Example 4.2 [Casandreas & LePoutre]
Consider the Petri net graph shown in Fig. 4.2. The Petri net it represents is specified by

\[ P = \{p_1, p_2, p_3, p_4\} \quad T = \{t_1, t_2, t_3, t_4\} \]

\[ A = \{(p_1, t_1), (p_1, t_2), (p_2, t_3), (p_3, t_4), (p_4, t_4), (t_1, p_1), (t_1, p_2), (t_2, p_3), (t_3, p_4), (t_4, p_3), (t_5, p_5)\} \]

\[ w(p_1, t_1) = 1 \quad w(p_1, t_2) = 1 \quad w(p_3, t_3) = 1 \quad w(p_4, t_4) = 2 \]

\[ w(p_2, t_3) = 1 \quad w(p_4, t_4) = 1 \quad w(t_1, p_1) = 1 \quad w(t_1, p_2) = 1 \]

\[ w(t_2, p_3) = 1 \quad w(t_3, p_4) = 1 \quad w(t_4, p_3) = 1 \]

\[ w(t_5, p_5) = 1 \]

---

Example 2.11 A marked Petri net. The Petri net structure is the same as Figures 2.1 and 2.4. The marking is [Peterson, J.L. Petri Net Theory and the Modeling of Systems, Prentice Hall, 1981.]

Examples

**MPN1:**

**MPN2:**

**MPN3:**

**MPN4:**

Figure 2.1 A Petri net structure represented as a 4-tuple. [Peterson]
Pearl net dynamics from Peterson:

Figure 2.15 A marked Pearl net to illustrate the firing rules. Transitions and are enabled.

Figure 2.16 The marking resulting from firing transition $t_4$ in Figure 2.15.

Figure 2.17 The marking resulting from firing transition $t_2$ in Figure 2.16.
**PETRI NET MODELING**

*FROM PETERSON*

Figure 3.20 A portion of a Petri net modeling a control unit for a computer with multiple registers and multiple functional units.

Figure 3.28 Mutual exclusion. Access to the critical sections of the two processes is controlled so that both processes cannot simultaneously execute their critical sections.

Figure 3.30 A Petri net representing the oxidation-reduction of oxalic acid and hydrogen peroxide into carbon dioxide and water.

Figure 3.32 The dining philosophers problem. Each philosopher is modeled by two places, meditating ($M_i$) and eating ($E_i$).
Petri Net Modeling.

Modeling a Queue + Server

How would you keep track of completed parts? (# of firings of c)

How would you model a queue with capacity 11?

An assembly which requires one of part 1 and three of part 2?
Example 4.6 (Dining philosophers)

Let us consider the example of two “dining philosophers” that we described in Example 2.17 in Chapter 2. Figures 2.15 and 2.16 show an automaton model of this system; the complete model is obtained by doing the parallel composition of four component models, one for each philosopher and one for each fork. Figure 4.9 shows a Petri net model of the same system, with the same set of events. In part (a) of Fig. 4.9, we have the model of one philosopher. There is a place corresponding to the condition “philosopher eating” and a place corresponding to the condition “philosopher thinking.” This model also accounts for the availability or unavailability of each fork since it includes two “fork” places. In this sense, this model is more elaborate than automaton \( P_1 \) in Fig. 2.16. The reason for using the two fork places is to allow easy interconnection of the two philosophers. If we create a second copy of Fig. 4.9 (a) for the second philosopher, then a little thought shows that the complete system model is obtained by combining the Petri net models of the two philosophers, where the fork places are simply overlapped while all other places, all transitions, and all arcs are preserved. This complete model is shown in Fig. 4.9 (b). This merger of the fork places works because the two forks are shared resources, and thus the conditions “fork available” and “fork unavailable” are common to the two component models. The initial state of the Petri net is obtained by placing four tokens, namely one in each of the places corresponding to: philosopher 1 thinking, philosopher 2 thinking, fork 1 available, and fork 2 available.

This example illustrates that while we could have constructed the complete Petri net model by transforming the finite-state automaton of the complete system model (two philosophers and two forks) shown in Fig. 2.16 into a Petri net using the method described earlier in this section, a model derived component by component from the original system description and associating places with conditions governing the operation of the system (as opposed to associating places with system states) is much more intuitive and modular.

Figure 4.9: Petri net model of the dining philosophers example, with two philosophers and two forks. (Example 4.6).
Part (a) shows the model of one philosopher interacting with two forks. Part (b) shows the complete model for two philosophers and two forks.
(c) FIRE $t_{1,1}$ THEN $t_{c,1}$

"1. EATING, 2. WANNING"

philosopher 1          fork 1 available          philosopher 2

philosopher 1          fork 1 available          philosopher 2

(d) FIRE $t_{1,1}$ THEN $t_{1,2}$

"DEADLOCK"

philosopher 1          fork 1 available          philosopher 2

(e): Proposed "fix" to single-philosopher Petri net.
Complete model is obtained by repeating as before, with "fork available" states shared.
/* Protocol 3 (par) allows unidirectional data flow over an unreliable channel. */

#define MAX_SEQ 1 /* must be 1 for protocol 3 */

typedef enum {frame_arrival, oksum_arr, timeout} event_type;

#include "protocol.h"

void sendr2(void)
{
    seq_nr next_frame_to_send;
    frame s;
    packet buffer;
    event_type event;

    next_frame_to_send = 0;
    from_network_layer(&buffer);

    while (true) {
        a.info = buffer;
        s.seq = next_frame_to_send;
        to_physical_layer(&s);
        start_timer(s.seq);
        wait_for_event(&event);

        if (event == frame_arrival) {
            from_physical_layer(&a.s);
            s.ack = next_frame_to_send;
            from_network_layer(&buffer);
            inc(next_frame_to_send);
        } /* get the acknowledgement */
        /* seq number of next outgoing frame */
        /* scratch variable */
        /* buffer for an outbound packet */

        /* initialize outbound sequence numbers */
        /* keep first packet */

        /* construct a frame for transmission */
        /* insert sequence number in frame */
        /* send it on its way */
        /* if answer takes too long, time out */
        /* frame_arrival, oksum_arr, timeout */

        /* get the next one to send */
        /* insert next_frame_to_send */
    }
}

void receive2(void)
{
    seq_nr frame_expected;
    frame s;
    event_type event;

    frame_expected = 0;

    while (true) {
        wait_for_event(&event);

        if (event == frame_arrival) {
            from_physical_layer(&s);
            if (s.ack == frame_expected) {
                to_network_layer(&s.info);
                inc(frame_expected);
            }
            s.ack = 1 - frame_expected;
            to_physical_layer(&s);
        } /* tell which frame is being asked */
        /* none of the fields are used */

        /* possibilities: frame_arrival, oksum_arr */
        /* a valid frame has arrived */
        /* go get the newly arrived frame */
        /* this is what we have been waiting for. */
        /* pass the data to the network layer */
        /* next time expect the other sequence nr */
    }
}

Fig. 3-23. A Petri net model for protocol 3.

Fig. 3-11. A positive acknowledgement with retransmission protocol.

SOURCE: A. S. TANENBAUM, COMPUTER NETWORKS, PRENTICE HALL, 1996, THIRD EDITION.
Capacity Planning of Web Servers using Timed Hierarchical Coloured Petri Nets
(Abridged Version)

Søren Christensen¹, Lars M. Kristensen¹, Kjeld H. Mortensen¹, and Jan S. Thomassen²

¹ HP-CPN Centre, Department of Computer Science, University of Aarhus, Denmark
(schristensen,lmkristensen,khm)@daimi.au.dk

² Hewlett-Packard Corporation, Advanced Technology Center, Roseville, CA.
jan.thomassen@hp.com

Abstract

This paper presents the main results of a research project conducted in cooperation between the CPN group and Hewlett-Packard Corporation on capacity planning and performance analysis of distributed computing environments. We present a general framework for modelling distributed computing environments for capacity planning by means of Timed Hierarchical Coloured Petri Nets. For validation of the proposed framework we build a Coloured Petri Net model of a HTTP web server and compare the performance results obtained by simulation with the performance measured in a corresponding physical environment. We demonstrate that the performance results obtained by simulation are close to those of the physical web server.

Keywords: Timed Hierarchical Coloured Petri Nets, Web Servers, Capacity Planning, Performance Evaluation, Performance Analysis, Simulation.

1 Introduction

The Internet and the World Wide Web (WWW) have experienced exponential growth since their inception. Popular web sites receives millions of hits per day, and it is not uncommon for these sites to exhibit extremely high response times. High response times is a source of frustration for users, and with the growing use of web sites for, e.g., electronic commerce this may damage the reputation of the company offering the web site, leading to loss of business. As a consequence, it is important to be able to identify bottlenecks, predict future capacity shortcomings, and determine the most adequate or cost effective way to reconfigure such distributed computing environments to overcome performance problems and cope with increasing workload demands. This is also referred to as capacity planning [20, 21].

This paper presents the main results of the CAPLAN project, a research project conducted as a cooperation between the CPN Group at the University of Aarhus and Hewlett-Packard (HP) Corporation on performance analysis and capacity planning of distributed computing environments. The overall goal of the CAPLAN project has been to investigate the use of Coloured Petri Nets (CP-nets or CPNs) [15–18] and simulation as an underlying technology for performance analysis and capacity planning of distributed computing environments.

CP-nets provide a framework for construction and analysis of distributed and concurrent systems. A CPN model of a system describes the states which the system may be in and the transitions between these states. CP-nets have been applied in a wide range of application areas and many projects have been carried out in industry [17] and documented in the literature, e.g., in the areas of communication
3 Basics of processor modelling

We model instruction types by first defining a set of predefined identifiers using an enumerated colorset as follows:

\[
\text{Color Instr} = \text{with INT, FPADD, MUL, DIV, BRA, NOOP;}
\]

It is then possible to create a record colorset using appropriate fields to model an instruction completely:

\[
\text{color Value} = \text{record}
\]

\[
\begin{align*}
\text{nc: Line} & \ast \\
\text{instr: Instr} & \ast \\
\text{d: Dep} & \ast \\
\text{d': Dep} & \ast \\
\text{t: Target timed} & 
\end{align*}
\]

assuming that the instruction has dependencies \(d, d'\) and a destination \(t\), which links those instructions sharing the same target register. It must be

---

**Figure 1:** Place-Transition net model of an asynchronous processor.

**Figure 2:** Design/CPN model of a superscalar processor.
the set of world states satisfying a goal. In a state-space search over these data structures, the data structures are states of the planning process. To distinguish between planning states and world states, I will usually refer to the former as state descriptions.

To avoid various technical difficulties, which need not concern us here, I restrict both the before- and after-action state descriptions to conjunctions (or sets) of ground literals. I allow one exception to this restriction, namely, state descriptions can contain arbitrary formulas true in all states—such as \( \text{in}((x, y)) \) restricted by \( y = 1 ) \). I restrict goals to (possibly existentially quantified) conjunctions of literals. In what follows, I write goal wffs of the form \( (\exists x_1, x_2, \ldots, x_n) \psi(x_1, x_2, \ldots, x_n) \) simply as \( \psi(x_1, x_2, \ldots, x_n) \)—assuming existential quantification of all the variables. Although these restrictions can be lifted in various ways, several interesting planning problems can be posed and solved even with them in place.

Given a goal wff \( \psi \), our search methods attempt to find a sequence of actions that produces a world state described by some state description \( \delta \), such that \( \delta \models \psi \). We say then that the state description satisfies the goal for our restricted state and goal wffs, \( \delta \models \psi \) when there exists a substitution, \( \sigma \), such that \( \psi \sigma \) is a conjunction of ground literals each of which appears in \( \delta \). Whether or not \( \delta \models \psi \) can be established by attempting to unify the first literal in \( \psi \) with a literal in \( \delta \), applying the unifying substitution to the rest of the literals in \( \psi \) and continuing for all of the literals in \( \psi \). (This process is the same as that used by the PROLOG interpreter when it checks to see if the body of a clause can be unified away with facts.)

We can search a space of state descriptions either in a forward direction, from start to goal, or in a backward direction, from goal back to start. Some authors refer to planning methods using forward search as progression planning and methods using backward search as regression planning. I begin with forward search.

### 22.1.2 Forward Search Methods

To do forward search over the space of state descriptions, we will need operators corresponding to actions—operators that change a before-action state description into an after-action state description. The search completes successfully when it produces a state description \( \delta \), such that \( \delta \models \psi \) for the goal wff \( \psi \). Our operators are on those of a system called STRIPS [Fikes & Nilsson 1971, Fikes, Hart, & Nilsson 1972]. A STRIPS operator consists of three parts:

1. A set, \( PC \), of ground literals called the preconditions of the operator. An action corresponding to an operator can be executed in a state only if all of the literals in \( PC \) are also in the before-action state description.
2. A set, \( D \), of ground literals called the delete list.
3. A set, \( A \), of ground literals called the add list.

To produce the after-action state description, we first delete from the before-action state description any literals in \( D \) and then add all of the literals in \( A \). All literals not mentioned in \( D \) carry over to the after-action state description. This carryover is called the STRIPS assumption and is one method of attacking the frame problem.

STRIPS operators are usually defined by schemata that correspond to action schemata. An operator schema, called a STRIPS rule, has free variables, and it is the ground instances of these rules that correspond to actual operators. Here is an example of a STRIPS rule with free variables \( x, y, ) and \( z \):

\[
\text{move}(x, y, z)\\
PC: \text{in}(x, y) \land \text{Clear}(x) \land \text{Clear}(y)\\
D: \text{Clear}(z), \text{in}(x, y)\\
A: \text{in}(x, y), \text{Clear}(x), \text{Clear}(z)
\]

(I include the formula \( \text{Clear}(z) \) explicitly in the add list because it is in \( \psi \), \( \text{Clear}(z) \) would be deleted and we want it always to be true.) An example of an instance of this STRIPS rule for the action of moving block \( B \) from \( A \) to \( B \) is shown in Figure 22.2.

Note that the precondition of a STRIPS rule itself is in the form of a goal, namely, a conjunction of literals. This is not a coincidence and will be exploited. When interpreted as a goal wff, the free variables in \( PC \) are assumed to be existentially quantified. An instance of a STRIPS rule can be applied to a state description \( \delta \), if a ground instance of the \( PC \) (considered as a goal) is satisfied by the state description. As mentioned earlier, such an instance can be found by unifying, in turn, each of the literals in the \( PC \) with a literal in the state description—applying the unifying substitution to the remaining literals in the \( PC \). The applicable operator instance is then obtained by applying the resulting substitution to all of the elements of the STRIPS rule. Rules must be written in such a way that applying such a substitution always results in a ground instance of the rule. That is, there can be no free variables in \( D \) or \( A \) that do not occur in \( PC \). Figure 22.2 is an illustration of the application of a STRIPS operator.