Model Checking Coloured Petri Nets Exploiting Strongly Connected Components

Allan Cheng, Søren Christensen, Kjeld Høyer Mortensen
{acheng,schristensen,khm}@daimi.aau.dk

Computer Science Department, Aarhus University
Ny Munkegade, Building 530–540
DK–8000 Aarhus C, Denmark

Abstract

In this paper we present a CTL-like logic which is interpreted over the state spaces of Coloured Petri Nets. The logic has been designed to express properties of both state and transition information. This is possible because the state spaces are labelled transition systems. We compare the expressiveness of our logic with CTL's. Then, we present a model checking algorithm which for efficiency reasons utilises strongly connected components and formula reduction rules. We present empirical results for non-trivial examples and compare the performance of our algorithm with that of Clarke, Emerson, and Sistla.

1 Introduction

Coloured Petri Nets (CP-nets or CPN) are convenient for specifying complex concurrent systems. Until now properties of CP-nets have mainly been specified directly in terms of the state spaces of CP-nets [4, 6]. Temporal logics such as CTL are also useful for expressing properties of concurrent systems (see, e.g., [1]). We show how we can define a CTL like logic, ASK-CTL, tailored especially for expressing properties of state spaces of CP-nets. We provide example formulas which indicate that the logic is powerful enough to express many of the standard CP-net properties. Use of a logic implies that we get a well understood and easy to use framework for expressing a much wider range of properties.

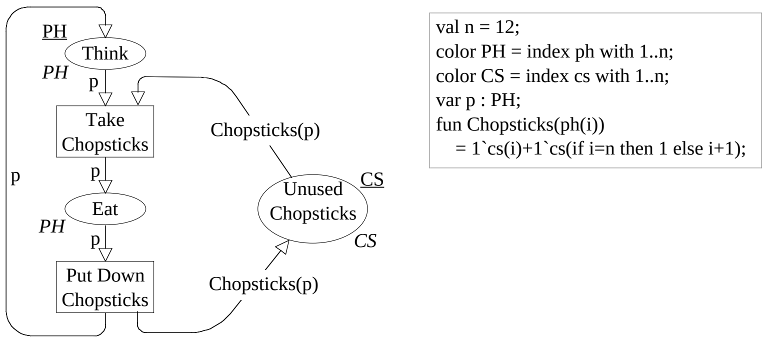
If a logic should be of practical use, it must be possible to verify formulas efficiently. The state space explosion problem often makes verification impractical. Solutions to this problem are mainly concerned with two methods: The first is state space reduction as proposed by, e.g., Valmari, Huber, Jensen, Godefroid, and Wolper [14, 3, 6, 2, 15]. The second method is concerned with algorithms which traverse the state space in a more efficient manner. The last point is addressed by this paper. We show how it is possible to improve the standard linear time model checking algorithm in that we, in some cases, avoid searching the complete state space, by taking into account strongly connected components (SCC's). Our algorithm has the same worst case complexity as the standard algorithm [1]. Nevertheless, our algorithm is faster in many interesting cases, depending on the topology of the SCC's and the combination sub-expressions in ASK-CTL formulas.

The rest of the paper is organised as follows. First we introduce Coloured Petri Nets and state spaces (section 2 and 3 respectively). Then we present our proposal for a suitable logic, called ASK-CTL, for CP-nets, and motivate its usefulness by expressing properties about example CP-nets (sections 4 and 5). The formal definition of ASK-CTL is then given in section 6 (including a definition of state spaces). In section 7 we show how to model check ASK-CTL formulas, taking advantage of strongly connected components. Performance measures of this model checking algorithm and comparisons with the standard algorithm [1] are given in section 8. Finally in section 9, the conclusion.

2 Introduction to Coloured Petri Nets

We give here a short and informal overview of CP-nets although we assume the reader to have some prior knowledge of CP-nets. For an in-depth introduction to CP-nets, see [4, 5]. We use three examples of CP-nets which are used for performance measures.

The first example introduces the notation used in this paper, and illustrates the classical scenario of the Chinese Dining Philosophers (figure 1). A number of philosophers share a bowl of rice, eating with chop-sticks. Exactly one chop-stick is located between each philosopher, i.e., two neighbours share a chop-stick. Each philosopher can be in a state where the philosopher is either eating or thinking, modelled by two places called Eat and Think respectively. In order to eat, the philosopher p needs to take two chop-sticks, $\mathit{Chopsticks}(p)$; one from the left and one from the right. Unused chop-sticks are located on the place called Unused Chopsticks. In order to resume thinking, the philosopher puts down both chop-sticks at the same time. In the initial state, all philosophers are located on Think indicated with the inscription PH, and all chop-sticks are located on Unused Chopsticks.



```
val n = 12;
color PH = index ph with 1..n;
color CS = index cs with 1..n;
var p : PH;
fun Chopsticks(ph(i))
   = 1`cs(i)+1`cs(if i=n then 1 else i+1);
```

Figure 1: The Dining Philosophers example.

The philosophers are modelled with the colour set (type) PH which is an indexed set (PH $= \{ph(1), \ldots, ph(n)\}$). The chop-sticks are likewise modelled with the indexed colour set CS. Finally, the function Chopsticks() returns a multi-set[1] of chop-sticks, given a philosopher. For example,

$$\texttt{Chopsticks}(\texttt{ph}(1)) = 1\text{`}\texttt{cs}(1) + 1\text{`}\texttt{cs}(2).$$

3 Introduction to State Spaces

We make extensive use of state spaces[2] so we give here an informal overview of some of the relevant concepts. We postpone the formal definition of state spaces until needed.

One approach to formal verification of a complex system is to generate all possible states that the system can reach, given an initial state as starting point. When also recording the transitions from state to state, we obtain a labelled transition system, which we refer to as the state space of the system. The transition system is a graph with the property that all nodes are reachable from the node representing the initial state (see figure 2). Given the full state

[1]A set where multiple occurrences of the same element is possible. Also called a bag.
[2]State spaces are elsewhere referred to as occurrence- or reachability graphs.

space we are now able to check properties that we expect the system to have. For example, we verify a safety/invariant property by traversing all states in the graph. We typically do this by quantifying over paths in the state space, where a *path* is a sequence of states and transitions, possibly infinite.

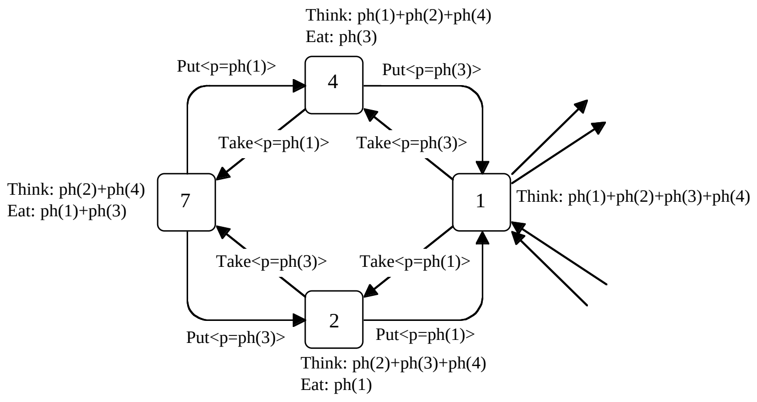


Figure 2: A part of the state space for the Dining Philosophers example with 4 philosophers.

An example of a partial state space can be seen in figure 2. Each node has a label indicating the marking, where the marking of `Unused Chopsticks` is left out, since it can be derived from `Think` and `Eat`. Each edge has a label indicating the *binding element* (occurring transition and values of variables).

4 The Logic ASK-CTL

The first contribution of this paper is the proposal of a CTL-like logic, ASK-CTL, useful for checking properties of CP-nets. The models over which we interpret ASK-CTL are state spaces of CP-nets. These graphs carry information on both nodes and edges. Hence, a natural extension of CTL is the ability to also express properties about the information labelling the edges. (E.g., edge information is needed when expressing liveness properties since liveness is expressed by means of transition information.) For this purpose we introduce two mutually recursively defined syntactic categories of formulas; state and transition formulas which are interpreted on the state space at states and transitions respectively.

As found in CTL and other temporal logics, quantified state formulas and transition formulas are interpreted over paths. Path quantification is used in combination with the CTL until-operator. This combination provides a means for expressing temporal properties.

In ASK-CTL we allow rather general predicates, since these are useful for verification of CP-nets. We argue that ASK-CTL is exactly as expressive as CTL in the case where we limit the basic predicates (α and β below) to atomic propositions. Since CTL cannot express standard fairness properties we inherit this inability, and lose the ability to express interesting properties such as impartiality, fairness, and justice (as defined in [4, 10]). However, there exist partial remedies for this drawback as shown by Clarke et al. [1]. In that paper the logic CTL^F is introduced as a slight extension of CTL. The purpose is to introduce fairness into CTL. This is done at the semantic level by interpreting CTL-formulas only over paths which are fair with respect to a set of "fairness" predicates. One observes that SCC-graphs are used in connection with CTL^F, but not for efficiency purposes as in our work. Fairness can be introduced in a similar fashion for ASK-CTL. We do not elaborate further on this subject as it is beyond the scope of this paper.

4.1 Syntax

Assume a fixed CP-net N. The logic, which is interpreted over the state space of N, has two categories of formulas: *state* and *transition* formulas. The two syntactical categories are mutually recursive.

State formulas: $\mathcal{A} ::= tt \mid \alpha \mid \neg\mathcal{A} \mid \mathcal{A}_1 \vee \mathcal{A}_2 \mid <\mathcal{B}> \mid EU(\mathcal{A}_1, \mathcal{A}_2) \mid AU(\mathcal{A}_1, \mathcal{A}_2)$

where tt is interpreted as the constant value true, α is a function from the set $[\mathcal{M} \rightarrow I\!B]$, i.e., a function mapping from markings to booleans, and $\mathcal{B}$ is a transition formula. *EU* and *AU* are explained below.

Transition formulas: $\mathcal{B} ::= tt \mid \beta \mid \neg\mathcal{B} \mid \mathcal{B}_1 \vee \mathcal{B}_2 \mid <\mathcal{A}> \mid EU(\mathcal{B}_1, \mathcal{B}_2) \mid AU(\mathcal{B}_1, \mathcal{B}_2)$

where β is a function from the set $[BE \rightarrow I\!B]$, i.e., a function mapping from binding elements to booleans, and $\mathcal{A}$ is a state formula.

We use the convention of always starting with $\mathcal{A}$, thus all ASK-CTL formulas are state formulas at the top-level, and transition formulas can only appear as sub-formulas. Furthermore, when model checking, we implicitly do this with respect to the initial state.

ASK-CTL resembles CTL except for the $< \cdots >$ operator. This operator provides the possibility of changing between state and transition formulas. In section 5.1, we give examples to demonstrate the usefulness of this operator.

Apart from the boolean operators $\neg$ and $\vee$ the above logic also contains the standard temporal operator U (until) combined with the path quantifiers E and A (exist and for-all respectively). E.g., the $EU(\mathcal{A}_1, \mathcal{A}_2)$ operator expresses the existence of a path from a given marking with the property that $\mathcal{A}_1$ holds until a marking is reached at which $\mathcal{A}_2$ holds. Dually, $AU(\mathcal{A}_1, \mathcal{A}_2)$ requires the property to hold along all paths from a given marking.

We have imposed no restrictions with respect to computability of the boolean functions α and β. We assume that they range over predicates which in practice are useful for verification purposes, i.e., they can be computed efficiently.

The syntax of ASK-CTL is minimal, which is an advantage when we define the formal semantics. In order to increase the readability of the formulas we make use of syntactic sugar. E.g., $Pos(\mathcal{A})$ means that it is possible to reach a state where $\mathcal{A}$ holds, $Inv(\mathcal{A})$ means that $\mathcal{A}$ holds in every reachable state, and $Ev(\mathcal{A})$ for all paths, $\mathcal{A}$ holds within a finite number of steps. Thus for the dining philosophers example CP-net we can easily formulate the question whether the initial marking is a home marking[3]. We only need to check if the state formula $Inv(Pos(IsInitial))$ is satisfied.

5 Example CP-nets

The second example is a CP-net taken from [8] where Kindler and Walter solves the problem of rearranging different integers asynchronously, see figure 3. Initially a number of different integers are distributed on the four places `small`, `great`, `compSM`, and `compGR`. The latter two places contain only one integer each. After a finite number of steps the system ends up in a dead state, i.e., a state where no binding elements are enabled, in which the following properties hold:

- The integers on the places `small` and `maxSM` are smaller than the integers on the places `great` and `minGR`.

[3] I.e., the initial state can be reached from every reachable state.

- The number of integers on place `small` and `great` is the same as in the initial state.
- The place `maxSM` contains exactly one integer, and this integer is larger than any of the integers on the place `small`.
- The place `minGR` contains exactly one integer, and this integer is smaller than any of the integers on the place `great`.

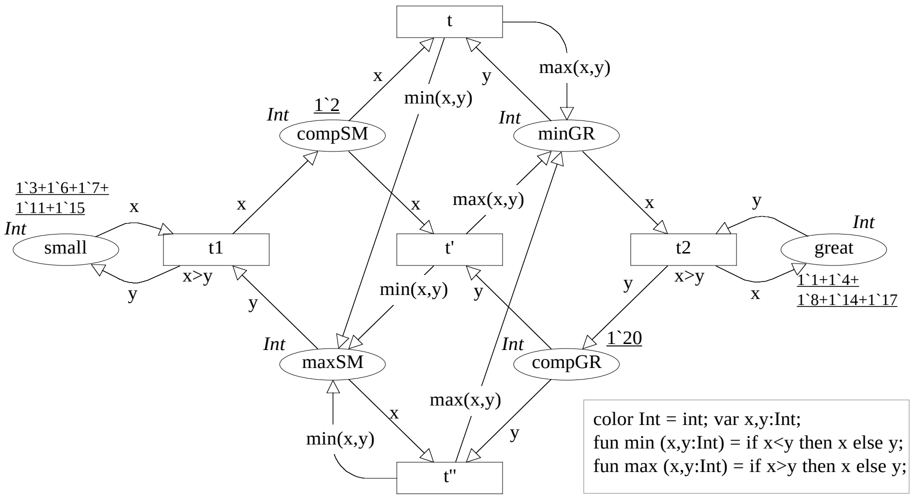


Figure 3: The Integer Rearrangement example.

As the third and final example, we present a CP-net we call the Multi Stage Process example (figure 4). The purpose of the example is to illustrate processes which go through multiple stages of success and failure. The idea is as follows: Processes perform some kind of testing and continue to do so as long as the test fails. If the test succeeds the process divides into two which continue in a new cycle where they change between modes of waiting and idle. Here processes perform tasks which also can fail or succeed. Furthermore, if all failed processes are simultaneously located on either `Wait1` or `Wait3`, then all processes may stop performing tasks and leave the cycle. When the processes leave this cycle, they become inactive and thus do not perform any further tasks.

The choice of these three examples are motivated in section 8.1, where we describe their state spaces.

5.1 Expressing CP-net Properties Using ASK-CTL

In this section we use ASK-CTL to express reachability, liveness, and home properties as presented in [6]. Then, we consider properties of the three examples.

We let M denote a marking, the state formula α_M denote the characteristic predicate for M, i.e., $\alpha_M(M')$ is true if and only if $M = M'$, and the transition formula β_t is the characteristic predicate for the transition t, i.e., $\beta_t(b)$ is true if and only if the transition in b is t. The formula $Inv(Pos(\alpha_M))$ then expresses that M is a home marking. Reachability of M is expressed by the formula $Pos(\alpha_M)$. M is dead if it satisfies $\neg <tt>$ and $Inv(Pos(<\beta_t>))$ expresses that t is live.

For the Integer Rearrangement example it is expected that the system will reach a state where it is totally sorted in a finite number of steps. This property can be expressed as $Inv(Ev(IsSorted))$, where *IsSorted* is the state predicate denoting that: all integers in `small`

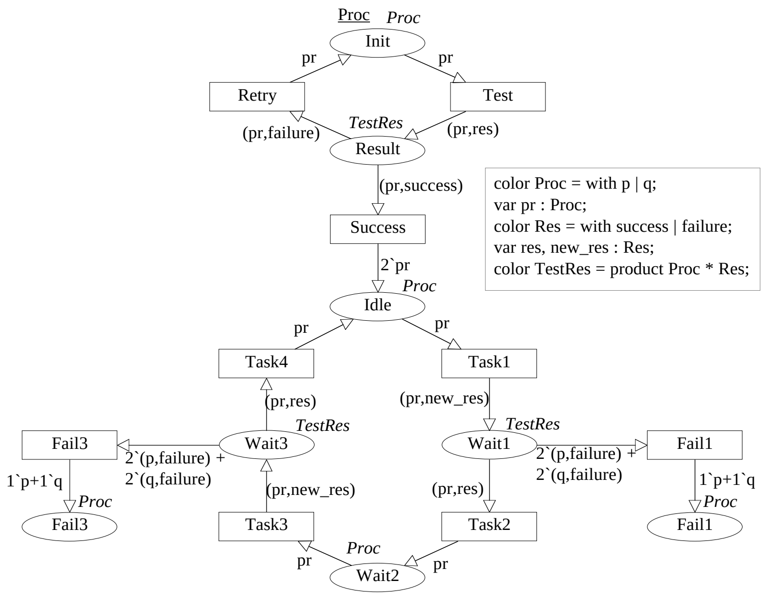


Figure 4: The Multi Stage Process example.

are less than the integer in `maxSM`, all integers in `great` are larger than the integers in `minGR`, and the integers in `maxSM` are smaller than the one in `minGR`.

For the Multi Stage Processes example we use the predicate *IsFailed* to express that both processes are in either `Fail1` or `Fail3`. We would expect that $Inv(Pos(IsFailed))$ is satisfied and $Inv(Ev(IsFailed))$ is not satisfied, i.e., the processes can always fail, but it is also possible that this never happens.

As another example of the usefulness of transition formulas, let us consider how one can express the property that one can reach a marking satisfying $\mathcal{A}$ by a sequence of steps involving only transitions from a set T. In the modal μ-calculus such a property would be expressed as $\mu X.\mathcal{A} \vee <T> X$, where the notation $<T>$ is borrowed from [12]. We notice that the formula uses the recursion operator μ. Without transition formulas we would not be able to express the above property easily. The state formula $\mathcal{A} \vee < EU(\beta_T, <\mathcal{A}>) >$ expresses the desired property, where β_T is the predicate that returns true if and only if the transition of a binding element is an element of T. For the Integer Rearrangement example the following formula is true: $IsSorted \vee < EU(\beta_{\{t1,t2,t'\}}, < IsSorted >) >$, i.e., either we have already reached a state where *IsSorted* holds or it is possible to reach such a state using only the transitions $t1$, $t2$, or t'.

6 Formal Definition of ASK-CTL

So far we have been informal about the meaning of ASK-CTL formulas. In the following we remedy this by giving the interpretation of ASK-CTL in terms of a formal semantics.

6.1 Definition of State Spaces

We use the concepts and notation of state spaces and SCC-graphs from [6].

Viewed as a definition of a directed graph, the definition of a state space is non-standard with respect to items 2 and 3 below. They are included for two reasons: Firstly, they allow multiple edges between two nodes. Secondly, they make later definitions simpler.

Definition 6.1 *The* **state space** *of a CP-net with initial marking M_0, is the directed graph $OG = (V, A, N)$ where:*

1. $V = [M_0\rangle$.
2. $A = \{(M_1, (t, b), M_2) \in V \times \mathrm{BE} \times V \mid M_1[t, b\rangle M_2\}$.
3. $\forall a = (M_1, (t, b), M_2) \in A : N(a) = (M_1, M_2)$.

Here V is the set of nodes (reachable markings), A the set of edges (occurrences of binding elements), and N a function relating edges to their end-point nodes. □

We always generate and explore the full state spaces when checking properties. Therefore we assume state spaces to be finite throughout this paper.

6.2 Interpretation

The logic is interpreted over state spaces, as defined above. For convenience we introduce the following notation: M denotes markings of the CP-net, b denotes binding elements, e denotes labelled edges of the corresponding state space, $\mathcal{A}$ denotes state formulas, and $\mathcal{B}$ denotes transition formulas.

First we define $M \models_{St} \mathcal{A}$, the interpretation of state formulas. The meaning of tt, α, $\neg$, and $\vee$ is standard and do not need further explanation:

- $M \models_{St} tt$ always holds
- $M \models_{St} \alpha$ iff $\alpha(M)$
- $M \models_{St} \neg\mathcal{A}$ iff not $M \models_{St} \mathcal{A}$
- $M \models_{St} \mathcal{A}_1 \vee \mathcal{A}_2$ iff $M \models_{St} \mathcal{A}_1$ or $M \models_{St} \mathcal{A}_2$

The next kind of state formula allows us to switch from state to transition formulas. Recall the motivation for introducing this operator, namely that it gives us the possibility to express properties about labels on edges in the state space. The operator, $<\mathcal{B}>$, means that we can find an immediate successor state from the current state and that $\mathcal{B}$ holds on the edge between the two states.

Before giving the formal definition of the $<\cdots>$operator, we need some convenient notation. We write $M \xrightarrow{b} M'$ whenever $(M, b, M') \in A$, i.e., (M, b, M') is an edge in the state space. Let $\mathcal{P}_M$ denote the set of paths starting in M, i.e., the set $\mathcal{P}_M = \{M_0 b_1 M_1 b_2 \cdots \mid M_0 = M \wedge M_0 \xrightarrow{b_1} M_1 \xrightarrow{b_2} M_2 \cdots\}$. Notice that a path, $\sigma \in \mathcal{P}_M$, may be either finite or infinite. We define the length of a path, $\sigma = M_0 b_1 M_1 \cdots M_{n-1} b_n M_n$, to be $|\sigma| = n$, otherwise infinite.

The formal definition of the $<\cdots>$ operator is as follows:

- $M \models_{St} <\mathcal{B}>$ iff $(\exists b.\, M \xrightarrow{b} M' \wedge (M, b, M') \models_{Tr} \mathcal{B})$

The last two state formulas to consider quantify over paths in combination with the until-operator, U, as known from CTL. A formula $U(F_1, F_2)$ is to be interpreted over a path. It holds for a path if there exists a state at which the (state) formula F_2 holds and F_1 holds at all preceding states along the path. In this logic, U only has meaning in combination with a path quantifier, existential (E) or universal (A). E.g., AU means that for every path, U holds (for the two given properties). The formal definition is as follows:

- $M \models_{St} EU(\mathcal{A}_1, \mathcal{A}_2)$ iff $(\exists \sigma \in \mathcal{P}_M.\, (\exists n \leq |\sigma|.(\forall 0 \leq i < n.\, M_i \models_{St} \mathcal{A}_1) \wedge M_n \models_{St} \mathcal{A}_2))$
- $M \models_{St} AU(\mathcal{A}_1, \mathcal{A}_2)$ iff $(\forall \sigma \in \mathcal{P}_M.\, (\exists n \leq |\sigma|.\, (\forall 0 \leq i < n.\, M_i \models_{St} \mathcal{A}_1) \wedge M_n \models_{St} \mathcal{A}_2))$

Notice that for the interpretation of $EU(\cdots,\cdots)$ and $AU(\cdots,\cdots)$, n is always a finite natural number, even if $|\sigma| = \infty$.

The interpretation of transition formulas, $a \models_{Tr} \mathcal{B}$, where $a = (M, b, M')$, is given in the following. Again, tt, β, $\neg$, and $\vee$ have a standard interpretation:

- $a \models_{Tr} tt$ always holds
- $a \models_{Tr} \beta$ iff $\beta(b)$
- $a \models_{Tr} \neg\mathcal{B}$ iff not $a \models_{Tr} \mathcal{B}$
- $a \models_{Tr} \mathcal{B}_1 \vee \mathcal{B}_2$ iff $a \models_{Tr} \mathcal{B}_1$ or $a \models_{Tr} \mathcal{B}_2$

Similarly, as for state formulas, we have a $<\cdots>$ operator in order to switch from transition to state formulas. I.e., if we currently consider a transition, the $<\cdots>$ operator allows us to express a property about the destination state of the transition. Note that the following formal definition is simpler than in the case of state formulas, because an edge always has a unique successor node.

- $a \models_{Tr} <\mathcal{A}>$ iff $M' \models_{St} \mathcal{A}$

The last two kinds of transition formulas, EU and AU, are defined in a dual fashion as in the case of state formulas:

- $a \models_{Tr} EU(\mathcal{B}_1, \mathcal{B}_2)$ iff $(\exists \sigma \in \mathcal{P}_M .$
  $(\exists n < |\sigma| . (\forall 0 \leq i < n . (M_i, b_{i+1}, M_{i+1}) \models_{Tr} \mathcal{B}_1) \wedge$
  $(M_n, b_{n+1}, M_{n+1}) \models_{Tr} \mathcal{B}_2))$
- $a \models_{Tr} AU(\mathcal{B}_1, \mathcal{B}_2)$ iff $(\forall \sigma \in \mathcal{P}_M .$
  $(\exists n < |\sigma| . (\forall 0 \leq i < n . (M_i, b_{i+1}, M_{i+1}) \models_{Tr} \mathcal{B}_1) \wedge$
  $(M_n, b_{n+1}, M_{n+1}) \models_{Tr} \mathcal{B}_2))$

Whenever we interpret a formula $\mathcal{A}$, we implicitly mean $M_0 \models_{St} \mathcal{A}$. For notational convenience we suggest to use the abbreviations (syntactic sugar):

$Pos\,\mathcal{A} \equiv EU(tt, \mathcal{A})$
: It is possible to reach a state where $\mathcal{A}$ holds.

$Inv\,\mathcal{A} \equiv \neg Pos \neg\mathcal{A}$
: $\mathcal{A}$ holds in every reachable state, i.e., $\mathcal{A}$ is invariant.

$Ev\,\mathcal{A} \equiv AU(tt, \mathcal{A})$
: For all paths, $\mathcal{A}$ holds within a finite number of steps, i.e., is eventually true.

$Along\,\mathcal{A} \equiv \neg Ev \neg\mathcal{A}$
: There exists a path which is either infinite or ends in a dead state, along which $\mathcal{A}$ holds in every state.

$<\mathcal{B}>\mathcal{A} \equiv <\mathcal{B} \wedge <\mathcal{A}>>$
: There exists an immediate successor state, M', in which $\mathcal{A}$ holds, and $\mathcal{B}$ holds on the transition between the current state and M'.

$EX(\mathcal{A}) \equiv <tt>\mathcal{A}$
: There exists an immediate successor state in which $\mathcal{A}$ holds.

$AX(\mathcal{A}) \equiv \neg EX(\neg\mathcal{A})$
: $\mathcal{A}$ holds for all immediate successor states, if any.

We use similar abbreviations for transition formulas. Notice how the formula $<\alpha>F$ from the Hennessy-Milner logic [11] can be captured using the state formula $<\beta_\alpha>\mathcal{A}_F$ where β_α is a predicate expressing that a binding element represents an α action and $\mathcal{A}_F$ is a state formula corresponding to F.

6.3 Expressiveness

It can be formally proven that the model checking problem of ASK-CTL reduces to the model checking problem of CTL. Here we just sketch the idea. Assume that we are given a state space and a state formula $\mathcal{A}$ (transition formula $\mathcal{B}$). In linear time the state space can be transformed into an R-structure [1]. Intuitively, the R-structure has a state for each marking M and each binding element e. We split a labelled edge into two unlabelled edges and an intermediate state which together with suitably defined atomic propositions represents the labelling of the original edge. The formula $\mathcal{A}$ ($\mathcal{B}$) can be transformed into a CTL formula $\mathcal{A}'$ ($\mathcal{B}'$) such that $\mathcal{A}$ ($\mathcal{B}$) is satisfied at a marking M (binding element e) if and only if $\mathcal{A}'$ ($\mathcal{B}'$) is satisfied at the unique state in the R-structure that corresponds to M (e). The translation of formulas is straightforward. CTL's X (next) operator is used to simulated the "switch" between state and transition formulas. For the U (until) operator, we use atomic propositions to distinguish between states which correspond to markings and states which corresponds to binding elements.

In fact, our result implies that by performing the transformation as sketched above, we could have used a standard CTL-model checker. However, we have chosen to avoid this transformation step for several reasons, the major being that our model checker is easier to implement directly in the Design/CPN environment [7].

7 Model Checking the ASK-CTL Logic

In this section we present an improved model checking algorithm. The approach is based on the "local model checking idea" from [13].

In [1] the complexity of model checking for a similar logic is shown to be linear in the product of the size of the formula and the size of the state space. We obtain the same worst case complexity result with ASK-CTL, assuming that the predicates can be evaluated efficiently, i.e., $O(N(V + E))$ where N is the length of the formula, V is the number of nodes, and E is the number of edges in the state space.

As the second contribution of this paper, we describe our improved model checking algorithm. The concept of strongly connected components allows us to improve the standard model checking algorithm [1].

7.1 Strongly Connected Component Graphs

We use a special kind of graphs derived from state spaces, namely strongly connected component graphs (SCC-graphs). In figure 5 a partial SCC-graph is shown for the Multi State Process example. The SCC-graph is indicated with large gray nodes and thick arrows. The underlying state space is shown with small nodes and thin arrows.

An SCC-graph is a graph where each node is a strongly connected component (SCC). Each SCC represents a subset of nodes in the state space with the property that each node is reachable from any other node in the subset. These subsets are mutually disjoint, maximal, and are a partition of the states in the state space. There is an edge between two SCC's in the SCC-graph if there is an edge between two nodes, one in each of the two SCC's. SCC-graphs are acyclic.

7.2 A More Efficient Algorithm

Our model checking algorithm is a modification of the standard algorithm given in [1]. We optimise the standard algorithm for some combinations of ASK-CTL formulas, partly by means of reduction rules, and partly by exploiting the SCC-graph. In the following we show how.

All formulas are expanded to the basic primitives of the logic, and reduced to eliminate redundant parts of the formula, e.g., $\neg(\neg\mathcal{A})$ is reduced to $\mathcal{A}$.

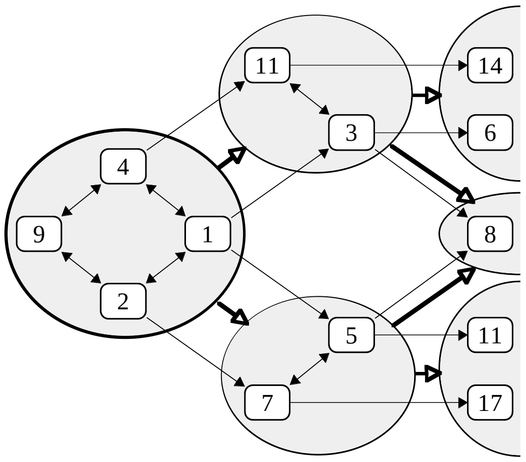


Figure 5: The partial SCC-graph for the Multi Stage Process example. Thin line graphics indicate the underlying state space.

We optimise the checking of formulas that are combinations of $EU(tt, \cdots)$, $AU(tt, \cdots)$, and $\neg$ (i.e., essentially combinations of *Pos*, *Inv*, *Ev*, and *Along*). Listing all combinations of two, we recognise eight basic patterns (f is either a state or a transition formula):

1. $EU(tt, EU(tt, f))$
2. $AU(tt, EU(tt, f))$
3. $EU(tt, AU(tt, f))$
4. $AU(tt, AU(tt, f))$
5. $EU(tt, \neg EU(tt, f))$
6. $AU(tt, \neg EU(tt, f))$
7. $EU(tt, \neg AU(tt, f))$
8. $AU(tt, \neg AU(tt, f))$

The first four patterns can be optimised by reduction rules, the next three patterns can be model checked more efficiently taking advantage of the SCC-graph, while the last pattern does not seem to have such property.

Other formula combinations exist with, e.g., $<f>$. Unfortunately we have not been able to improve the model checking algorithm for other cases by taking into account the SCC-graph.

The three formula patterns 1–3 above can easily be reduced to $EU(tt, f)$. Furthermore, the pattern 4 can be reduced to $AU(tt, f)$. We omit the formal proof here.

Instead we explain informally why the pattern 2 is the same as $EU(tt, f)$ (only the state formula is considered in this section to simplify the discussion). Assume $AU(tt, EU(tt, \mathcal{A}))$ and a given initial state M_0. This formula says that eventually a state is reached from where it is possible to reach a state, $M_{\mathcal{A}}$, where $\mathcal{A}$ holds. Then certainly it is possible to reach $M_{\mathcal{A}}$ from M_0, thus $EU(tt, \mathcal{A})$ also holds. Conversely assume that $EU(tt, \mathcal{A})$ holds, i.e., it is possible to reach a state $M_{\mathcal{A}}$ from M_0 where $\mathcal{A}$ holds. Observe that eventually a state is always reached (viz. M_0 in zero steps) from where it is possible to reach $M_{\mathcal{A}}$ (our assumption). Thus $AU(tt, EU(tt, \mathcal{A}))$ holds. In general we can conclude that the following is a sound reduction rule: $AU(tt, EU(tt, f)) \equiv EU(tt, f)$.

Similar arguments apply for the pattern 3 while the reduction rules for the patterns 1 and 4 are more straightforward to prove.

The three patterns 5–7 (containing one negation) can all be optimised using the SCC-graph. However, the pattern 8 does not seem to have similar properties, and is thus not considered further. Below we illustrate the optimisation idea for one of the three patterns (again, only the state formula is considered to simplify the discussion).

We use the formula pattern 5 as an example. In order to make the following discussion more intuitive we negate the formula. Thus consider the formula in question; $h(\mathcal{A}) = \neg EU(tt, \neg EU(tt, \mathcal{A}))$. The outer part $\neg EU(tt, \neg\mathcal{A}')$ means that it is not possible to reach a state in which $\mathcal{A}'$ does not hold, where $\mathcal{A}' = EU(tt, \mathcal{A})$. This is equivalent of saying that $\mathcal{A}'$ holds invariantly. The inner part $\mathcal{A}'$ says that there exists a path to a state in which $\mathcal{A}$ holds, i.e., it is possible to reach a state where $\mathcal{A}$ is true. Thus the whole formula says that no matter where you go, it is possible from there to reach a state in where $\mathcal{A}$ holds, i.e., $Inv(Pos(\mathcal{A}))$.

How do we model check such a formula using the SCC-graph? In order to motivate the usage of SCC's, consider for the moment the specific case where $\mathcal{A}$ identifies a set of markings. Now $h(\mathcal{A})$ expresses the home space property as defined in section 4.3 of [4]. In [6] section 1.4, proposition 1.14 indicates that home spaces are related with SCC-graphs. In the proposition it is stated that a set of markings, X, is a home space iff there exists a marking from X in each of the terminal SCC's. In general $h(\mathcal{A})$ can be checked by only considering the terminal SCC's. If $\mathcal{A}$ holds somewhere in each terminal SCC, then $h(\mathcal{A})$ also holds, and vice versa. We omit the formal proof here.

This means that the complexity of checking this formula is linear in the sum of sizes of the terminal SCC's (times the size of the formula). We gain a significant improvement in the performance when the number of nodes and edges in the terminal SCC's are small compared with the full graph. If the SCC-graph consists of only one node which is the worst case, we get the same performance as with the original algorithm.

Similar significant performance improvements can be found for the remaining two cases (6–7) of formula patterns to consider.

8 Performance Measures

Above we have shown that a set of formula patterns can be model checked more efficiently compared to the standard algorithm, when taking into account SCC-graphs. In practice we can compare implementations of the standard algorithm and our improved algorithm by making performance measures on state spaces generated from specific CP-nets. We use the three CP-nets already presented (section 5). These examples result in three very different state spaces and SCC-graphs. Thus the examples provide reasonably representative material for a variation of experiments.

In the subsections following, we first show the characteristics of the state spaces of the example CP-nets. Then we show that we gain significant performance improvements with our improved model checking algorithm.

8.1 State Spaces of Example CP-nets

The size of the state space are given for each example (see table 1). All computations in this paper are made on a Sparc20 workstation. The generation time information is included to give an impression of the relative speed of the computations[4].

We now describe the characteristics of the state spaces of the three examples used in this paper. The Dining Philosophers is an example of a totally cyclic system. This implies that the initial state is reachable from any reachable state. From this we conclude that there is exactly one SCC in the SCC-graph containing all reachable states of the CP-net.

Simulating the Integer Rearrangement example always terminates in a finite number of steps. This implies that the state space does not have any cycles. A totally acyclic state space has

[4]Note that there is not a direct connection between the number of nodes+arcs and the generation time, since the generation time depends on the topology of the state space and the complexity of the CP-net inscriptions.

		The Dining Philosophers	Rearrangement Problem	Multi Stage Process
SS-graph	nodes	322	895	578
	arcs	2136	2548	2498
	gen. time	14 sec	10 sec	5 sec
SCC-graph	components	1	895	11
	terminal comp.	1	1	2
	gen. time	4 sec	1 sec	1 sec

Table 1: Characteristics of the three examples.

an isomorphic SCC-graph with only trivial components, i.e., an SCC for each node of the state space.

The Multi Stage Process example has a behaviour which includes both local cycles and non-reversible changes between stages of the behaviour. The full SCC-graph of the Multi Stage Process example is shown in figure 6. For each SCC we have shown the identity of the component (a natural number), the number of states, and internal transitions. For each arc connecting two SCC's we have indicated the number of binding elements between these SCC's.

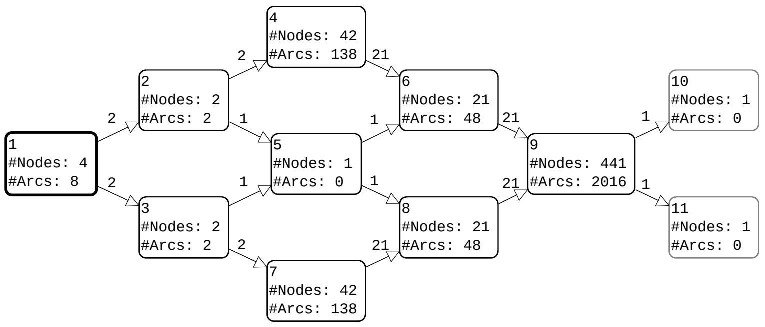


Figure 6: The SCC-graph for the Multi Stage Process example. The node with a thick border contains the initial state.

8.2 Performance Comparison of Algorithms

We have implemented two versions of the model checking algorithm on top of the Design/CPN tool [7], one corresponding to the standard algorithm presented in [1], and the other which is the standard algorithm including the above described improvements taking advantage of the SCC-graph.

To investigate the performance of our implementation of the model checking algorithm we have measured the time to check some formulas from section 5.1 using both the standard algorithm and the improved algorithm proposed above. The formulas are used on the examples from section 4.1 and 5.1 (see also table 1). The results in table 2 show a significant speed-up for all basic formula patterns (section 7.2), here ranging from a factor 2.4 to approximately 1300.

Note that the second formula for the Rearrangement Problem does not show any performance improvements, since the formula does not match any of the basic patterns. Furthermore, note the dramatic difference between the formula combinations $Inv(Pos(IsFailed))$ and $Inv(Ev(IsFailed))$, the reason being that the latter often requires the checking of more SCC's compared to the former formula.

	Standard Algorithm	Improved Algorithm	Speedup factor
Dining Philosophers			
$Inv(Pos(IsInitial))$	440 msec	0.33 msec	1333
Rearrangement Problem			
$Inv(Ev(IsSorted))$	1330 msec	103 msec	13
$IsSorted \vee <EU(\beta_{\{t1,t2,t'\}}, <IsSorted>)>$	60 msec	60 msec	1
Multi Stage Proc.			
$Inv(Pos(IsFailed))$	740 msec	1.02 msec	725
$Inv(Ev(IsFailed))$	4.1 msec	1.7 msec	2.4

Table 2: Performance measures for the three examples.

9 Conclusion

Three factors determine the usefulness of having a logic to express behavioural properties in terms of state spaces of CP-nets. First of all the logic must be sufficiently powerful to express interesting properties of the behaviour. Secondly, there must exist efficient algorithms to validate the properties. Finally, the implementation must be able to handle interesting problems, i.e., combinations of large state spaces and complex formulas.

We have provided a CTL-like logic which can express interesting properties about state spaces of CP-nets. In particular the logic allows properties of both states and transitions to be expressed directly. This duality gives a very direct formulation of standard CP-net properties such as liveness and home properties. At the same time we have shown how a linear time model checker for the logic still can be applied. Our model checker has been implemented on top of Design/CPN, which is a commercially available tool based on CP-nets. The tool offers the possibility of automatic generation of the full state space graph of a CP-net. As we have access to the full state space graph we can, in some important cases, improve the performance of the standard CTL model checking algorithm by exploiting strongly connected components. We have presented empirical results which show that, in some cases, our technique is much more efficient than the standard CTL model checking algorithm.

Contrary to the work of, e.g., Valmari [14] and Jensen [6] our technique does not perform any state space reduction. Our model checking technique is "orthogonal" to the symmetry based state space reduction technique described by Jensen [6]. The symmetry reduced state space of a net contains all information about the full state space. Future work should investigate the possibilities of applying the technique proposed in the present paper on symmetry reduced state spaces. We expect this to be possible under certain restrictions on the properties to be verified. Such properties could, e.g., be that predicates are invariant under symmetries.

Acknowledgements. We wish to thank Kurt Jensen for numerous constructive comments on earlier versions of the paper.

References

[1] Edmund M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[2] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In Kim G. Larsen and Arne Skou, editors, *Proc. CAV'91, Computer-Aided Verification*, pages 332–343. Springer-Verlag (*LNCS* 575), 1991.

[3] P. Huber, A. M. Jensen, L. O. Jepsen, and K. Jensen. Reachability trees for high-level petri nets. *Theoretical Computer Science*, 45(3):261–292, 1986.

[4] K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.

[5] K. Jensen. An Introduction to the Theoretical Aspects of Coloured Petri Nets. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *Lecture Notes in Computer Science*, pages 230–272. Sprinter-Verlag, June 1993.

[6] K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1994.

[7] K. Jensen, S. Christensen, P. Huber, and M. Holla. *Design/CPN. A reference manual.* Meta Software Corporation, 125 CambridgePark Drive, Cambridge MA 02140, USA, 1991.

[8] Ekkart Kindler and Rolf Walter. Rearranging problems. *Petri Net Newsletter*, 43:5–8, May 1993.

[9] Kim G. Larsen. Proof systems for Hennessy-Milner logic with recursion. In *Proceedings of CAAP, Nancy France*, pages 215–230. Springer-Verlag (*LNCS* 299), March 1988.

[10] D. Lehmann, A. Pnueli, and J. Stavi. Impartiality, Justice and Fairness: The Ethics of Concurrent Termination. In S. Even and O. Kariv, editors, *Automata, Languages and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 264–277. Springer-Verlag, 1981.

[11] Robin Milner. *Communication and Concurrency.* Prentice Hall International Series In Computer Science, C. A. R. Hoare series editor, 1989.

[12] Colin P. Stirling. Modal and temporal logics for processes. Technical report, Department of Computer Science, University of Edinburgh, August 1993. Stirling-1, Hand-outs at Summerschool in Logical Methods In Concurrency Aarhus'93.

[13] Colin P. Stirling and David Walker. Local model checking in the modal mu-calculus. Technical Report ECS–LFCS–89–78, Laboratory for Foundations of Computer Science, Department of Computer Science – University of Edinburgh, May 1989.

[14] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, pages 491–515. Springer-Verlag (*LNCS* 483), 1990.

[15] Pierre Wolper and Patrice Godefroid. Partial-order methods for temporal verification. Technical report, Université de Liège, Institut Montefiore, August 1993. Wolper-1, Hand-outs at Summerschool in Logical Methods In Concurrency Aarhus'93, To appear in: Concur'93 Proceedings.