.

In this paper we will study another way to model the environment for coloured Petri nets. The idea is to separate the actions of the system into two parts, the controllable and the uncontrollable actions. The controllable actions are, as the name suggests, controllable by the system we model. The system can

choose which and when to execute controllable actions, e.g. transmit a packet onto the network. The uncontrollable actions are not controllable by the system, but can occur whenever they are enabled. The inspiration of this is classical games such as tic-tac-toe, in which two players, cross and naught, play against each other. Say we model the game from the point of view of the player drawing crosses. The action of adding a new cross to the board is controllable, whereas the action of adding a naught is uncontrollable.

While it is interesting in itself to be able to study small toy-games such as tic-tac-toe using coloured Petri nets, the real power of separating the actions into uncontrollable and uncontrollable ones comes when regarding more realistic examples, e.g. a model of a network protocol. The network protocol contends against the network, which may lose packets, duplicate packets, or even alter packets. In this case all actions of the protocol (such as retransmitting a packet) are controllable, whereas the actions of the environment (such as transmitting or dropping the packet) are uncontrollable. This illustrates how separating the actions into controllable and uncontrollable actions allow us to explicitly specify what parts of the model comprise the system and what parts comprise the environment. Actions of the system are modeled as controllable transitions (by convention drawn as a rectangle) and actions of the environment are modeled as uncontrollable transitions (drawn as a dashed rectangle). Allowing this distinction directly in the model has several uses. Firstly, it alleviates the need for an informal textual description of which parts of the model are the modeled, interesting system, and which parts just make the assumptions of the environment explicit. Secondly, we can use the distinction to do better analysis of the properties of the system, and to automatically generate strategies (corresponding to programs) making sure, e.g. that the system always reaches a state where all packets have been successfully transmitted. Thirdly, we can use the extra information to generate visualizations of the system allowing users to interact with the model without looking at the model. We could have chosen to distinguish controllable actions from uncontrollable actions on the level of tokens or transition modes, but have chosen to make the distinction on the level of transitions for simplicity. Refer to the conclusion (Sect. 6) for a brief discussion of another way to do the distinction.

The contribution of this paper is three-fold: Firstly, the introduction of game coloured Petri nets, secondly, the adaptation of an algorithm for reachability analysis of finite games [2] to game coloured Petri nets, and, thirdly, an application of game coloured Petri nets to automatically tie CPN models to visualizations.

The rest of this paper is structured as follows: Firstly, in Sect. 2, we will give an informal introduction to game CP-nets and introduce a simple example, which will be used in the rest of the paper. Secondly, in Sect. 3, we will formalize the notion of game coloured Petri nets (game CPNs or game CP-nets). In Sect. 4, we outline how to do reachability analysis of game CP-nets. After that we will turn to a simple yet powerful application of games to automatically generate visualizations of CPN models. Finally, we will give our conclusions and provide

directions for future research in Sect. 6.

2 Example

In this section, we will introduce a simple example game CP-net, which we will use in the following sections. The example is a slight modification of one of the sample nets supplied with CPN Tools, namely a simple stop-and-wait protocol.

The modified example can be seen in Fig. 1. In the example, a sender, on the left, wants to transmit some packets lying on the place Send. The sender has a counter telling which packet to send next, NextSend. When the transition Send Packet occurs, it puts the packet onto the network (place A). Now the network, in the middle of the model, can either choose to Transmit Packet or Drop Packet. These two actions are uncontrollable, modeling that the protocol has no control over what happens on the network. If the packet is dropped, it is simply removed from place A, otherwise it is moved onto place B, where the receiver, on the right of the model, can Receive Packet. When this happens, either the packet is received for the first time or it is a retransmission. The receiver keeps track of which packet it expects by a counter on the place NextRec. If the sequence number of the incoming packet is not equal to the expected sequence number, the packet is discarded. Otherwise, the new packet is stored on the place Received. In either case, an acknowledgment is sent back to the network by putting a token on place C, containing the sequence number of the next expected packet. The acknowledgment can be either transmitted or dropped by the network. If it arrives safely on place D the sender can Receive Ack, and update the number of the next packet to send, and start a new cycle sending the next packet.

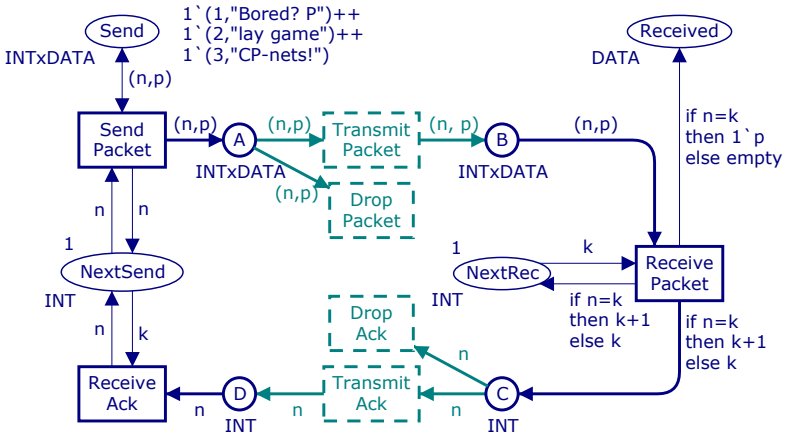

Fig. 1. A simple stop-and-wait protocol modeled using game CP-nets.

In this example we have modeled all actions of both sender and receiver as controllable and all actions of the network as uncontrollable. That is, we regard the network protocol as the modeled system and the network as the environment. We could have modeled the actions of e.g. the receiver as uncontrollable as well, thereby only regarding the sender as the modeled system.

3 Formal Definition

In this section we will give a formal definition of game coloured Petri nets (game CPNs or game CP-nets). The intuition of the definition is that we partition the set of transitions into controllable and uncontrollable transitions. We will do this by first recalling the definition of standard coloured Petri nets.

We will assume that the relations $<$, $=$, $\leq$, $>$, and $\geq$, and operations $+$ and $-$, on multi-sets are defined as usual. We use $\mathbb{N}^S$ to denote the set of all multi-sets over S.

Definition 1 (Coloured Petri net (Def. 5.1 and 5.2 in [13][1])). *A* coloured Petri net *is a tuple,* $\mathcal{CPN} = (P, T, D, Type, Pre, Post, M_0)$*, where*

- P *is a finite set of* places*,*
- T *is a finite set of* transitions *such that* $P \cap T = \emptyset$*,*
- $D \neq \emptyset$ *is a finite set of non-empty* types*,*
- $Type : P \cup T \rightarrow D$ *is a* type function *assigning a type to each place and transition,*
- $TRANS = \{(t, m) \mid t \in T, m \in Type(t)\}$ *is the set of all* transition modes*,*
- $\mathbb{N}^{PLACE} = \mathbb{N}^{\{(p,g) \mid p \in P, g \in Type(p)\}}$ *is the set of all* markings*,*
- $Pre, Post : TRANS \rightarrow \mathbb{N}^{PLACE}$ *are the* backward *and* forward incidence functions*, assigning to each arc an* annotation*, and*
- $M_0 \in \mathbb{N}^{PLACE}$ *is the* initial marking*.*

The state of a CP-net is given by a marking of the places, which is a multi-set, $M \in \mathbb{N}^{PLACE}$.

Definition 2 (Enabling and occurence of transitions (Def. 5.3.1 and 5.4 in [13])). *A transition mode,* $(t, m) \in TRANS$*, is* enabled *in marking* $M \in \mathbb{N}^{PLACE}$ *if* $Pre(t, m) \leq M$*. A transition* $t \in T$ *is enabled if there exists* $m \in Type(t)$ *such that* (t, m) *is enabled. If* (t, m) *is enabled in* M*, it may* occur *and lead to a marking* M'*. This is written* $M \stackrel{(t,m)}{\rightarrow} M'$*, where* M' *is defined by* $M' = M - Pre(t, m) + Post(t, m)$*.*

Definition 3 (Game coloured Petri net). *A* game coloured Petri net *(game CP-net or GCPN) is a tuple,* $\mathcal{GCPN} = (P, T_c, T_u, D, Type, Pre, Post, M_0, W)$*, where*

[1] In this paper we will use the term coloured Petri net rather than high-level Petri net as used in [13].

- T_c *is a finite set of* controllable transitions,
- T_u *is a finite set of* uncontrollable transitions *such that* $T_c \cap T_u = \emptyset$,
- $\mathcal{CPN} = (P, T_c \cup T_u, D, Type, Pre, Post, M_0)$, *the* underlying coloured Petri net, *is a coloured Petri net, and*
- $W : \mathbb{N}^{PLACE} \rightarrow \{\mathtt{tt}, \mathtt{ff}\}$ *is a predicate identifying* winning markings.

The notions of markings, transition modes, enabling, and occurrence for game CP-nets are the same as for coloured Petri nets. Furthermore, we shall allow the notation $TRANS_c = \{(t, m) \,|\, t \in T_c, m \in C(t)\}$ for the *controllable transition modes* and $TRANS_u = \{(t, m) \,|\, t \in T_u, m \in C(t)\}$ for the *uncontrollable transition modes*

One thing to note is that we do not impose any ordering on controllable or uncontrollable actions. We could have required that a controllable action must always be followed by an uncontrollable action and vice versa, but we will not do this, as enabling suddenly becomes dependent on the level of detail used to model the system. If such behavior is required (e.g. in the case of the tic-tac-toe game), it must be modeled explicitly. We will discuss this further in Sect. 5.4, where we talk about fairness when using game CP-nets to automatically generate visualizations of models.

4 Analysis

For finite games, i.e. games constructed by separating the actions of finite automata into controllable and uncontrollable actions, we can do several kinds of analysis. One of the simplest properties we can check is a reachability problem, namely whether it is possible to find a strategy ensuring we will end up in one of the winning states. A strategy is a mapping from states to controllable transitions. A strategy is a *winning strategy* iff we are ensured, no matter what uncontrollable transitions are executed, to end up in one of the winning states if we in every state pick the transition specified by the strategy. The reachability property is interesting when the set of the winning states, W, represent desirable final markings, e.g. that all packages have been received successfully. A winning strategy corresponds to a recipe for what the modeled system should do in order to ensure it will end up in a desirable state. Using a solution for the simple reachability problem, we can also solve the dual safety problem, whether we can ensure that no matter which uncontrollable transitions are executed, we will never reach a state which is not winning. This property is interesting for reactive systems, where the set W corresponds to states where nothing bad has happened, e.g. that the system has not dead-locked or received a damaged packet.

Rather than going through the definitions required to solve the reachability problem directly for game CP-nets, we will go through how to translate the reachability problem for game CP-nets into a reachability problem for finite state systems. This, of course, cannot be done in general (as the reachability graph of the underlying CP-net can be infinite), but we do it in a way that

ensures that if the reachability graph of the underlying CP-net is finite, the algorithm will terminate with the correct answer.

In [5], Cassez, David, Fleury, and Larsen instantiate an algorithm by Liu and Smolka from [19] to obtain an efficient (and optimal) algorithm to decide whether a given finite reachability game has a winning strategy and to extract that strategy. The intuition of the algorithm is to calculate a minimal fix-point of all good states, Win, where all states in W are good and all states where we can take a controllable step to a good state and all uncontrollable steps lead to a good state are good. This corresponds to the intuition that in any given state, we have a winning strategy if we can execute a controllable transition and end in a new state where we have a winning strategy, and that no matter what the opponent does, we end up in a state where we have a winning strategy. The algorithm assumes a finite game as a tuple $(Q, q_0, Act_c, Act_u, \delta, Goal)$ (all states, the initial state, the controllable and uncontrollable actions, the transition relation, and the winning states). The adaptation of the algorithm to game CP-nets is easy. Given a game CP-net, $\mathcal{GCPN} = (P, T_c, T_u, D, Type, Pre, Post, M_0, W)$, we will assume that all types in D are finite and that the number of tokens on all places are bounded. Then we can take $Q = \mathbb{N}^{PLACE}$, $Act_c = TRANS_c$, and $Act_u = TRANS_u$, all of which are finite. We set $Goal = W$, $q_0 = M_0$, and let $\delta = \{(M, (t, m), M') \mid M \overset{(t,m)}{\rightarrow} M'\}$. We then obtain Algorithm 1. The algorithm works by forward traversal of the reachability graph. Whenever a state is found to be winning, it is added to Win (ll. 3, 10, and 19). Whenever we find a new state, we mark it as dependent on the winning status of its successors (ll. 9 and 22). When a state is marked as winning, we schedule all states dependent on its winning status for re-evaluation (ll. 13 and 18). With a little cleverness in the implementation of line 16[2], the algorithm is shown to find a winning strategy in time linear in the number of nodes and edges in the reachability graph. We use the standard notation that the empty conjunction is tt and the empty disjunction is ff.

A nice property of this algorithm is that the while loop (ll. 5–25) has the invariant that if $\mathsf{Win}[M] = \mathtt{tt}$ then there exist a winning strategy for $\mathcal{GCPN}$ where we use M as the initial marking (rather than M_0). Furthermore this invariant holds irregardless of which element the operation *pop* picks in line 6. The first property allows us to implement early termination by adding the additional constraint $Win[M_0] \neq \mathtt{tt}$ to the while loop in line 5, and the second property allows us to do more intelligent search for winning strategies. The algorithm has been implemented in the model checker included in the BRITNeY Suite [28, 29]. The

[2] Rather than re-evaluating both the conjunction and the disjunction each time we reach this line, we notice that the entire expression is true iff just one state reachable by a controllable action is winning and if all states reachable by uncontrollable actions are winning. Using an integer to keep track of how many states reachable by an uncontrollable action are currently not marked as winning and a boolean to keep track of whether a state reachable by a controllable action has been marked as winning, we only do a constant amount of work each time we need to re-calculation the expression.

Algorithm 1 SolveGCPN

Require: $\mathcal{GCPN} = (P, T_c, T_u, D, Type, Pre, Post, M_0, W)$, a game CP-net

1: $\mathsf{Storage} \leftarrow \{M_0\}$
2: $\mathsf{Waiting} \leftarrow \{(M_0, (t, m), M') \mid M_0 \overset{(t,m)}{\rightarrow} M'\}$
3: $\mathsf{Win}[M_0] \leftarrow W(M_0)$
4: $\mathsf{Depend}[M_0] \leftarrow \emptyset$
5: **while** $\mathsf{Waiting} \neq \emptyset$ **do**
6: $\quad (M, (t, m), M') \leftarrow pop(\mathsf{Waiting})$
7: $\quad$ **if** $M' \notin \mathsf{Storage}$ **then**
8: $\quad\quad \mathsf{Storage} \leftarrow \mathsf{Storage} \cup \{M'\}$
9: $\quad\quad \mathsf{Depend}[M'] \leftarrow \{(M, (t, m), M')\}$
10: $\quad\quad \mathsf{Win}[M'] \leftarrow W(M')$
11: $\quad\quad \mathsf{Waiting} \leftarrow \mathsf{Waiting} \cup \{(M', (t', m'), M'') \mid M' \overset{(t',m')}{\rightarrow} M''\}$
12: $\quad\quad$ **if** $\mathsf{Win}[M']$ **then**
13: $\quad\quad\quad \mathsf{Waiting} \leftarrow \mathsf{Waiting} \cup \{(M, (t, m), M')\}$
14: $\quad\quad$ **end if**
15: $\quad$ **else** {reevaluate}
16: $\quad\quad \mathsf{Win}^* \leftarrow \bigwedge_{(t_u,m_u)\in TRANS_u,\, M \overset{(t_u,m_u)}{\rightarrow} M'} \mathsf{Win}[M'] \wedge \bigvee_{(t_c,m_c)\in TRANS_c,\, M \overset{(t_c,m_c)}{\rightarrow} M''} \mathsf{Win}[M'']$
17: $\quad\quad$ **if** Win^* **then**
18: $\quad\quad\quad \mathsf{Waiting} \leftarrow \mathsf{Waiting} \cup \mathsf{Depend}[M]$
19: $\quad\quad\quad \mathsf{Win}[M] \leftarrow \mathtt{tt}$
20: $\quad\quad$ **end if**
21: $\quad\quad$ **if** $\mathsf{Win}[M']$ **then**
22: $\quad\quad\quad \mathsf{Depend}[M'] \leftarrow \mathsf{Depend}[M'] \cup \{(M, (t, m), M')\}$
23: $\quad\quad$ **end if**
24: $\quad$ **end if**
25: **end while**

current implementation uses either a simple depth-first search or a breath-first search. The depth-first search has a tendency to find winning strategies faster than the breath-first search, as it favors making controllable moves rather than waiting for the opponent to make "good" uncontrollable moves. More intelligent search can be made by making e.g. an $\alpha - \beta$ search [25] as known from the AI world to allow computers to play difficult games, such as chess.

Due to the general result of [19], we can replace the calculation in line 16 of the algorithm with any monotone property (any propositional formula not using negation or implication). For example, we may find that it is more intuitive that a state is winning if we have a controllable move leading to a winning state *or* all uncontrollable moves lead to winning states.

We also notice that we never use that $\mathbb{N}^{PLACE}$ (the syntactically possible states of $\mathcal{GCPN}$) is finite, but only that the reachable states of $\mathcal{GCPN}$ are, so we can in fact remove the requirement that all types in D are finite as long as we only use a finite subset of the values. This allows us to analyze arbitrary game CPN models using the algorithm. If the algorithm terminates, we will know that

the result is correct (due to the loop invariant, which is proven correct in [5]).

4.1 Experimental Results

We cannot directly analyze the network protocol from the previous example, as the place A is unbounded (and the algorithm will keep executing the Send Packet transition, which remains enabled). However, if we put a bound on the number of tokens on the places A, B, C, and D, we can analyze the model. We will then discover that there is no winning strategy (the network can just keep on throwing away packets). If we furthermore limit the number of times the network can drop packets, we will find that we have a winning strategy. When we add a limit on the number of packets on the network and limit the number of times a packet can be lost, we obtain the model in Fig. 2. The only changes from the original net in Fig. 1 is that we have added two new places Capacity (bounding the number of tokens on A, B, C, and D) and May lose (bounding the number of times Drop Packet and Drop Ack can fire). Also the place NextRec has been moved to improve the layout.

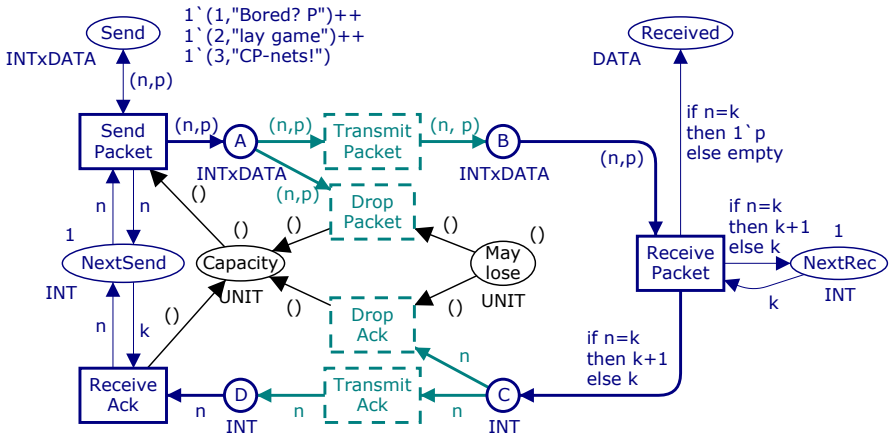

Fig. 2. The stop-and-wait protocol modified for analysis.

We have conducted four series of experiments: two for the model in Fig. 2 and two for the same model but with the May lose place and all connected arcs removed (i.e. removing the limit on the number of packet losses). For each of the two models, we have conducted two series of experiments, one without early termination and one with early termination, where we allow the algorithm to terminate as soon as we have a proof either for or against the existence of a winning strategy. Each series consists of a number of experiments with varying capacity and bound of the number of packets we may lose. We regard as winning states the states where all packets has been received successfully. All series are

conducted using a recursive depth-first traversal and using double hashing [8] with 15 combinations and 10^8 buckets for the Win and Storage data-structures (resulting in a total memory use of about 25 MiB). Depend is represented implicitly on the recursion stack. The experiments are not meant to show the general performance of the algorithm, as this has already been done in [5], but they are meant to give an impression of the performance of the algorithm when adapted to game CP-nets. All experiments are conducted on an Apple MacBook Pro with a 2.16 GHz Intel Core Duo processor[3] and 2 GiB RAM, using BRITNeY Suite version 0.9.75.101.

In Table 1, we see the results from analysis of the model with no limit on the number of losses of packets. The table has 6 columns. The first column, Early termination allowed, indicates whether early termination is allowed or not. The second column, Tokens on Capacity place, indicates how many tokens the place Capacity contains initially. The columns Storage and Win indicates how many states are stored in the corresponding data-structures. Note that we do not store states with no successors in Storage (as they are winning iff they are contained in Win), which is why the number of winning states may be larger than the number of stored states. The next column, Winning strategy exists indicates whether the algorithm concludes that a winning strategy exists. The final column, Execution time, indicates how long time (in seconds) the algorithm used to reach that conclusion.

The first thing we notice is that early termination really speeds the calculation up. It seems that the time spent increases exponentially without and quadratically with early termination. This is a lucky coincidence in this model, as the algorithm can easily see that no matter how many packets we pump onto the network, the network can just drop them and leave us at the initial state, thereby preventing us from even getting started with the protocol. We also note that it seems like the algorithm has a start-up cost of about 0.1 second, which is probably used for allocating the bit-arrays used for storing states.

In Table 2 we see the results for the analysis with a limit on the number of packets the network is allowed to lose. The columns have the same meaning as in Table 1, except we have added a column, Tokens on May lose place, which contains the number of tokens initially on the May lose place.

We notice that the algorithm now states that we do indeed have a winning strategy (and gives us a function which returns for each reachable state which event we should execute in order to win). Furthermore, we notice that when we do not allow early termination, the number of states and the execution time grows approximately linearly in the number of allowed losses (which makes sense, as we basically have a copy of the entire reachability graph for each possible marking of May lose). We also note that the number of states and the execution time when no packets can be lost correspond approximately to the number of states when we allow an arbitrary number of packet losses (this also makes sense, as a packet loss basically goes back an event).

[3] The processor contains two cores, but the analysis is currently only able to use one of them.

Table 1. Data from analysis of the network model in Fig. 2 without limit on number of losses (i.e. the May lose place and its surrounding arcs have been removed).

Early termination allowed	Tokens on Capacity place	Storage	Win	Winning strategy exists	Execution time (seconds)
No	1	20	2	ff	0.153
	2	115	32	ff	0.171
	4	1807	1032	ff	0.588
	6	14917	11508	ff	3.208
	8	83173	76208	ff	22.046
	10	355842	365770	ff	115.459
Yes	1	3	0	ff	0.111
	2	4	0	ff	0.116
	10	12	0	ff	0.145
	100	102	0	ff	0.282
	500	502	0	ff	3.392
	1000	1002	0	ff	12.338
	2000	2002	0	ff	55.809
	2800	2802	0	ff	123.180

When we allow packet loss, we see, again, that the number of states and the execution time grows linearly with the number of packets the network can lose, and they seem to grow quadratically in the capacity of the network. The calculation is significantly more difficult than before, when there was no limit on the number of packets we can lose, as we will have to search deeper for a winning strategy than we did to find a counterexample previously. We also note a significant speedup compared to not allowing early termination.

When we have found a winning strategy (or concluded that no winning strategy exists), we would like to show this strategy to the user. Ways to do this include printing all reachable markings and the winning move or to annotate a graphical representation of the reachability graph with the suggested move. This is very easy to implement, but not very good to convince users. A better way to present a winning strategy to a user is presented in the next section.

5 Automatically Generated Visualizations

While CP-nets are graphical and can be communicated to computer scientists, they contain a lot of details that are not needed to grasp the essentials of a model. Often we also need to communicate the ideas of the model to domain experts in order to validate that the constructed model actually represents the problem domain. In order to facilitate easy communication of (CPN) models, a lot of tools [10, 16, 24, 29] have been conceived with the purpose of constructing domain specific visualizations. These tools either mainly allows simple inspection

Table 2. Data from analysis of the network model in Fig. 2.

Early termination allowed	Tokens on Capacity place	Tokens on May lose place	Storage	Win	Winning Winning exists	Execution Time (seconds)
No	4	0	1732	2248	tt	0.340
	4	1	3538	4570	tt	0.602
	4	5	10762	13858	tt	1.679
	4	50	92032	118348	tt	18.752
	4	200	362930	466638	tt	140.891
	8	0	82768	120871	tt	16.561
	8	1	165939	242146	tt	39.675
	8	5	498612	727198	tt	131.478
Yes	4	200	10903	11688	tt	1.847
	8	5	290	297	tt	0.147
	10	0	50	51	tt	0.144
	10	1	185	192	tt	0.147
	10	10	737	757	tt	0.232
	10	100	12134	13043	tt	1.594
	10	1000	126699	136647	tt	71.228
	10	2000	253778	273733	tt	260.406
	100	0	410	411	tt	0.335
	100	1	825	827	tt	0.561
	100	10	4608	4625	tt	1.638
	100	100	43630	43946	tt	14.112
	100	500	438319	467416	tt	247.340
	500	73	149511	149636	tt	246.562
	1000	17	72265	72283	tt	247.027
	1500	3	24055	24059	tt	294.097

of the state of the model during execution, or requires that the modeler spends a lot of time constructing an visualization and tying it to the model.

Using the concept of games, we can do better. Firstly, we notice that visualizations often allow the user to experiment with the modeled system and observe how the system reacts to different stimuli, thereby playing the role of the environment. Another kind of visualization allows the user to play the role of the modeled system, contending against the environment. Both of these correspond nicely to how we play a game. We make a move and observe how the opponent reacts. So, basically, what we want to is to allow an external visualization component to control one side of the game and let the tool control the other side. In our example, we can let the user control the controllable actions, thereby taking the role of the protocol, or let the user play the role of the environment, thereby letting the user try to make the protocol fail. As the distinction between the modeled system and the environment is part of the model in game CP-nets, it is possible to automatically execute transitions from the environment when the user plays the role of the modeled system (and vice versa).

The rest of this section starts by introducing two simple kinds of automatically generated visualization, then goes on to discuss fairness during the visualization, and finally extends the scope to more general visualizations. All of the visualizations presented in this section have been implemented in the BRITNeY Suite [28, 29].

5.1 Binding Index Inspired Visualization

CPN Tools allows users to select specific modes for each transition by using binding indexes (in CPN Tools the notion of a high-level Petri net Graph [13, Sects. 7–9] is used, where binding elements, a transition and an assignment of values to variables on the surrounding arcs, correspond to transition modes). In Fig. 3 we see an example of a binding index in CPN Tools. Here the transition Transmit Packet is enabled, and the variable n can be assigned either the value 1 or 2, and the variable p can be assigned either "Bored? P" or "lay game". Binding indexes make it possible to exercise very detailed control of the model, but also requires the user to browse through the entire model to find enabled transitions and to select the desired mode.

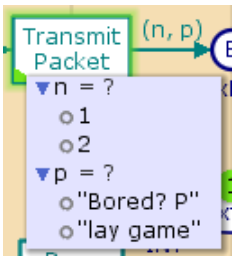

Fig. 3. A binding index from CPN Tools.

One way to automatically generate a "visualization" is to simply show the user a list of all enabled binding elements (i.e. hide any binding elements corresponding to uncontrollable transitions and execute them automatically). An example of this can be seen in Fig. 4. Here we see that the transitions Receive Packet and Send Packet are enabled. We also see that Receive Ack is disabled (normally the list of disabled transitions are hidden, but they can be made visible by the user by clicking on the double arrow). We have currently selected the Receive Packet transition, and we see that we can select one of two binding elements. We note that we do not see any of the transitions for transmitting and dropping packets, as they are part of the environment and cannot be controlled by the user. At the top, we can see that the last move of the tool was to execute the transition Drop Packet and the values assigned to its surrounding variables. This visualization is very straightforward to implement and use, but requires that the user has a very detailed knowledge of the model (or that the model is very simple).

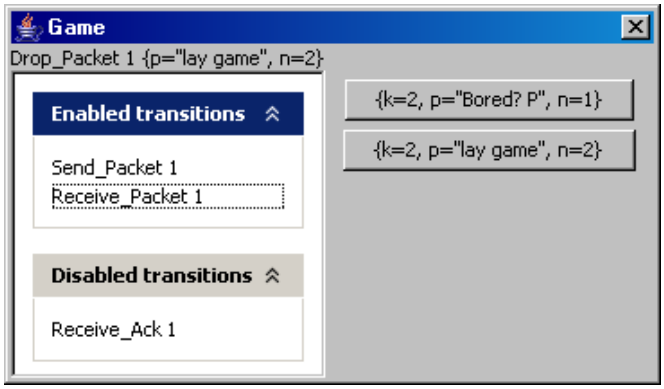

Fig. 4. A visualization generated by showing enabled binding elements.

A better way to automatically generate visualizations is to take the idea of the binding index a bit further. We will show a dialog for each enabled transition. The dialog can be placed by the user, allowing him to arrange the constructed work space at will, but will be opened/closed by the system depending on whether the transition is enabled. Instead of just listing the possible assignments, the dialog contains well-known widgets for displaying booleans, integers, and strings. More complex types can currently be selected by selecting the values from a list. An example of this can be seen in Fig. 5. Here three transitions are enabled, Send Packet, Receive Packet, and Receive Ack. We can see that we are currently executing Send Packet as it is grayed out and a rotating progress indicator indicates that the transition is being executed. In the Receive Packet window we see that K is assigned the value 3, N to 2, and P to "Rocks". The small triangle with an exclamation mark near the input field of P indicates that the value selected is indeed valid, but it is inconsistent with the other values (the packet containing "Rocks" has a sequence number of 3). We cannot change the value of K—it is shown for informational purposes only (this feature can be used to display a message to the user, e.g. by adding a variable message and assigning it one of the values "Please enter your name" or "The packet has been transmitted", depending on what action is required of the user and the model). In the window for Receive Ack we see that K is assigned to 2 and N to 0. The circle with a cross indicates that the value is not valid, and another value should be chosen.

Apart from being a natural way to select between available tokens, this visualization also allows the user to generate new tokens by adding a controllable source transition, which has free variables. This is also how we would normally generate new tokens in CPN Tools, but CPN Tools only allows users to bind variables freely from "small colour sets", which basically means booleans and enumerations. BRITNeY Suite extends this to also include strings and integers, which e.g. makes it possible to ask a user for name and age and use that later in the model.

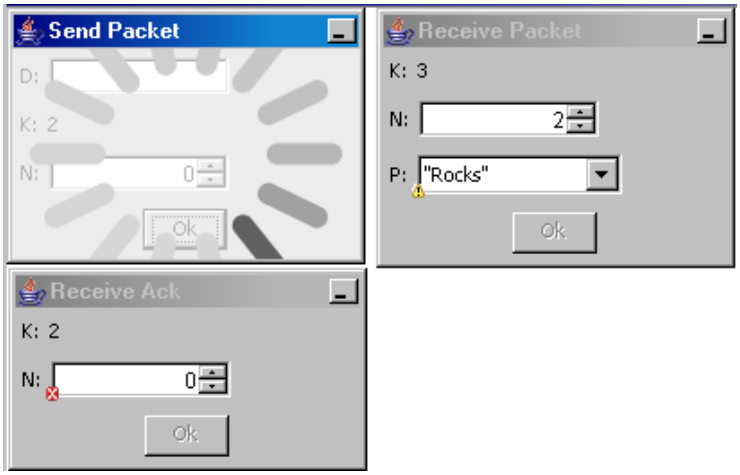

Fig. 5. Visualization generated from the net using common widgets.

This visualization can also be used without using the notions of games, as we only use the separation of transitions to make some choices on behalf of the user. In the next section we shall see an example of a visualization which crucially depends on the separation in order to work by using controllable actions to provide stimuli to the model and uncontrollable actions to provide feedback to the user.

5.2 Visualization using SceneBeans

BRITNeY Suite integrates the SceneBeans [26] library for use with visualizations. The library uses an XML specification for describing visualizations and allows the model to interact with the specified visualization by invoking commands in the visualization and receiving events from the visualization.

Normally the programmer would use the interface from Listing 1 and call the functions from declarations or from code segments attached to the transitions of the model. The first two functions in Listing 1 are used to setup or reset the visualization, invokeCommand (l. 5) is used to invoke commands in the visualization, which basically means to show some animation or feedback to the user. The next four functions, in lines 7–10, makes it possible to listen for events from the visualization (e.g. the user has clicked a certain object, dragged a certain object, or some other action). The last two function makes it possible to set certain values in the visualization, such as the position or color of an element. The use of this interface has a tendency to clutter the model and special precautions must be taken in order to turn the visualization on or off, e.g. for analysis.

Rather than letting the user tie the model and the visualization together manually, we shall identify events of the SceneBeans visualization with controllable actions and commands with uncontrollable actions. Thus, whenever an event is dispatched, the system will try to execute the corresponding transition

Listing 1 The interface to the SceneBeans library.

```
signature SCENEBEANS = sig
  val setAnimation: unit -> string
  val reset: unit -> unit

  val invokeCommand: string -> unit

  val hasMoreEvents: unit -> bool
  val getNextEvent: unit -> string
  val peekNextEvent: unit -> string
  val waitForEvent: string -> unit

  val setValue: string * string * string -> unit
  val getValue: string * string -> string
end
```

as soon as possible. If an uncontrollable transition is fired, the corresponding command (if one exists) will be called. We shall identify events and commands with transitions simply by name. A more complex mapping can be made (e.g. by introducing a "synchronize" predicate, which determines whether a transition should be synchronized with a command or event), but we prefer this simple approach, as it is simpler to understand for the user, and eliminates the need to manually specify such a predicate.

Suppose we have created an visualization showing two computers and a network. The visualization has commands for moving dots from one computer to the other and vice versa as well as for dropping these dots from the cloud. If we name these commands after the transitions, we can, without altering the net in Fig. 1 at all, generate a SceneBeans visualization like the one in Fig. 6. Here we have a sender (on the left) and a receiver (on the right) and a network connecting them. The dots represent data or acknowledgment packets sent over the network. When a transition fires in the model, the visualization is automatically updated. When we click on the sender, an event is generated, which causes the sender to send a packet, so initially the model does nothing (it waits for packets to appear on the network). We can send as many packets as we want as fast as we want. The model will transmit or drop the packets automatically. We could also have added a way to manually receive packets, but we have chosen that the model can do this itself.

We note that we only have concepts corresponding to the command and event parts of the SceneBeans interface. It would be very nice to extend the automatic synchronization of SceneBeans visualizations and game CPN models to also include reading from and writing to properties of the visualization, for example to show the NextSend and NextRec counters directly in the visualization, or to attach package numbers to the dots. This could probably be done by associating values in the visualization with tokens on global fusion places in the game CPN model. How to do this is future work.

Fig. 6. SceneBeans visualization automatically synchronized with the model.

5.3 Intelligent Playing

Using the algorithm outlined in Sect. 4, we can allow the computer player to play more intelligently by using the winning strategy (if one exists) as guidance rather than random selection.

In principle, this requires that we calculate the entire winning strategy beforehand. As explained in Sect. 4, we know that we can use $\alpha - \beta$ heuristics to speed up the search, and we can use early termination of the analysis algorithm to speed up the initial start up time. If a winning strategy is found during a previous game, it can of course be reused during future visualizations.

Allowing the computer player to use a winning strategy is a nice way to present a winning strategy to the user, as it may be difficult to grasp the winning strategy as a mapping from states to transition modes. By allowing the user to play against the strategy, he can easily convince himself that the strategy proposed by the algorithm is indeed winning. All visualizations introduced in this section can be used to demonstrate that a winning strategy is winning.

5.4 Fairness in Execution

When we make visualizations using games, we may need to impose some fairness during the execution, as the model can be executed fast enough to make it difficult for the user to interact with the model. This can be done in different ways.

The simplest way to impose some kind of fairness is to make the game strictly turn-based: the model makes one step, followed by the user, repeat until no more transitions are enabled. If a transition is not enabled for the player, who is to make the next move, the turn is passed on to the other player. This has the advantage of being simple, easy to understand, and it makes implementing simple games, such as tic-tac-toe very easy. The disadvantage is that depending on the modeling detail level, one player may gain an unfair advantage.

Another way to impose fairness is to allow the model to execute some transitions at will, but force it to synchronize on others. This is in particular useful for SceneBeans visualizations, where the model is often expected to be simulated while the user is observing, but we want interaction to happen immediately when the user requests it. In the network example this corresponds to how the network is able to transmit or drop packets at will, with the user observing the transmis-

sions, but when the user clicks on the sender, the model will immediately send a new packet onto the network.

A third way to impose fairness is to make the execution of transitions take time. The most obvious way to use this is to identify model time with real time and use a coloured variant of Time Petri nets [20]. In this case transitions may only be enabled for a certain period of time or only a certain period of time after another action.

5.5 More General Visualizations

Even though one goal of this work is to automatically generate visualizations, we have also made available a very generic interface, which makes it possible for developers to build visualizations or other programs interfacing with game CPN models using a high-level interface.

The interface is written in Java and gives developers the ability to write their own programs interfacing with the model. The interface, which can be seen in Listing 2, informs a consumer, i.e. any class implementing the interface, whenever a transition becomes enabled or disabled along with the possible bindings of the variables of the transition (ll. 2–3). In addition the consumer is informed whenever the computer makes a move (the uncontrollable moves, l. 8) and allows the consumer to specify a controllable move (l. 5). Furthermore, to make it easy to write programs, which only listens to what happens (and are not interested in interfering with the execution), consumers are also informed when a controllable move is made (l. 7) and when the game has ended (l. 10).

Apart for all of this transition-based information, the consumer has access to the markings of all places in the model as well as type information about all places and variables in the net.

Listing 2 The GameListener interface.

```
1  public interface GameListener {
2    public void transitionEnabled(Instance<Transition> ti, Bindings bindings);
3    public void transitionDisabled(Instance<Transition> ti);
4
5    public Pair<Instance<Transition>, Binding> controllableMove();
6
7    public void controllableMove(Instance<Transition> ti, Binding binding);
8    public void uncontrollableMove(Instance<Transition> ti, Binding binding);
9
10   public void gameOver();
11 }
```

All of the visualizations presented in this section have been implemented using this interface, so even though the interface is simple, it is still very versatile, and has, in addition, been used to automatically generate message sequence

charts from game CPN models as well as basis for implementing a workflow system on top of game CP-nets.

6 Conclusion

In this paper, we have defined the notion of game coloured Petri nets. We have argued that they provide a nice way to separate the model of the system from the model of the environment. We have introduced a way of finding winning strategies for game CP-nets by means of reachability analysis and introduced an intriguing application of game CP-nets to visualization of CP-net models.

The distinction between controllable and uncontrollable actions for Petri nets has also been done in [7], but only workflow nets are considered and the paper requires that uncontrollable transitions can only exist in free choices consisting of uncontrollable transitions only. This makes it impossible for uncontrollable actions to depend of different conditions and for controllable actions to co-exist with uncontrollable transitions. Game coloured Petri nets are more widely applicable, as they extend the idea to coloured Petri nets and impose no restrictions on the modeling style, yet still offer powerful analysis techniques.

Future work includes extending the notion of partial observability [3] to game CP-nets. One way to do this, would be to only allow the player to observe certain places. This can be done by only allowing the player to observe global fusion places.

We could also use the notion of symmetry [6] as another way to specify the different players. In our implementation controllability/uncontrollability is tied to transitions, which makes it difficult to implement systems with a number of peers, say a peer to peer network with 3 nodes, where one of the peers is controllable and the other are not. With our implementation, we would need to model "peer 1" separately from "peer 2" and "peer 3", even though they can do the same actions. It would be nice to be able to specify in the declaration of the type representing the network nodes that actions involving "peer 1" are controllable whereas actions involving the other peers are not. This seems trivial to do, but we have to overcome the problem of what happens when one action involves both controllable and uncontrollable parts (say "peer 1" sends data to "peer 2"). It would be natural to look into how symmetries are specified in [9] as annotations on the types.

Another problem we saw during the analysis of the protocol in Sect. 4 was that we did not have a winning strategy if we allowed the network to drop as many packets as it wants to. We saw that for all examples with a bound on the number of packets which could be lost, we actually had a winning strategy. However, using the current method, we do not have a means to prove that we have a winning strategy which either leads us to the desired goal state or which drops infinitely many packets. This corresponds to the difference in checking simple reachability properties (such as safety properties, like deadlock-freeness) versus checking more complex properties (e.g. demand-response) using modal logics such as linear temporal logic (LTL) [23, 27] or computation tree logic

(CTL) [1]. In [2,22] the notion of extended goals for games is introduced. It would be nice to come up with an algorithm for deciding extended goals, formulated using either LTL or CTL.

Acknowledgments The author wishes to thank the members of the CPN Group at the University of Aarhus, in particular Kristian Bisgaard Lassen, for discussions on some of the topics described in this paper.

References

1. M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
2. P. Bertoli, A. Cimatti, M. Pistore, and P. Traverso. A Framework for Planning with Extended Goals under Partial Observability. In *Proc. of ICAPS 2003*, pages 215–225, 2003.
3. P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso. Planning in Nondeterministic Domains under Partial Observability via Symbolic Model Checking. In *Proc. of IJACI 2001*. AAAI Press, 2001.
4. C. Bossen and J.B. Jørgensen. Context-descriptive prototypes and their application to medicine administration. In *DIS '04: Proc. of the 2004 conference on Designing interactive systems*, pages 297–306, Boston, MA, USA, 2004. ACM Press.
5. F. Cassez, A. David, F. Emmanuel, K.G. Larsen, and D. Lime. Efficient On-the-fly Algorithms for the Analysis of Timed Games. In *Proc. of CONCUR 2005*, volume 3653 of *LNCS*, pages 66–80. Springer-Verlag, 2005.
6. E.M. Clarke, T. Filkorn, and S. Jha. Exploiting Symmetries in Temporal Logic Model Checking. In *Proc. of CAV'93*, LNCS, pages 450–462. Springer-Verlag, 1993.
7. J. Dehnert. Non-controllable Choice Robustness Expressing the Controllability of Workflow Processes. In *Proc. of ICATPN 2002*, volume 2360 of *LNCS*, pages 121–141. Springer-Verlag, 2002.
8. P.C. Dillinger and P. Manolios. Fast and accurate Bitstate Verification for SPIN. In *Proc. of SPIN 2004*, volume 2989 of *LNCS*. Springer-Verlag, 2004.
9. L. Elgaard. *The Symmetry Method for Coloured Petri Nets*. PhD thesis, Department of Computer Science, University of Aarhus, 2002. Also available as DAIMI PB-564.
10. G. Gallasch and L.M. Kristensen. A Communication Infrastructure for External Communication with Design/CPN. In *Proc. of 3rd Workshop on Practical Use of Coloured Petri Nets and the CPN Tools*, volume PB-554 of *DAIMI*, pages 79–93. Department of Computer Science, University of Aarhus, 2001.
11. S. Gordon, L.M. Kristensen, and J. Billington. Verification of a Revised WAP Wireless Transaction Protocol. In *Proc. of ICATPN'02*, volume 2360 of *LNCS*, pages 182–202. Springer-Verlag, 2002.
12. B. Han and J. Billington. Formalising the TCP Symmetrical Connection Management Service. In *Proc. of Design, Analysis, and Simulation of Distributed Syste ms*, pages 178–184. SCS, 2003.
13. Software and system engineering – High-level Petri nets – Part 1: Concepts, definitions and graphical notation. ISO/IEC 15909-1:2004, 2004.
14. K. Jensen. *Coloured Petri Nets—Basic Concepts, Analysis Methods and Practical Use. Volume 1: Basic Concepts*. Springer-Verlag, 1992.

15. J.B. Jørgensen and K.B. Lassen. Aligning Work Processes and the Adviser Portal Bank System. In *REBNITA05*, 2005.
16. E. Kindler and C. Páles. 3D-Visualization of Petri Net Models: Concept and Realization. In *Proc. of ICATPN 2004*, volume 3099 of *LNCS*, pages 464–473. Springer-Verlag, 2003.
17. L.M. Kristensen and K. Jensen. Specification and Validation of an Edge Router Discovery Protocol for Mobile Ad-hoc Networks. In *Integration of Software Specification Techniques for Application in Engineering*, volume 3147 of *LNCS*, pages 248–269. Springer-Verlag, 2004.
18. L.M. Kristensen, M. Westergaard, and P.C. Nørgaard. Model-based Prototyping of an Interoperability Protocol for Mobile Ad-hoc Networks. In *Proc. of Fifth International Conference on Integrated Formal Methods*, volume 3771 of *LNCS*, pages 266–286. Springer-Verlag, 2005.
19. X. Liu and S.A. Smolka. Simple Linear-Time Algorithms for Minimal Fixed Points (Extended Abstract). In *Proc. of ICALP 1998*, volume 1443 of *LNCS*, pages 53–64. Springer-Verlag, 1998.
20. P. Merlin and D. J. Farber. Recoverability of communication protocols - implication of a theoretical study. *IEEE Trans. on Communications*, 24(2):1036–1043, 1976.
21. C. Ouyang and J. Billington. On Verifying the Internet Open Trading Protocol. In *Proc. of 4th International Conference on Electronic Commerce and Web Technologies*, volume 2738 of *LNCS*, pages 292–302. Springer-Verlag, 2003.
22. M. Pistore and P. Traverso. Planning as Model Checking for Extended Goals in Non-Deterministic Domains. In *Proc. of IJCAI 2001*, pages 479–486. AAAI Press, 2001.
23. A. Pnueli. The temporal logic of programs. In *Proc. of 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.
24. J.L. Rasmussen and M. Singh. *Mimic/CPN. A Graphical Simulation Utility for Design/CPN. User's Manual*. `www.daimi.au.dk/designCPN`.
25. Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2002.
26. SceneBeans. Online `www-dse.doc.ic.ac.uk/Software/SceneBeans`.
27. M. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *In proc. of IEEE Symposium on Logic in Computer Science*, pages 322–331, 1986.
28. M. Westergaard. BRITNeY Suite website. `wiki.daimi.au.dk/britney/`.
29. M. Westergaard and K.B. Lassen. The BRITNeY Suite Animation Tool. In *Proc. of ICATPN 2006*, volume 4024 of *LNCS*, pages 431–440. Springer-Verlag, 2006.
30. P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, 1997.