ISSN 0105-8517

Petri Nets 2000

21st International Conference on
Application and Theory of Petri Nets

Aarhus, Denmark, June 26-30, 2000

Workshop Proceedings
Practical Use of High-level Petri Nets

Organised by

Kurt Jensen

DAIMI PB – 547
June 2000

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF AARHUS
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark

Preface

This booklet contains the proceedings of the Workshop on Practical Use of High-level Nets, June 27, 2000. The workshop is part of the 21st International Conference on Application and Theory of Petri Nets organised by the CPN group at Department of Computer Science, University of Aarhus, Denmark. The workshop papers are also available in electronic form via the web pages: http://www.daimi.au.dk/pn2000/proceedings

The aim of the workshop is to bring together researchers and practitioners with interests in the use of high-level nets and their tools for practical applications. The submitted papers were evaluated by a programme committee with the following members:

W. van der Aalst	The Netherlands	w.m.p.v.d.aalst@tm.tue.nl
G. Chiola	Italy	chiola@disi.unige.it
S. Donatelli	Italy	susi@di.unito.it
C. Girault	France	Claude.Girault@lip6.fr
N. Husberg	Finland	Nisse.Husberg@hut.fi
K. Jensen	Denmark (chair)	kjensen@daimi.au.dk
S. Kumagai	Japan	kumagai@pwr.eng.osaka-u.ac.jp
C. Lakos	Australia	Charles.Lakos@adelaide.edu.au
A. Levis	USA	alevis@gmu.edu
D. Moldt	Germany	moldt@informatik.uni-hamburg.de
L. Petrucci	France	petrucci@lsv.ens-cachan.fr
W. Reisig	Germnay	reisig@informatik.hu-berlin.de
M. Silva	Spain	silva@posta.unizar.es
E. Stear	USA	estear@aol.com
R. Valette	France	robert@laas.fr
K. Voss	Germany	klaus.voss@gmd.de
W. Zuberek	Canada	wlodek@cs.mun.ca

The programme committee was assisted by the following referees:

S. Christensen	C. Dutheillet	L. M. Kristensen	B. Lindstrøm
T. Mailund	K. H. Mortensen	E. Paviot-Adet	D. Poitrenaud
T. Wareham	L. M. Wells		

The programme committee has accepted 8 papers for presentation. Most of these deal with different projects in which high-level nets and their tools have been put to practical use – often in an industrial setting. The remaining papers deal with different extensions of tools and methodology.

After an additional round of reviewing and revision, some of the best papers from the workshop will be published as a special section in the International Journal on Software Tools for Technology Transfer (STTT). For more information see: http://sttt.cs.uni-dortmund.de/

Kurt Jensen

Table of Contents

Executable Petri Net Models for the Analysis of Metabolic Pathways

Hartmann Genrich, Robert Küffner, Klaus Voss

GMD – German National Research Center for Information Technology
Institute for Algorithms and Scientific Computing (SCAI)
Schloss Birlinghoven, D-53754 Sankt Augustin, Germany

Abstract

Computer simulation of biochemical processes is a means to augment the knowledge about the control mechanisms of such processes in particular organisms. This knowledge can be helpful for the goal oriented design of drugs. Normally, continuous models (differential equations) are chosen for modelling such processes. The application of discrete event systems like Petri nets has been restricted in the past to low-level modelling and qualitative analysis. To demonstrate that Petri nets are indeed suitable for simulating metabolic pathways, the glycolysis and citric acid cycle are selected as well understood examples of enzymatic reaction chains (metabolic pathways). The paper discusses the steps that lead from gaining necessary knowledge about the involved enzymes and substances, to establishing and tuning high-level net models, to performing a series of simulations, and finally to analysing the results. We show that the consistent application of the Petri net view to these tasks has considerable advantages, and – using advanced net tools – reasonable simulation times can be achieved.

1 Introduction

Finding promising targets for the development of new drugs very much depends on certified knowledge about the metabolism (metabolic processes in the human organisms). Then, this knowledge can be exploited to avoid unnecessary, costly and dangerous experiments and to conduct the remaining unavoidable experiments effectively to the goal of finding drug targets.

Traditionally, the dynamics of metabolic processes is investigated by simulations on the basis of differential equations (e.g. [FrCa84, LePa92, ScHo95]). This is usually done by providing a particular kinetic equation for each reaction of the pathway requiring a considerable number of kinetic constants derived from experimental data. The simulation then proceeds by executing cyclically these equations (and updating the concentrations of the involved substances) in very small timesteps. A prominent example is E-CELL ([Tom99]), a particular software environment for whole-cell simulation. E-CELL offers, among others, graphical user interfaces to observe the cell's state and manipulate it interactively.

An alternative is the simulation by discrete event systems. Petri nets have been proposed in [RML93] because of their appropriate semantics (occurrence rule), the inherent precise concurrency notion, their intuitive graphical representation and their capabilities for (mathematical) analysis. [RLM96] stresses the straightforward representation of a metabolic reaction (cf. section 2). It demonstrates the significance of net abstraction, boundedness, S-invariants, T-invariants and liveness to draw important "preliminary conclusions about the metabolic pathway". However, the approach in [RLM96] aims at a purely qualitative analysis of biochemical pathways and does not allow to simulate quantitative kinetic effects. This is motivated by the fact that "modelling a complex biochemical system involves data that are incomplete, uncertain or unreliable". Fortunately, the availability of reliable data has improved during the last years although it still remains a serious problem.

To our knowledge, attempts to simulate metabolic pathways by Petri nets, up to now, are restricted to relatively small reaction chains modelled as low-level nets which are constructed more or less by hand. In [MDNM00], so-called "hybrid Petri nets" are chosen to model and simulate a gene regulatory network. This low-level net class contains discrete as well as continuous nodes, where continuous places are marked with real numbers (instead of unit tokens). Section 2 shows, however, that time intervals and real numbers can be handled easily in timed high-level nets and, hence, there is no obvious need to leave the standard Petri net classes with their advanced theory and tools.

What we aim at next, is the automatic creation and implementation of high-level Petri net models which allow the simulation and quantitative discussion of networks of metabolic processes. This paper describes a promising first step towards this goal and demonstrates it by use of a well-known example.

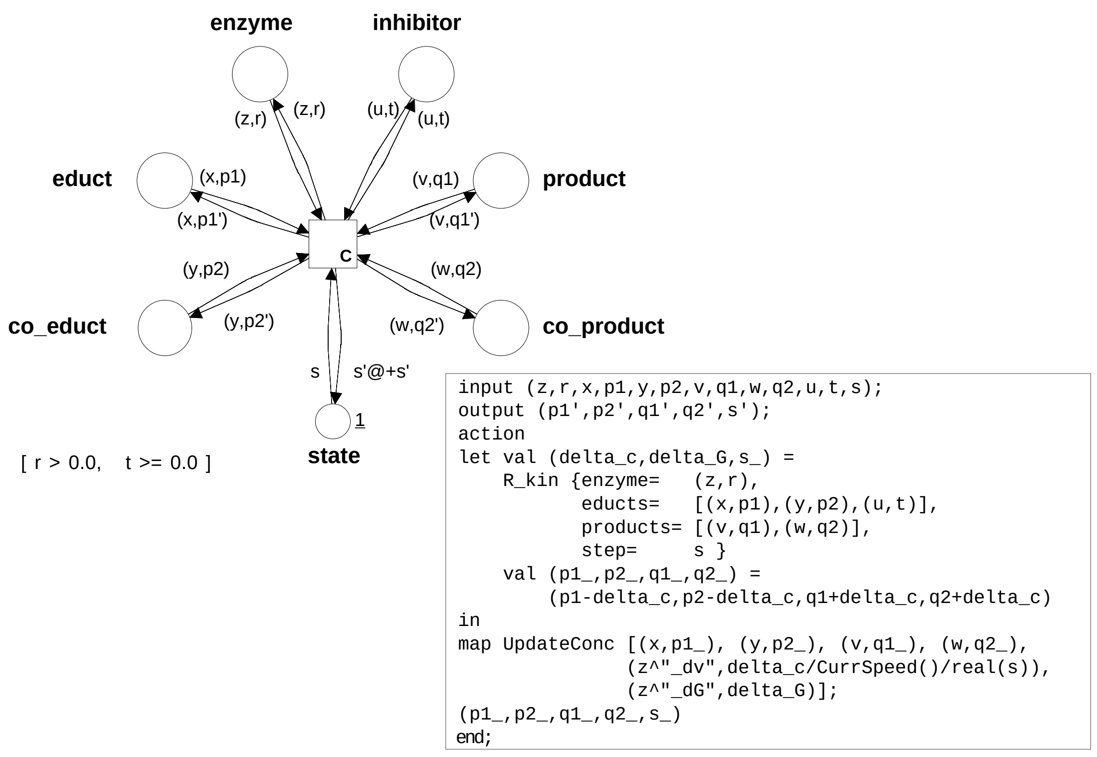
Section 2 introduces the basic notions and concepts used for representing a metabolic reaction as a Petri (sub-)net, and it discusses our choice of the kinetic reaction function. Section 3 deals with the problem of systematically constructing pathways for metabolic processes and of assembling the reactions and metabolic constants for the chosen pathways from the databases. Section 4 then explains the most prominent features of the Petri net models that have been developed for investigating the well-known processes contributing to the glycolysis and – in connection with this – the citric acid cycle. Section 5 presents some typical simulation results and performance figures, followed by a short paragraph with conclusions and some suggestions for the future work.

The discussed application runs on a Power Macintosh G3, using the software packages [Design/CPN] (for modelling and simulation), Excel (for plotting) and MacPerl (for data extraction from databases).

2 A Sample of a Reaction and the Kinetic Function

An enzymatic reaction changes the concentrations of the involved *substrates* (the substances participating in the reaction), catalyzed by a reaction-characteristic enzyme. Some reactions may be slowed down by substances called *inhibitors*. In principle, every enzymatic reaction is reversible; however, most of them have a preferred direction. Therefore, the substrates whose concentrations are decreased in this preferred mode are called *reactants* (called *educts* in this paper), those with increased concentrations are called *products*. The *speed* of the reaction is the amount of the concentration change within a time unit, $\Delta c \,/\, \Delta t$.

An enzymatic reaction can be modelled as a high-level Petri net transition in a straightforward manner (figure 1). Every substrate is represented as a place connected through an outgoing and an ingoing arc to the transition. For each of these places, its colour set is chosen to be a set of pairs, each consisting of the name and the concentration of a particular substrate. The code section of the transition calls two functions: α) the kinetic function *R_kin* that determines the speed of the reaction (see below) and β) the function



```
input (z,r,x,p1,y,p2,v,q1,w,q2,u,t,s);
output (p1',p2',q1',q2',s');
action
let val (delta_c,delta_G,s_) =
    R_kin {enzyme=   (z,r),
           educts=   [(x,p1),(y,p2),(u,t)],
           products= [(v,q1),(w,q2)],
           step=     s }
    val (p1_,p2_,q1_,q2_) =
        (p1-delta_c,p2-delta_c,q1+delta_c,q2+delta_c)
in
map UpdateConc [(x,p1_), (y,p2_), (v,q1_), (w,q2_),
               (z^"_dv",delta_c/CurrSpeed()/real(s)),
               (z^"_dG",delta_G)];
(p1_,p2_,q1_,q2_,s_)
end;
```

Figure 1: The model of an enzymatic reaction

UpdateConc that writes selected values, in particular the just computed new concentrations, into a plot file. After the end of the simulation, the plot file serves as input to Excel to produce a plot diagram of these values as a function of time.

For the calculation of the reaction speed we use a general (reaction independent) kinetic function *R_kin*. It is a reversible *Michaelis Menten* equation augmented by an additional term for the free reaction energy, thus combining kinetic and thermodynamic information. The parameters of the kinetic function *R_kin* are the current concentrations of all involved substances. It (essentially) computes the reaction speed, i.e., the decrease resp. increase Δc (in time unit 1) of the concentrations of the educts resp. products. The concentrations of the enzyme – being a catalyst – and of the inhibitor(s) remain unchanged. In the present implementation, *R_kin* only needs a few chemical and enzyme specific constants for its computation. Before starting any simulation, the constants of all relevant enzymes are extracted from the database [BRENDA] (see section 3) and collected in a data structure called *enzyme catalog*. Given a reaction (without inhibitors) catalyzed by enzyme e, let S resp. P denote the set of its educts resp. products. The concentration of a substance x shall be denoted by $[x]$. Then a slightly simplified version of *R_kin* reads:

$$\Delta c = (\overrightarrow{K} - \overleftarrow{K}) * k_{cat} * [e] * (\overrightarrow{K} * \overrightarrow{Q} + \overleftarrow{K} * \overleftarrow{Q})$$

with

$$\Delta G = \Delta G0 + R * T * ln\,\frac{\Pi_{s\in S}\,[s]}{\Pi_{p\in P}\,[p]}$$

$$K = e^{-\Delta G/R*T}\,,\ \ \overrightarrow{K} = \frac{1}{1+K}\,,\ \ \overleftarrow{K} = \frac{K}{1+K}$$

$$\overrightarrow{Q} = Min_{s\in S}\,\frac{[s]}{[s]+Km_s}\,,\ \ \overleftarrow{Q} = Min_{p\in P}\,\frac{[p]}{[p]+Km_p}$$

where R, T are constants of nature and $k_{cat}, \Delta G0, Km_x$ are enzyme constants contained in the enzyme catalog.

The feasibility of *R_kin* can be checked by applying it to four particular situations.

1 In the irreversible case it degenerates to the well known *Michaelis-Menten* equation $\Delta c = k_{cat} * [e] * [s] \,/\, ([s] + Km_s)$.

2 Always, $\Delta c \leq k_{cat} * [e]$ holds.

3 In case $\overrightarrow{Q} = \overleftarrow{Q} = 1$, Δc only depends on $\overrightarrow{K} - \overleftarrow{K}$.

4 $\Delta G = 0 \Rightarrow \Delta c = 0,\ \ \Delta G < 0 \Rightarrow \Delta c > 0,\ \ \Delta G > 0 \Rightarrow \Delta c < 0.$ [1]

3 Metabolic Pathways

As main sources of information on metabolic pathways the internet-accessible databases [BRENDA], [ENZYME] and [KEGG] are used. Entries of these databases describe one enzymatic function each and are indexed via their EC-number.[2] The chemical reaction equations contained in the database entries can be used for two purposes: first to define transitions of a Petri net, and second, to define a network of enzyme–substrate–enzyme edges via matching and identifying the educts and products of reactions.

The key problem here is the unification of the substrate names, due to different naming conventions. By manually augmenting existent alias lists, detection of typos etc. the contents of the diverse databases can be compared and compiled into a unified Petri net. In the actual state of the databases the unified Petri net contains about 3 200 EC-entries, 11 300 reactions (transitions) and 12 300 substrates (places) leading to 164 000 enzyme–substrate–enzyme edges.

For the purpose of simulating metabolic pathways derived from the databases (see below) kinetic enzyme parameters (*Michaelis-Menten* constants Km_x of the substrates x, maximum reaction velocity $k_{cat} * [e]$ of the enzyme e) are needed. Fortunately, BRENDA covers these parameters for a wide range of enzymes, substrates and organisms. However, only for a subset of the well known pathways those parameters are complete. So, in this paper, we restrict our analysis to the best examined pathways like glycolysis and citrate cycle.

The next step to go comprises developing an appropriate language to access the various entries of a database from within the CPN model. This language can then be applied to find the relevant reactions and to compute the necessary metabolic constants. Having checked these data for completeness, they can be inserted into the *enzyme catalog* which in turn will be inspected by the kinetic function *R_kin* during a simulation to compute the actual reaction speed (cf. section 2).

A (metabolic) path is a coherent set of enzymatic reactions. The reactions are interconnected via the substrates (educts and products) they act upon. In contrast to naive graphs, Petri nets allow for representing and distinguishing different constellations in biochemical networks which is a prerequisite for the systematic construction of pathways in

[1]Hence, $\Delta G0$, the change of the free enthalpy under standard conditions, determines that constellation of concentrations at which the (reversible) reaction changes its direction, i.e., the sign of Δc.

[2]The EC-numbers reflect the official classification of the enzymes. The first three numerals in the EC-number hierarchically define the type of the enzymatic function, the fourth numeral increments over different enzymes which catalyze the same function.

such nets. In particular, the difference between *branching reactions* (one reaction producing more than one product) and *conflicting reactions* (several reactions competing for the same educt) is of substantial importance (see below).

In the following, three rules are defined which shall serve to find sensible and manageable (in size and speed) pathways among the millions of possibilities.

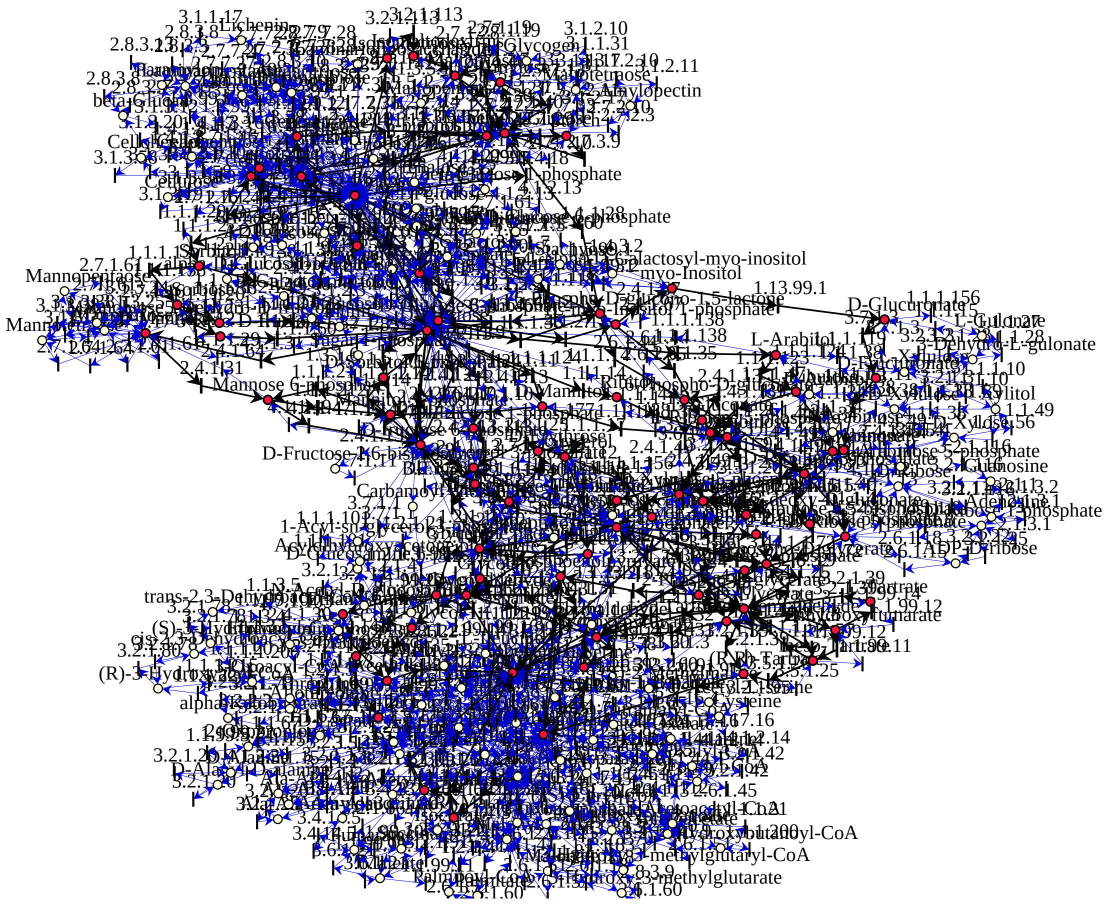


Figure 2: The complete glycolysis pathway

First rule. Inspired by the occurrence rule of Petri nets, only those paths – then called *pathways* – shall be considered which are *closed* in the sense that they take care of the availability of all educts and the consumption of all intermediate products. The result of the first rule is that there are no loose ends in such a pathway. An exception from this rule are small molecules like $H_2O, NADH, ADP, CO_2$ found in sufficiently large amounts in all organisms, called *ubiquitous* molecules.

The essential task prior to constructing a metabolic Petri net model is the sensible selection of the pathways to be modelled. To start with, the initial and final substrates, i.e., the *source* and the *sink* of the envisaged metabolic process have to be determined.

For example, the *glycolysis* comprises all metabolic processes leading from glucose as the initial to pyruvate as the final substance. An unrestricted search in the database [BRENDA] would result in about 500 000 paths (of a length of at most 9; not shown). Applying the above mentioned first rule leads to figure 2, showing about 80 000 paths. Clearly, the resulting number of pathways is still much too large to be handled.

Hence, a **second rule** is applied which mirrors the observation that very long paths connecting two substances contribute much less to the concentration changes of these

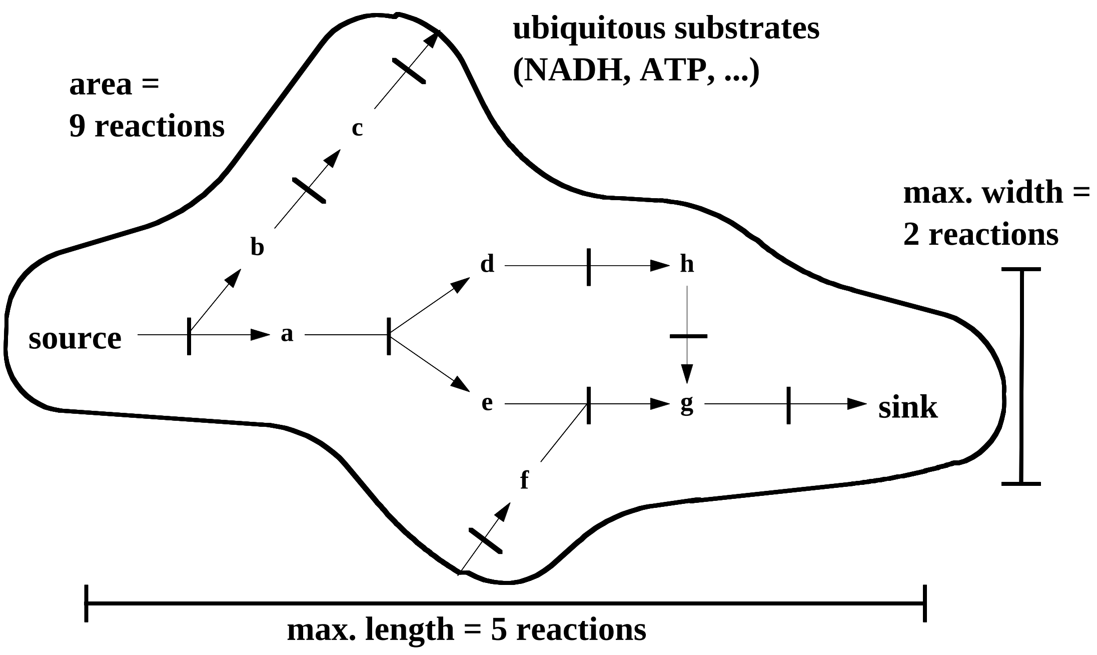


Figure 3: Pathway reduction principles

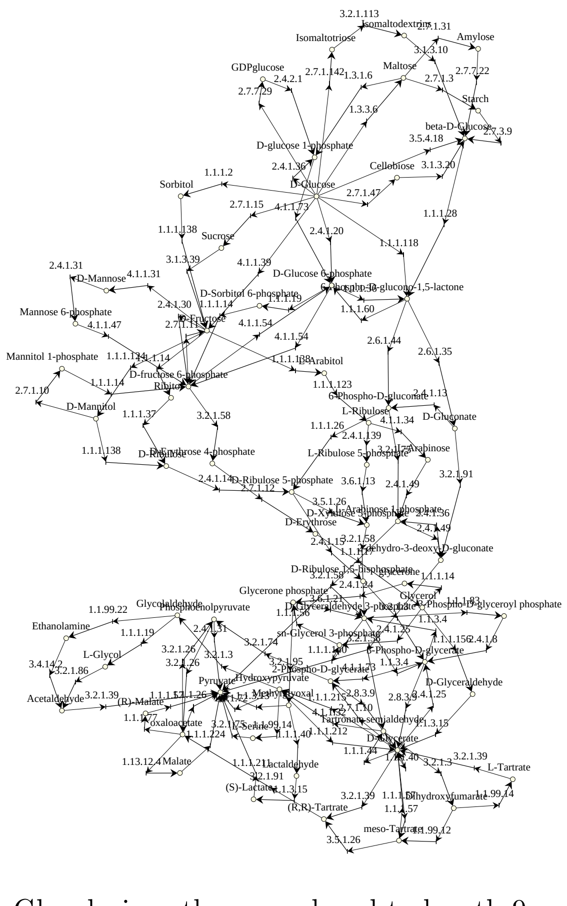


Figure 4: Glycolysis pathways reduced to length 9 and width 1

substrates than short ones. This second rule is depicted in figure 3. It cuts off all those paths which exceed a certain length and a certain width (max ST-cut). Delimiting the length of the glycolysis pathways to 9 and their width to 1, leads to figure 4 which contains 170 pathways. However, to include the simplified glycolysis as presented in most textbooks (figure 5), the width has to be at least 2, rendering 541 pathways (not shown).

As a **third rule**, which in practice should not be applied *after* but *before* or *in conjunction with* the former two rules, we restrict those pathways to contain only enzymes which exist in the organism under examination. Most of the mentioned metabolic databases include this information. For the sample organism yeast this finally results in 8 pathways for the glycolysis.

4 The Petri Net Models

Having chosen a set of closed pathways, we are at first interested in finding constellations for which its execution reaches a dynamic (i.e. flow) equilibrium: Given steady sources and steady sinks of a pathway, each substrate concentration must converge. To reach that goal, we model such pathways as a Petri net and then simulate it with diverse parameters.

Our first attempt to model metabolic pathways consists in simply concatenating the particular reactions by identifying the products of any reaction with the educts of the following one(s). In case of the textbook glycolysis pathway this leads to the intuitive model $\mathcal{A}$ of figure 5.

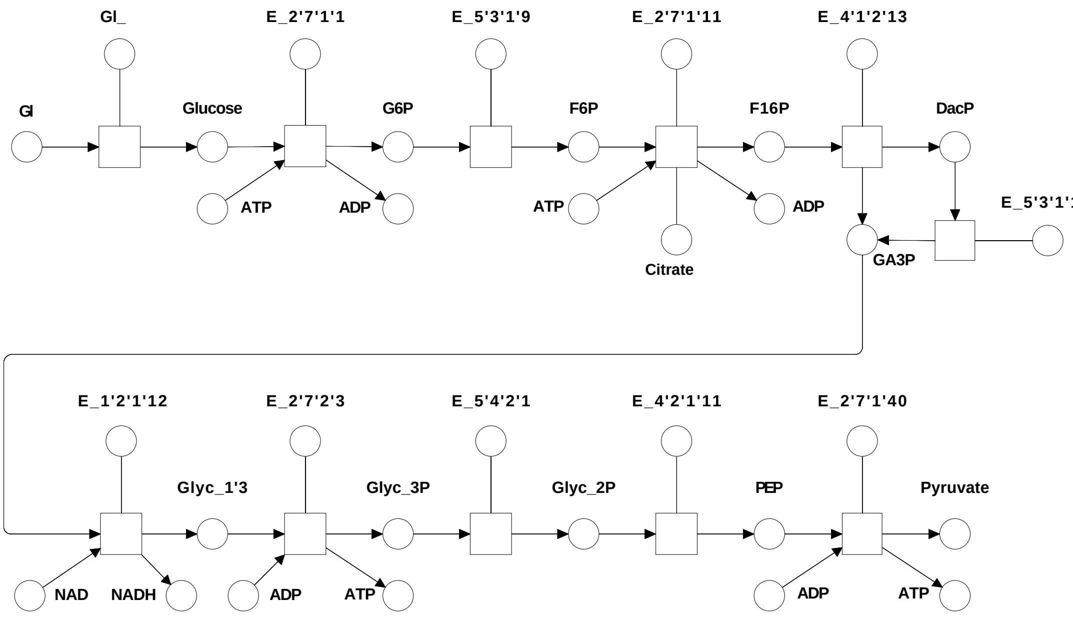


Figure 5: The model $\mathcal{A}$ of the textbook glycolysis

The model $\mathcal{A}$ is a timed high-level net diagram as expressed by Design/CPN.[3] Every

[3]Observe that, in figure 5, the arcs connecting the transitions with the substrates places are directed merely to indicate the preferred direction of the reaction (cf. section 2). In the corresponding subpages they are replaced by a pair of arcs, one pointing to and one from the transition because the substrate concentrations are changed by a transition occurrence. Undirected arcs (between enzyme places and reaction transitions) indicate that the enzyme concentrations stay unchanged but are needed for the kinetic reaction function. – A similar remark holds true for figure 8.

reaction in figure 5 is a substitution transition standing for the transition of a subpage like that shown in figure 1. Of course, the substituting subpages have to fit to the actual numbers of educts, products and inhibitors. Thus we have one subpage for every possible combination of these numbers (not shown in the paper).

Initially, we assume that all reactions run at the same *step width* of 1, increasing the simulation time after each occurrence by 1. This is achieved by setting the delay expression s' of the arc pointing to the place *state* of figure 1. It can be noticed, however, that the speed of certain reactions are rather robust whereas others change quite dramatically in case of slight enzyme or substrate concentration deviations. The latter enzymes – called *key enzymes* – constitute the target of control mechanisms in the organism that regulate the metabolic processes according to the actual situation. This observation is taken advantage of in our model by adapting the step width (i.e., the time increment s') to the current speed of the reaction. If, for example, its actual speed is very low then the next occurrence of this reaction may be delayed by more than the actually specified time unit (not exceeding a maximum value of, say, 12), and if the speed is very high then the actual delay expression may be decreased (with a minimum value of 1). To find out an appropriate function for the speed dependent adjustment of the step width of the individual reactions is a tricky task. We checked out several strategies, but we shall not discuss this problem further in the paper. The effect of applying this step adjustment function leads to an acceleration of the simulation because, in every time step, only a subset of the reactions have to be executed.

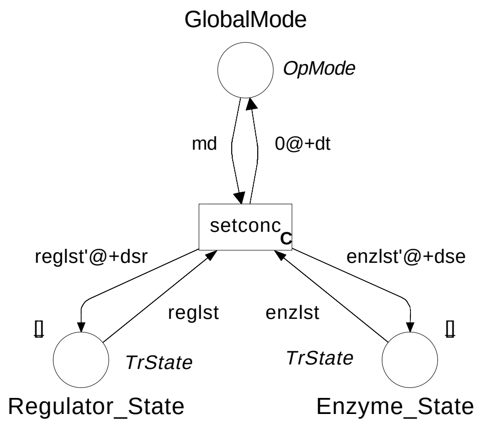
After it became evident that $\mathcal{A}$ constituted a sensible model that could be generalized to arbitrary metabolic pathways, a software package[4] was developed that automatically generates such a model from the data extracted from the databases. The resulting model can be executed immediately in the simulation mode. No much attention was paid to the graphical layout of the model: the reactions are simply arranged in lines until a page is filled, and then a new page is started. Afterwards, a final manual revision of the diagrams is advisable. The glycolysis model of figure 5 was constructed this way.

The simulation performance of the model $\mathcal{A}$ on a Power Macintosh G3 was not satisfactory (cf. the figures in section 5). Therefore, we built a new model $\mathcal{B}$ (not shown) by folding all places with the substrate concentrations into a single one, doing the same with the enzyme places, and connecting these two places to one single reaction transition. With a new simulator version of Design/CPN developed by the CPN group at Aarhus University, this would have lead to a better performance. As we did not intend to use this simulator at the moment, not surprisingly the performance got even worse: The calculation of enabling leads to a combinatorial explosion if the number of tokens in the places is very high. – Hence we looked for a more appropriate solution.

In the models $\mathcal{A}$ and $\mathcal{B}$, all substrate and enzyme places are always marked. This means that all reaction transitions are, in principle, always enabled. What changes are the second elements of the pairs in the substrate places, i.e., the concentrations. Even when an enzyme is transitorily de-activated – which was necessary for the experiments to be conducted –, this was achieved by setting its concentration r to zero; the corresponding transition in this case is disabled due to its guard $[r > 0.0, ...]$ (figure 1).

If we focus on those transitions which are not enabled permanently, we can omit all reaction transitions or replace them all by just one transition. The current substrate

[4]Standard ML was used as programming language because of its flexibility, its integration in Design/CPN and its use for the annotations in CPN models.



```
input (enzlst,reglst,md);
output (enzlst',dse,reglst',dsr,dt);
action
let
 val ((enl,dse),(rgl,dsr),dt) =
   if md=0 then        (* execute reactions *)
     (exec_step(ENZYM,enzlst),
      exec_step(ENVIR,reglst),0)
   else if md=(~1) then  (* initialize lists *)
     (init_list(!pEnz_List,md),
      init_list(!pReg_List,md),1)
   else (*md>0*)            (* update lists *)
     (upd_list(!pEnz_List,enzlst,md),
      upd_list(!pReg_List,reglst,md),1)
in (enl,dse,rgl,dsr,dt)
end;
```

Figure 6: The kernel page of the net model $\mathcal{C}$

concentrations can be stored in a particular *concentrations* record. This leads to the new model $\mathcal{C}$, in which there exists just one transition *setconc* comprising all reactions (figure 6). Model $\mathcal{C}$ is behaviourly equivalent to model $\mathcal{A}$. However, the intuitive pathway diagram (figure 5) is no longer needed, and the automatic construction of model $\mathcal{C}$ degenerates to the generation of the enzyme catalog and figure 6. This figure shall now be discussed briefly.

The colour set *TrState* consists of one list in which the necessary variables for the enzymes and the step width are encoded. To be more specific, this colour set is a list of pairs (time t, $sublist(t)$) whose first member, t, determines the simulation time at which those reactions included in the $sublist(t)$ have to occur next. Thus, at each simulation time t, only the $sublist(t)$ has to be inspected for the reactions to be executed. The members of each sublist are quadruples containing the enzyme name (characterizing the reaction), its concentration (constant), the last step width and the last speed of the reaction.

The place *Enzyme_State* stores all information for the proper execution of the enzymatic reactions through the function *R_kin*. As mentioned above, *R_kin* takes the metabolic constants of the enzyme from the enzyme catalog and the current concentrations of the substrates from the concentrations record.

The place *Regulator_State* deals with metabolic processes that are attributed to the environment of the modelled (glycolysis) pathway. Such reactions are necessary, e.g., to regulate particular ubiquitous molecules or to provide for steady sources and sinks of the entire pathway (cf. section 3). Normally this kind of reaction is not catalyzed by an enzyme, rather its speed is controlled by a (properly chosen) factor which – in this context – can be treated like an enzyme.

The transition *setconc* serves three purposes, distinguished by the value of *md*, the *global mode*.

1. At the beginning of the simulation ($md = -1$) the lists in *Enzyme_State* and *Regulator_State* are initialized via the function *init_list*.

2. Later, during the simulation ($md = 0$), the function *exec_step* is applied, which invokes *R_kin* for all enzymatic and environment reactions that are due at the current time, say tc, i.e., are a member of the $sublist(tc)$. These reactions are executed in a random sequence to take into account their inherent concurrency. After the execution of a reaction, yielding a new step width dsx, an updated entry for the reaction is inserted into the appropriate $sublist(tc + dsx)$. In this manner, the new lists *enzlst'* resp. *reglst'* are generated which, at the end of this process, replace the old lists in the places *Enzyme_State*

resp. *Regulator_State*.

Provision is taken in our metabolic net models that the set of enzymes (and hence of the enzymatic reactions) can be altered during the simulation, splitting the simulation into several so-called *simulation intervals*.[5] Prior to running the simulation, these intervals have to be specified. Each of them is characterized by one specific global mode value md with $md \in \{-1, 1, 2, 3, ...\}$. For each interval, the corresponding enzymes and substrates together with their initial (w.r.t. the interval) concentrations have to be specified in a particular mode file. Moreover, for every interval, a reaction speed factor sp and the model time te of its end has to be chosen in advance. A typical interval definition could read ($md = -1, sp = 0.2, te = 600$). In this case, every reaction speed dc computed by *R_kin* is multiplied by the factor sp. This finer granularity of the reactions is mainly needed to cope with substrate concentrations near zero to avoid meaningless negative concentrations.[6] As a result, if the total model time te equals 600 and $sp = 0.2$, then the total simulation time[7] would become $t_{real} = te/sp = 3000$.

3. Whereas the global mode value $md = -1$ is reserved for the initialization and $md = 0$ for the "normal" processing (see 1. and 2. above), modes with $md > 0$ are used to cope with a switch between intervals. To perform such a switch the function *upd_list* is applied. It updates the concentrations of the involved substances and initializes the lists in *Enzyme_State* and *Regulator_State* for the new interval to be encountered.

The entire net model $\mathcal{C}$ of the textbook glycolysis pathway consists of three pages. The most prominent one, modelling the metabolic processes, has been shown as figure 6. The other pages shall only be mentioned here.

One of them deals with the quite simple initialization of different values. The last page controls the management of the global mode values and the writing into the plot file from which – after the end of the simulation – a variety of plot diagrams can be constructed which depict the development of the concentrations (and of other values) as a function of time.

5 Simulation Results and Performance

The result of each simulation is usually represented as a graph that shows the concentration/time curves for the substrates involved. These diagrams are drawn by use of Microsoft Excel with some VBA macros. A typical example of such a plot diagram is figure 7 showing the concentration curves of the substrates participating – during a first interval of 1000 model time steps – in the (textbook) glycolysis, followed – in a second interval also of 1000 time steps – by the gluconeogenesis. As can be observed, towards the end of the entire simulation, each of the substrate concentrations converges, i.e., a flow equilibrium is reached.

For analysing the metabolic processes, also the temporal development of other values, e.g. of the reaction speed dc, may be of interest. This allows to identify, most often after a number of experimental steps, the cause of a deviation from the expected equilibria and to alter the responsible values. Such a series of experimental simulation runs can lead to identifying the key enzymes which most efficiently control and influence the entire pathway

[5]Obviously, it would make no sense to choose totally divergent sets of enzymes (and substrates) for the intervals. Rather the intervals should, taken together, constitute again a closed metabolic pathway.

[6]A negative concentration is the result of a too wide extrapolation, an "over-reaction". Anyhow, to be on the safe side, a resulting negative concentration is always cut off to zero.

[7]The simulation time must not be confused with the real (clock) time that a simulation run takes.

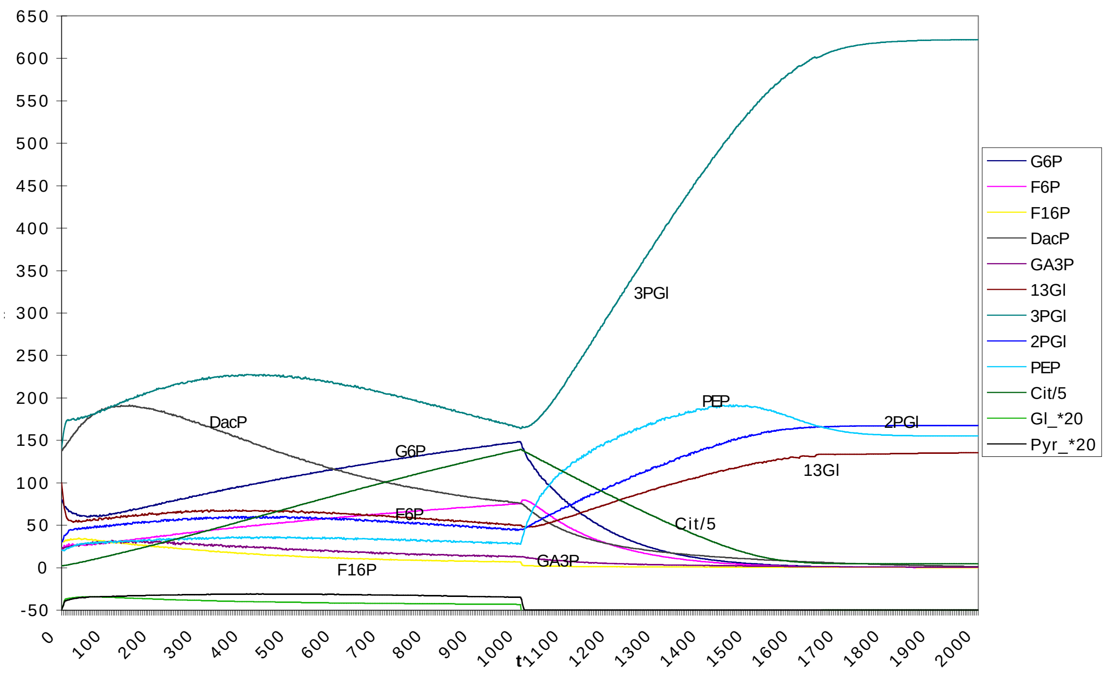


Figure 7: A concentration diagram for the glycolysis followed by gluconeogenesis

Pathway	Model	simulation time	number of steps	real time (sec)
Glycolysis	$\mathcal{A}$	6 051	45 669	1 127
	$\mathcal{C}$	6 050	6 174	70
Glycolysis, Glyc. & Gluconeogen., Glycolysis	$\mathcal{A}$	7 551	59 524	1 579
	$\mathcal{C}$	7 551	7 705	78
Glycolysis & Citric Acid Cycle	$\mathcal{A}$	5 051	42 351	1 070
	$\mathcal{C}$	5 050	5 154	69

Table 1: Performance figures for models $\mathcal{A}$ and $\mathcal{C}$

processes.

Before presenting performance figures of a few typical simulation runs, it should be emphasized that the simulations have been performed on a Power Macintosh G3 under OS 8 using Design/CPN 3.0.5. Hence, we did neither profit from more powerful computers nor from the improved versions of Design/CPN developed recently by the CPN group in Aarhus.

Table 1 originates in the simulation of three different pathways:

– (textbook) glycolysis,
– glycolysis, then combination of glycolysis and gluconeogenesis, then again glycolysis,
– combined glycolysis and citric acid cycle.

Each pathway has been simulated, under identical conditions, with the models $\mathcal{A}$ and $\mathcal{C}$.

As mentioned in the previous section 4, before a simulation run is started, among other parameters, the maximal *simulation time* has to be set. In principle, at every simulation time instance, several transition occurrences, i.e. *steps*, may happen.

In the model $\mathcal{A}$, each reaction is represented by one separate transition. Hence, the total number of steps is always much greater than the maximum simulation time. In model $\mathcal{C}$, however, one single transition stands for the set of all reactions. Thus the number of steps should be equal to the simulation time.[8]

A considerable part of the *real time* consumed for a simulation run is used for calculating the enabled transitions. As a consequence, the simulations with model $\mathcal{C}$ are much faster than with model $\mathcal{A}$ (factors 16.1, 20.2, 15.5, respectively). As can be seen from the table above, instead of about 20 minutes, a typical experiment with model $\mathcal{C}$ now takes a bit more than one minute.

6 Conclusions

The use of Petri nets to model and analyse metabolic pathways is promising. It renders intuitive diagrams and allows the automatic generation of the net models. The simulation speed – one of the crucial shortcomings as compared to differential equation models – can be substantially increased by choosing an appropriate modelling approach.

To summarize, applying the rules for equivalence transformations of high-level Petri nets implies a substantial flexibility in constructing net models that serve different intentions but behave equivalently. The models $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ represent the same metabolic processes, their execution renders identical results, although their graphical appearance and their performance differs strongly. The merits of model $\mathcal{A}$ lies in its graphical structure which directly reflects the connections among the biochemical reactions. This appeal of intuition is lost in the model $\mathcal{C}$ for the sake of a considerable performance improvement.

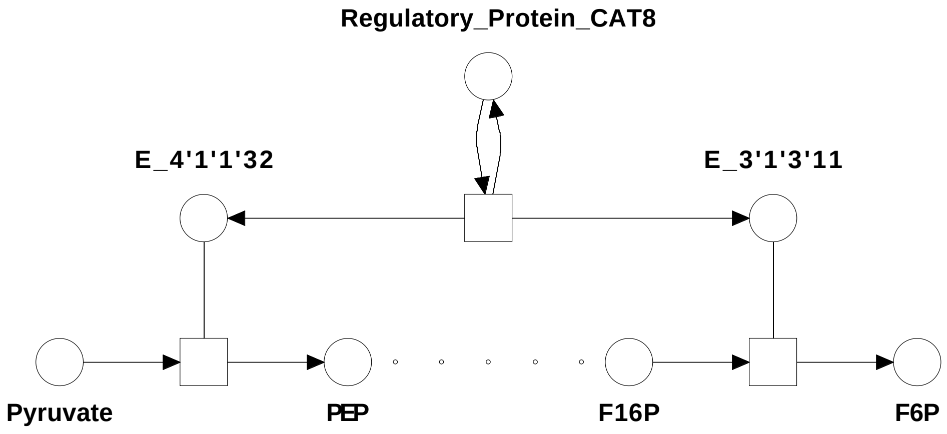


Figure 8: The model of a regulated pathway (gluconeogenesis)

In this paper we restrict ourselves to metabolic networks and leave out the so-called regulatory mechanisms[9] (enzymes directly activating or inactivating other enzymes by modifying them). The kinetic mechanism of regulatory enzymatic reactions is much less

[8]The number of steps is slightly greater because the transitions that are in charge of writing the actual values into the plot file have to be added.

[9]Regulatory mechanisms include
(1) Transcription Control: control of biosynthesis of enzyme proteins through regulator proteins,
(2) Interconversion: switching enzyme from active to inactive, and vice versa, through (de–)activating enzymes by signals of e.g. hormones via Second Messengers,
(3) Modulation by ligands, e.g., by coenzymes or diverse inhibitors.

understood compared to metabolic ones. Modelling regulatory enzyme relations as a Petri net would create no major difficulties. A sample is shown in figure 8 (cf. footnote 3).

Prospective future work in the area of metabolism is mainly determined by the needs and plans of the project in biochemistry we are collaborating with at GMD–SCAI. One research direction shall make use of annotated sequence databases and organism-specific databases to complement the metabolic information to get a more complete coverage of the functions coded in a genome. This means that we get regulatory proteins which can activate or de-activate certain enzymes of the pathway, leading to structures like the one sketched in figure 8.

Acknowledgements

We are indebted to Kurt Jensen for his suggestions concerning the enhancement of the model performance. We also would like to thank the five anonymous reviewers for their valuable remarks and hints.

References

[BRENDA] The Comprehensive Enzyme Information System. `http://www.brenda.uni-koeln.de:80/`. Cf. Schomburg, D., Salzmann, D., Stephan, D.: Enzyme Handbook, Classes 1-6, 1990-1995, Springer-Verlag

[Design/CPN] Design/CPN. `http://www.daimi.au.dk/designCPN/`

[ENZYME] Enzyme nomenclature database. `http://www.expasy.ch/enzyme/`. Cf. Bairoch, A.: The ENZYME data bank in 1999. Nucleic Acids Res. **27**(1), 1999, 310-311

[FrCa84] Franco, R., Canela, E.: Computer simulation of purine metabolism. Eur. J. Biochem. **144**, 1984, 305-315

[KEGG] Kyoto encyclopedia of genes and genomes. `http://www.genome.ad.jp/kegg/`. Cf. Ogata, H. et al.: KEGG: Kyoto encyclopedia of genes and genomes. Nucleic Acids Res. **27**, 29-34

[LePa92] Lee, I-D., Palsson, B.O.: A Macintosh software package for simulation of human red blood cell metabolism. Computer Methods amd Programs in Biomedicine **38**, 1992, 195-226

[Hof94] Hofestädt, R.: A Petri Net Application of Metabolic Processes. Journal of System Analysis, Modelling and Simulation **16**, 1994, 113-122

[KZL00] Küffner, R., Zimmer, R., Lengauer, T.: Pathway Analysis in Metabolic Databases via Differential Metabolic Display (DMD). Submitted to "Bioinformatics 2000"

[MDNM00] Matsuno, H., Doi, A., Nagasaki, M., Miyano, S.: Hybrid Petri Net Representation of Gene Regulatory Network. *In* Proceedings of the Fifth Pacific Symposium on Biocomputing. World Scientific Press, 2000, 338-349

[RLM96] Reddy, V.N., Liebman, M.N., Mavrovouniotis, M.L.: Qualitative Analysis of Biochemical Reaction Systems. Comput. Biol. Med. **26**(1), 1996, 9-24

[RML93] Reddy, V.N., Mavrovouniotis, M.L., Liebman, M.N.: Petri Net Representation in Metabolic Pathways. *In* Hunter, L. et al. (eds.): Proc. First Intern. Conf. on Intelligent Systems for Molecular Biology, AAAI Press, Menlo Park, 1993, 328-336

[ScHo95] Schuster, R., Holzhütter, H-G.: Use of mathematical models for predicting the metabolic effect of large-scale enzyme activity alterations. Application to enzyme deficiencies of red blood cells. Eur. J. Biochem. **229**, 1995, 403-418

[Tom99] Tomita, M. et al.: E-CELL: software environment for whole-cell simulation. BIOINFORMATICS **15**(1), 1999, 72-84

Web Based Interfaces for Simulation of Coloured Petri Net Models

Bo Lindstrøm
Department of Computer Science, University of Aarhus
IT-Parken, Aabogade 34, DK-8200 Aarhus N, Denmark
E-mail: blind@daimi.au.dk

Abstract

There are many situations in which we can use models of systems to help make decisions about their operation. This paper describes an approach which allows users without knowledge of Coloured Petri Nets to control the simulation of Coloured Petri Net models and interpret the results obtained from simulations via web based interfaces. We describe the architecture design of facilities in a simulation tool for making it possible to simulate a Coloured Petri Net model via a web based interface. As a representative example we show how to implement the approach in the Design/CPN tool.

Keywords. Coloured Petri Nets, Web Interfaces, Design/CPN, CGI Scripts, HTML Forms, Batch Simulations.

1 Introduction

There are many situations in which we can use models of systems to help make decisions about their operation. In many cases discrete event models are the most appropriate computational engines for these decision support tools. Petri Nets, in general, and Coloured Petri Nets [4] (CPNs or CP-nets) in particular, are general modelling languages for creating discrete event system models of systems. Petri Nets support both analysis of the logical properties of systems and simulation so that logical properties and behaviour of systems can be examined [5]. Petri Nets can also be used to investigate the performance of systems [9]. While Petri Net tools, such as Design/CPN [1], offer powerful capabilities for verification [6] and performance analysis [8] of models, their complexity and the need for understanding Petri Net theory requires specially trained personnel.

The motivation behind this paper has been to put Petri Net technology in the hands of application users who are not experienced with Petri Nets. The development of CP-nets and their tools has progressed to an industrial strength modelling language that retains the theoretical foundation of the CP-net theory. Though, not much focus has been on integrating simulation models into real applications.

Until now, the same graphical user interface (GUI) is often used for all activities involved in creating, simulating and analysing CPN models. Figure 1 illustrates this approach. First, the GUI is used to create the CPN model. Then the same GUI is used for simulation activities, such as setting the initial state/conditions of the model, and afterwards, it is used for simulating the CPN model. Finally, the simulation output produced during simulations is often displayed using the same GUI.

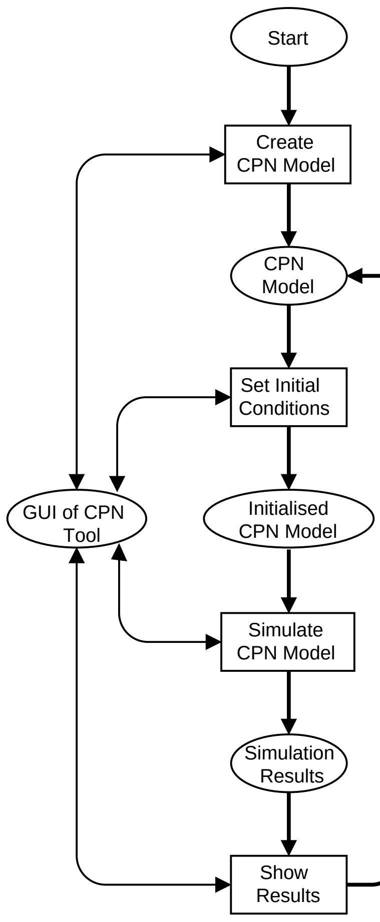


Figure 1: Original approach for creating and simulating CPN models.

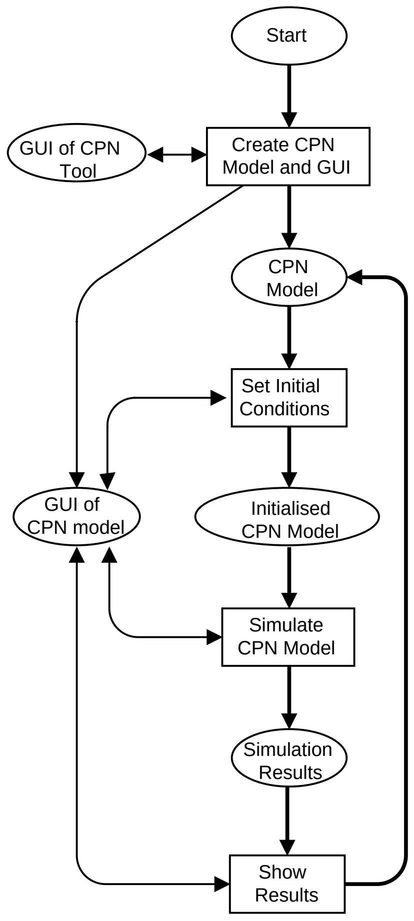


Figure 2: New approach for simulating CPN models via a web browser.

The architecture envisioned in this paper is to leave the creation of the CPN models to CPN experts, and let the application users simulate the models using another domain specific interface (see Fig. 2). First of all the CPN expert creates a CPN model together with a suitable GUI for the CPN model. Creating the CPN model and the GUI tailored to the CPN model allows application users without knowledge of the simulation tool to simulate the CPN model over a range of conditions (initial markings). Application users use the specially tailored GUI to set the initial state/conditions of the CPN model and then to run the simulation. Finally, the CPN model dependent GUI may be used to display the results of the simulation.

The architecture described above was expanded to provide the application user with remote access to the simulator of a CPN model using web technology such as so-called Common Gateway Interface scripts (CGI scripts) [3]. Thus, the application user has the advantage of having access to a rigorous discrete event system model of a complex distributed or concurrent system over a network using any web browser. The application user would control the executable model through a *Hyper Text Markup Language* (HTML) [12] form that provides inputs to the CPN model. The output would possibly be returned to the web browser of the application user. In this manner, the application user could perform analysis of the CPN model of the system without needing to understand or even see either the Petri Net models and Petri Net formalism nor the user interface of the Petri Net tool. Thus, the application user simply provides input via a form in a HTML document which is tailored to the CPN model, while the actual CPN model and the modelling tool are never seen by the application user. The remaining sections of this paper provide a detailed description of the design, implementation, and use of the approach outlined above. The Design/CPN tool will be used as a basis for the implementation of the approach.

The paper is organised as follows. Section 2 presents a realistic example which illustrates the approach for accessing simulators of discrete event systems via a web browser. This example will be used as a running example throughout the rest of the paper. Section 3 contains a discussion of the design considerations related to developing the approach. This includes both a general discussion not related to a specific tool, and a discussion of how to realise the design in the Design/CPN tool. Section 5 illustrates the amount of code a CPN expert using Design/CPN needs to write to apply the approach to a specific CPN model. Finally, Section 6 concludes the paper and gives suggestions for future work.

2 Example: Backup Company

This section gives an example of a realistic scenario, where a web based interface for simulating CPN models would be very useful. We will use the example presented in this section for illustration purposes in the rest of the paper. We will not describe the CPN model itself because it is not necessary to understand the CPN model in order to understand the approach of simulating a CPN model via a web browser. In later sections we will discuss how to modify the environment of a CPN model to allow being simulated via a web browser.

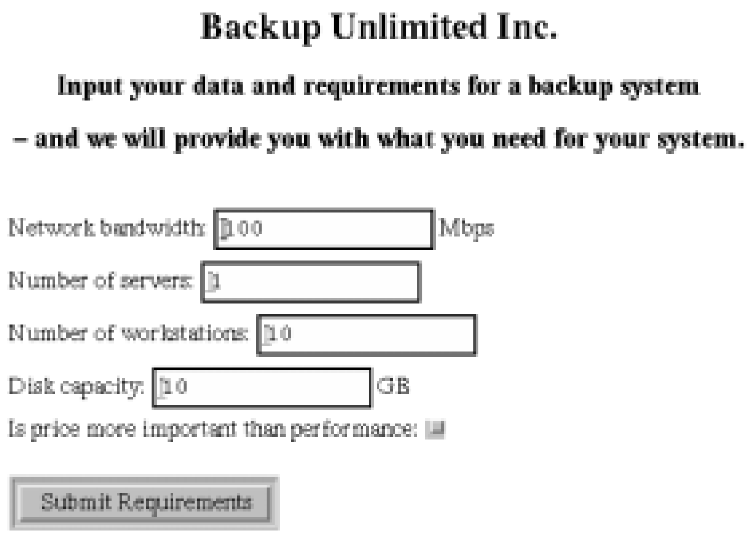
A company sells backup devices from a web site. When a customer needs to decide which backup devices to buy for a complex network environment, it is necessary to take several factors into account. The customer needs to consider his existing system in order to obtain the most suitable backup system. In particular he needs to consider the following factors.

- What is the network bandwidth on the local area network?
- How many machines do exist of each type: servers and workstations?
- How much disk capacity is present?
- Which is most important: price or performance?

Today many of the customers call the company and ask what components they should buy for their particular system. The company has created a CPN model that can provide a specification of the needed backup system given system requirements like the ones stated above. The CPN model helps the employees answering the phone calls to answer the questions of the customers.

It takes a lot of time for the company to answer questions from customers who ask what to buy for their specific system. Therefore, the company wants to provide their potential customers with a tool that can help decide which components to buy. An obvious solution would be to simply give the customers access to the CPN model from a web page. In this way both answering technical questions about what to buy and placing an actual order can be handled via the Internet. This means that customers do not have to call the company to figure out what to buy, thus reducing the production costs for the company.

After having applied the approach described in this paper to the CPN model, it is possible to simulate the CPN model via a web browser. A customer who wants to buy some backup devices accesses the web page containing an HTML form, like the one in Fig. 3, which is the interface to a simulator of the CPN model. The customer types the data of the existing system together with the requirements to the new system into the form.


Backup Unlimited Inc.

Input your data and requirements for a backup system

– and we will provide you with what you need for your system.

Network bandwidth: 100 Mbps

Number of servers: 1

Number of workstations: 10

Disk capacity: 10 GB

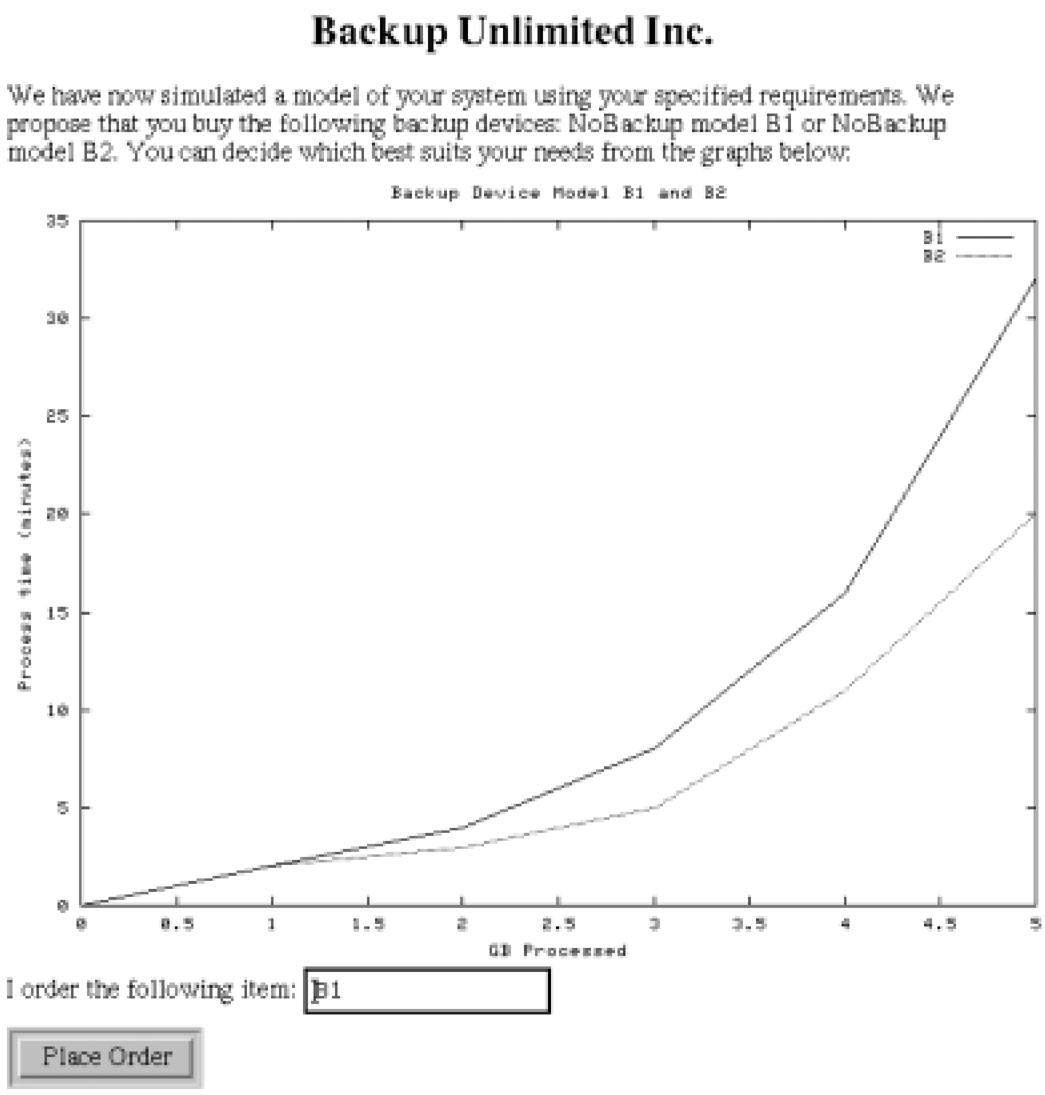
Is price more important than performance: ▨

Figure 3: HTML form as interface to the CPN model.

The system replies after having simulated the CPN model based on the input given in the form, see Fig. 4. Different kinds of output are produced. A textual description of the necessary backup components is given, and the expected performance of alternative components is illustrated using graphs.

The CPN model used in this example is a general model which can simulate any backup device the company sells. The model is fixed to a specific backup device by using different initial markings.

To determine which backup device is the most suitable for a customer, the CPN model is simulated several times with different initial markings – once for each backup device that the


Backup Unlimited Inc.

We have now simulated a model of your system using your specified requirements. We propose that you buy the following backup devices: NoBackup model B1 or NoBackup model B2. You can decide which best suits your needs from the graphs below:

I order the following item: B1

Figure 4: An HTML document containing the results of simulating the CPN model.

company produces. Based on the results obtained from the simulations, the most suitable ones are selected and displayed to the customer.

3 Design

The general description of the approach for creating web based interfaces for simulating CPN models introduced in Sect. 1 will be considered further in this section. The section describes the general design. Section 4 will focus on how to implement the design in the Design/CPN tool.

Several different options exist for executing and controlling a program via a web browser. In this paper we describe an approach for controlling simulations using so-called *Forms* and *Com-*

mon Gateway Interface scripts (CGI scripts) [3], which are both well-known Internet techniques.

Figure 5 illustrates the setup for using CGI scripts, while the message sequence chart in Fig. 6 illustrates a scenario where a CGI script is activated by submitting a form from a web browser. The scenario is as follows: a hypertext document (`backup_form.html`) is located on Computer B which is a web server (HTTP server B). The document could, e.g. be an HTML document like the one in Fig. 3. The application user using the web browser at Computer A downloads the HTML document. The application user then fills out and submits the form. When the application user submits the form, a link (URL) to a file on Computer C that holds the CGI script (`backup.cgi`) will be followed. (The CGI script may also be placed on the same computer as the HTML form.) This link is a "normal" HTTP link, but the file on the web server on Computer B is stored in such a way that the web server on Computer C can tell that the file contains a CGI script that is to be executed, rather than a document that is to be sent to the client as usual. The web server then executes the CGI script which can read the input that the user typed into the form. Based on the input the CGI script dynamically generates an HTML document. The HTML document is sent to the client while it is being generated as a stream. The web browser on the client computer displays the document while receiving the stream of HTML code from the web server, as it would display any other HTML document. The HTML document being received could, e.g. be like the one in Fig. 4.

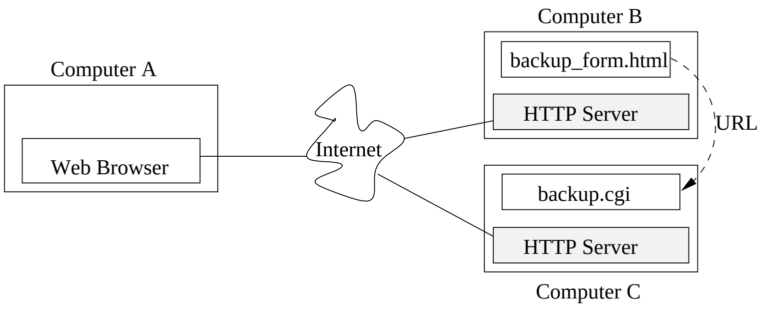


Figure 5: A web browser and two web servers.

Forms are very useful for specifying textual input via a web browser. Furthermore, they provide a simple mechanism for submitting the contents of the form to a CGI script. The HTML code for specifying a form is also very simple. Figure 7 shows most of the HTML code (it has been shortened where the `...` are placed) used to display the HTML document in Fig. 3. The URL `www.daimi.au.dk/cgi-sim/cpn.cgi` in line 2 identifies the CGI script to be activated when the user submits the form. The button for submitting the form is created using line 9 in Fig. 7. Lines 6 and 8 specify that two fields for input should be created. The input fields are named uniquely in the form using names (`name=disk` and `name=price`). These names are used by the CGI script to access the values of the fields when the form is submitted.

Note that only a few lines of HTML code are necessary to create a form, thus most people who have learned to create CPN models and do some basic programming should be able to learn to write such HTML code without too much difficulty. Though, in the future it will be possible to generate most of this code automatically.

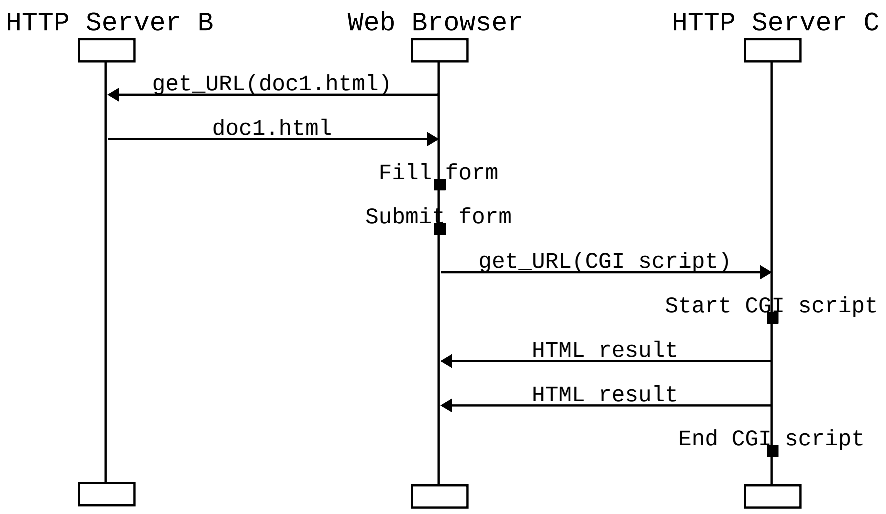


Figure 6: A web browser requests a CGI script to be executed.

```
<HTML>
   <FORM method=GET
         action="http://www.daimi.au.dk/cgi-sim/cpn.cgi">
       ...
     Disk capacity:
       <INPUT size=15 type=text name=disk value=10> GB <BR>
     Is price more important than performance:
       <INPUT size=15 type=checkbox name=price><BR><BR>
     <INPUT type=submit value="   Submit Requirements  ">
   </FORM>
</HTML>
```

Figure 7: HTML code for creating a form.

The program or CGI script that reads the information submitted in the form and processes it is more complex. Specialised scripts are required to handle the incoming form information. CGI scripts may be written in scripting languages like Perl [13] or in programming languages like C and Standard ML (SML) [10]. In general, a CGI script can be considered as an executable program that can be executed on a web server on request from a web browser.

To be able to simulate CPN models using CGI scripts in a controllable manner there are some requirements to the Petri Net tool. First of all, the Petri Net tool should be able to start a simulation of a specific CPN model without requiring a user to interact with the Petri Net tool via a GUI. For example, it should not be necessary to use dialog boxes to start a simulation – it should be possible to automate everything in a script. The tool should allow users to write user-defined functions, e.g. for retrieving input, saving files, and printing to standard output. The reason for these requirements is that because a CGI script is invoked via a form, the CGI script must be able to control everything related to reading input from the form, setting the initial state, starting the simulation, and producing results.

To make the CGI script as user-configurable as possible with respect to controlling a simulation, the CGI script must be able to control the simulation tool in the following way. When the CGI script is executed from a web browser, it should automatically execute a sequence of commands. In the following we will refer to this sequence of commands as a batch script [7]. A batch script can be considered as a simulation control script with the purpose of specifying exactly what the CGI script is intended to do – including when to start a simulation. Figure 8 illustrates the architecture of a CGI script using a batch script to control the simulator. In this context, a CGI script will be defined as an executable CPN simulator together with a batch script which defines what happens when the CGI script is executed.

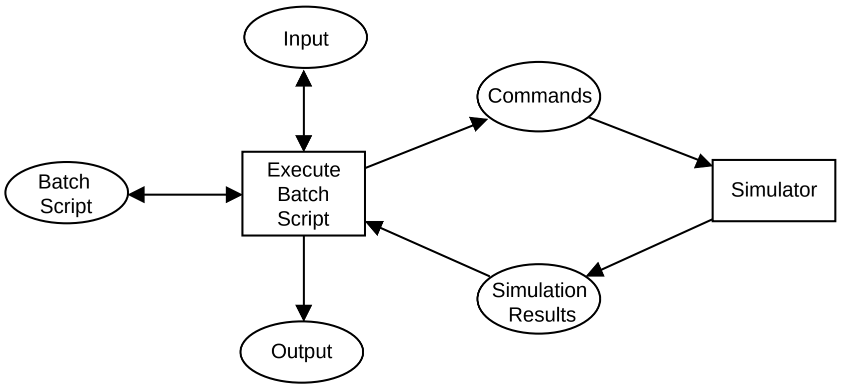


Figure 8: The architecture of a CGI script for simulating CPN models.

To give the user the largest possible freedom for defining CGI scripts the batch script needs to be specified by the user. In the context of CGI scripts the batch script typically has a certain structure which is likely to include the following actions:

1. Retrieve parameters from the form (see Fig. 7 for example of HTML code for a form).

2. Calculate markings to be used to initialise the state of the CPN model.

3. Initialise the state of the simulator.

4. Run simulation – and collect data.

5. Save results and/or send HTML code to the client web browser.

6. Goto 2 if the batch script is not finished.

7. Quit the CGI script.

First of all, the batch script needs to retrieve the parameters that are sent to the CGI script from the form. After having extracted the data from the input fields in the form, the batch script needs to specify how to run the simulation(s). The batch script often starts by calculating the markings for the initial state. The calculation is likely to use input parameters from the form or results from previous simulations. Now the state of the simulator is ready for starting the simulation.

While the simulation runs, data is often collected and saved in files for later processing. Therefore, the model often needs to be instrumented to collect the needed data. When a simulation has finished the results collected during simulations may be sent directly to the web browser (by printing HTML code to standard output). In addition results from the simulation may be saved to a file for later post processing. Finally, the batch script may decide that yet another simulation is to be performed. Therefore, the batch script may continue by restarting the script, otherwise the CGI script terminates.

Simulation tools that support converting a simulator into a CGI script may also include some auxiliary and high-level facilities. Examples of such high-level facilities are facilities for creating graphs to be saved in files, and printing HTML code for referring to graphs in the HTML document being generated by the CGI script. Then the web browser will download the image containing the graph while displaying the HTML document. Such an image `result.graph.png` was included in the HTML document in Fig. 4 using HTML code like line 5 in Fig. 9

```
<HTML>                                                  1
       ...
    <H1>Backup Unlimited Inc.</H1>                      3
       ...
    <IMG SRC="http://www.daimi.au.dk/result.graph.png"> 5
       ...
</HTML>                                                 7
```

Figure 9: HTML code for creating a form.

If the simulation tool does not contain the needed post processing facilities itself, it may possibly use external programs for doing the post processing of the data. There are only two requirements of the post processing tool to be used from the CGI script. The first one is that the tool should support being executed from the command line using a script containing all the needed commands to create the graph. Secondly, the tool should be able to save the output to files. Gnuplot [2] is an example of a tool that can be used for generating plots and graphs, and it supports command line scripts, too.

Given a Petri Net tool that fulfils the above mentioned requirements by allowing to create batch scripts and then be remotely controlled, it is be possible to create CGI scripts that fulfil many needs for simulation of CPN models from web browsers. In total the CPN expert needs to create two different files: an HTML form (like Fig. 7) to be used for specifying input to the CGI script, and the batch script for retrieving input from the form, running the simulation, and for producing results to be displayed at the web browser.

In the approach described in this paper, all input from the application user should be ready before submitting the form. In this way it is not possible to control the simulation after it is started – the simulation will be non-interactive but results can be shown gradually. In Sect. 6 we discuss future work which addresses this issue

4 Implementation in Design/CPN

Design/CPN [1] is a widely used tool supporting editing, simulation and verification of CPN models. In this paper, Design/CPN is used to prove the usefulness of the CGI concept for simulating CPN models via a web browser. In this section we describe some of the design considerations and facilities that are implementation specific for Design/CPN.

According to the description in Sect. 3, a CGI script is nothing more than a program that can be executed from a web page and then dynamically produces a new web page. A few modifications are made to Design/CPN in order to make it possible to turn the simulator of Design/CPN into a CGI script, or rather to drive the Design/CPN simulator from a CGI script. To understand why these changes need to be done, we need to describe some of the architecture of the Design/CPN tool.

Design/CPN is divided into two parts: one part implementing the GUI, and another part containing a simulation engine for simulating a CPN model. When a CPN expert has created a CPN model using the editor (see Fig. 10), simulator code can be generated containing the simulation engine and some model dependent code. This code contains everything needed to simulate the CPN model. Figure 10 differs a little from Fig. 1 in the sense that a simulator is generated from the CPN model before being able to do the actual simulation.

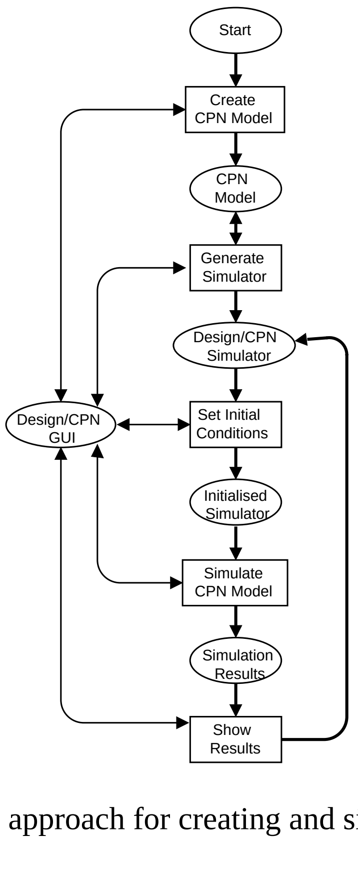


Figure 10: Design/CPN approach for creating and simulating CPN models.

In the context of CGI scripts only the simulator of Design/CPN is of interest. The reason is that it contains the entire executable simulator which is generated from the CPN model created in the editor of Design/CPN. Thus, the Design/CPN simulator is the only part of Design/CPN that need to be used when creating a CGI script.

4.1 Disabling the GUI of Design/CPN

Due to the fact that the simulator and the GUI of Design/CPN communicate with each other, the SML functions contained in the simulator for updating the graphical user interface (GUI) of Design/CPN should be modified to not update the GUI while simulating. The reason for this modification is that the GUI of Design/CPN is not present in a CGI script. Only the code constituting the simulator and user-defined functions is contained in a CGI script. Remember that a CGI script is invoked from a form in a browser, while the CGI script itself is running on the web server. Therefore, the web page shown by the web browser can be considered to be the GUI of the CGI script.

4.2 Creating CGI Scripts

Another modification of Design/CPN makes it possible to save a CGI script in an executable file. The executable file will contain the entire simulator code for the CPN model and some user-defined functions. The simulator code is model dependent SML code which is automatically generated by Design/CPN. This code makes it possible to simulate the CPN model. The user-defined function will be the batch script, as described in Sect. 3. Batch scripts in Design/CPN are written in SML.

An example of a batch script can be found in Fig. 11. The function getValOfField reads the value that the user has input in the form in the field named by `disk` which is the name of the field with the title “Disk Capacity” in Fig. 3. The user may want to do some calculations (calculate_initial_marking) before initialising the state of the simulator by means of the function init_state. The simulation is started using the function simulate. The function save_results is supposed to save results in files and/or send them to the web browser.

When the user has finished creating the batch script to be included in the CGI script, the final CGI script can be generated. The CGI script is generated and saved in a file by simply invoking a SML function. The overall approach for creating a CGI script using Design/CPN is illustrated in Fig. 12. The model is created and the simulator is generated using Design/CPN. Then a batch script is created by the CPN expert to control the actions of the CGI script. Finally, the CGI script containing the batch script and the Design/CPN simulator is generated and is then saved in a file.

4.3 High-level Functions

Due to the fact that SML is the language used for specifying CGI scripts which contain the Design/CPN simulator; or in particular that the batch script contained in the CGI script is written

```
fun batch_script (_, _) =
    let fun parse_form_input () = getValOfField "disk";
        fun cycle_script () =
            (calculate_initial_marking();
             init_state(); (* Initialise state of the simulator *)
             simulate(); (* Run the simulation *)
             save_results();
             if not_finished() then cycle_script ()
             else ());
    in
        (parse_form_input ();
         cycle_script ())
    end
```

Figure 11: A simple batch script.

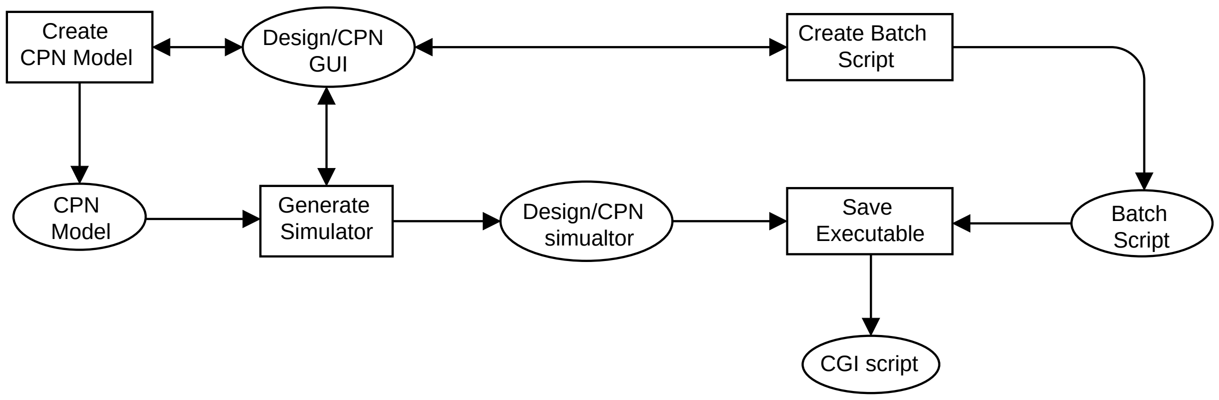


Figure 12: Creating a CGI script from Design/CPN.

in SML, the CPN expert can use the full power of the language SML for getting input and producing output from the CGI script. This section describes some auxiliary and high-level functions that may be useful when creating batch scripts to be used in CGI scripts. In particular we will focus on functions for reading input from HTML forms and generating HTML code as simulation output.

We will now describe how to read the parameters entered in the browser from the CGI script. When discussing the general structure of a batch script in Fig. 11, we introduced the SML function getValOfField. This function is very useful when reading a value that an application user has input in a form. The function can be used to read any input field in a form, thus it is very general. The only parameter given to the function is the name of the field in the form. The function retrieves the input sent from the form and returns the value contained in the field.

When we want to produce an HTML document as output from running a CGI script, the CGI script needs to print some HTML code to standard output. SML provides the function print for printing to standard output. By printing HTML code to standard output it is possible to create complex web pages. The HTML document may even include embedded graphics, so-called Java scripts [11], and Java applets which allow programs to be executed directly on the client holding the web browser. Figure 13 illustrates a simple SML function which prints some simple HTML code.

```
fun print_header n =
    (print ("<B><FONT SIZE=+2>Results of running"^
            (Int.toString n)^
            " simulations</FONT></B>"));
```

Figure 13: Print HTML code to a web browser.

It may also be useful to save HTML documents as files. This is particularly useful if several data files and figures are generated by the CGI script. In this way it is possible to divide the results of running the CGI script over several HTML documents. The page printed directly to the browser could simply be a kind of index page for the rest of the HTML documents. In this way the results of executing the CGI script may be shown in a structured manner. SML contains several functions for creating files and directories. Thus, it is immediately possible to save several HTML documents, to be referred to from other HTML documents.

Results or raw data obtained during or after a simulation can also be saved in files. Collecting data can, e.g. be done using the Design/CPN Performance Tool described in [8]. By printing HTML code which provides a URL link to the data file (see Fig. 14) to the web browser, the user can download the result files when the CGI script (or simulation) ends. After downloading the file, the user can analyse the produced results using his favourite analysis tool. As an alternative to downloading the raw data, the CGI script can post process the data itself.

```
fun print_url (URL, description) =
    (print ("<A HREF=\""^URL^"\">"^description^"</A>"));

print_url ("www.daimi.au.dk/res1.txt", "Results from simulation");
```

Figure 14: Print URL to a web browser.

In Sect. 3 we discussed generating graphics using external programs directly from the CGI script. We said that Gnuplot [2] is a tool that can be controlled from a CGI script. To make it as easy as possible to create plots using Gnuplot, and because Gnuplot is a non-commercial product, we provide a SML function for plotting graphs using Gnuplot. Figure 15 contains the interface of the Gnuplot SML function. The function is simply called with a list of file names of the raw data files and some textual information to be included in the graph. Finally, the user also needs to specify a destination file name where the plot is supposed to be saved. This SML function implies that users not familiar with Gnuplot are also able to easily create plots using the tool. The plot in Fig. 4 was generated using this function.

```
    fun gen_gnuplots
            {filenamesxtitles : (string * string) list,
             title : string,
             xlabel : string,
             ylabel : string,
             dest_filename : string,
             gnuplot_path : string}
```

Figure 15: Interface to Gnuplot function.

5 An Example of a CGI Scripts

In this section we give an overview of how simple it is to create CGI scripts from Design/CPN. In particular we illustrate how a typical batch script to be included in a CGI script will look.

Below we include most of the SML code for creating the CGI script used in Sect. 2. The purpose of including the code is to give the reader an idea of the complexity and the amount of code to be written to create CGI scripts. It is not important to understand every detail of the code. Some of the functions which are not directly associated with CGI scripts and batch scripts are not defined here. Please note that the code is rather general and can easily be modified to be used for another CPN model – or even be generated automatically.

Figure 16 contains the SML function which is to be invoked when the CGI script is executed. First the function retrieves the input that an application user has entered in the form, and then updates the CPN model with the extracted data. The function update_model is not included here. Then 20 simulations are executed to investigate the CPN model using the input parameters just retrieved from the form. Finally, some output is generated and sent to the web browser.

```
fun batch_script (_, _)=
    ((* Retrieve parameters from form *)
     update_model(retrieve_input ());

     (* Run one simulation for each of 20 different backup devices *)
     run_simulations (1, 20);

     (* Generate output *)
     gen_output ());
```

Figure 16: Batch script.

The function retrieve_input for retrieving the data input by an application user in the HTML form is contained in Fig. 17. The contents of each field is retrieved using the predefined function getValOfField. The function is very simple, and in the future it may be possible to generate the code for the function automatically.

When all parameters are retrieved from the form, we define how to run the simulations. Figure 18 describes how to configure the CPN model, run the simulations, save results, and

```
(* Retrieve data from the form *)
fun retrieve_input () =
    {networkField      = getValOfField "network",
     serversField      = getValOfField "servers",
     workstationsField = getValOfField "workstations",
     diskField         = getValOfField "disk",
     priceField        = getValOfField "price"}
```

Figure 17: Retrieve input.

finally decide if further simulations are to be performed.

The function load_model_configuration_parameters is not contained here. The purpose of this function is to load some configuration parameters into the CPN model. These parameters are supposed to specify initial markings which do not depend on the input from the form. In the context of the backup CPN model, these parameters would specify the configuration of the specific backup device to be simulated, e.g. capacity on tape, speed, etc.

After loading configuration parameters, the state of the CPN model can be initialised using the function init_state. To collect the needed results from the simulation, the CPN expert may have defined some functions for collecting data. We assume that this is done using a function named createDataCollectors. Now the CPN model is ready to be simulated. The simulation is executed using the function simulate. When the simulation ends we can examine the state, and observe if the results obtained suits the needs that the application user has initially requested using the form. We assume that the function do_results_suit_user_needs takes care of that. Finally we call the function run_simulations recursively for possibly running yet another simulation.

```
(* Specification of how to run simulations *)
fun run_simulations (i, (n:int)) =
    if (i <= n) then
        (load_model_configuration_parameters i;
         init_state();
         createDataCollectors i;
         simulate(); (* Simulate the CPN model *)
         if (do_results_suit_user_needs ()) then
             remember_model_no i
         else ();
         run_simulations (i+1, n))
    else ();
```

Figure 18: Run simulations.

Figure 19 contains the function gen_output which produces the HTML document to be displayed at the web browser. The output includes both textual information and a graph. The graph is plotted using the function in Fig. 20. The graph is saved in a file with a unique file name, and appropriate HTML code referring to the graph is printed to standard output. In this way the web browser receiving the HTML code from the CGI script automatically downloads and displays the graph.

```
fun gen_output () =
    ((* Print HTML directly to web browser *)
     print "Content-type: text/html\n\n"; (* CGI-header *)
     print "<HTML><BODY BGCOLOR=#FFFFFF>";
     print ("<FORM method=GET action=\"http://www.daimi.au.dk/"^
            "cgi-sim/place_order.cgi\">");
     print "<CENTER>";
     print "<H1>Backup Unlimited Inc.</H1>";
     print "</CENTER>";

     (* Print the results of running the simulations *)
     print ("We have now simulated a model of your system using your specified "^
            "requirements. We propose that you buy the following backup devices:");
     print (model_alternatives (!alternatives));
     print "You can decide which best suits your needs from the graphs below:<BR>";
     print "<CENTER>";
     (* Generate graphics using Gnuplot *)
     print_graphics (data_files());
     print "</CENTER>";
     print "I order the following item: <input size=15 type=text name=item_no><BR>";
     print "<INPUT type=submit value=\"   Place Order  \"><BR><BR>";
     print "</FORM></BODY></HTML>");
```

Figure 19: Generate output similar to Fig. 4.

```
(* Create a plot using Gnuplot *)
fun print_graphics (datafilesxtitles) =
    let val unique_filename = get_unique_filename ();
    in (gen_gnuplots {filenamesxtitles = datafilesxtitles,
                      title= ("Backup Device Model "^
                              (model_alternatives (!alternatives))),
                      xlabel = "GB Processed",
                      ylabel = "Process time (minutes)",
                      dest_filename = unique_filename,
                      gnuplot_path = "/usr/local/bin/gnuplot"};
        print ("<IMG SRC=\"http://www.daimi.au.dk/"^unique_filename^"\" ><br>"))
    end;
```

Figure 20: Create a plot using Gnuplot.

6 Conclusion

In this paper we have described how a web interface to a simulator of Coloured Petri Net models can be designed. In particular we have illustrated how it has been done in the Design/CPN tool. The approach is based on giving CPN experts the ability to easily create a CGI script containing the entire simulator. The initial conditions of the simulator can be specified via an HTML form on a web page. The fact that the initial conditions of a simulation can be specified via a domain specific form, gives users without knowledge of Design/CPN the ability to use pre-constructed simulators for specific analysis purposes.

The paper has also illustrated that integrating batch scripts into a CGI script has some advantages. Batch scripts give the user the ability to run several simulations after having specified input for all the simulations. Thus we will be able to first specify input via a web page and then based on the input run several simulations. The fact that the input to the CGI script can be specified in a HTML form on a web page means that the interface to the simulator can be domain specific and configurable. The domain specific and user-relevant graphical interface to simulators makes simulations of CPN models interesting for non-CPN experts.

Future work may include investigating the ability to explore state spaces via a browser - again possibly with a user relevant web page as graphical interface. This will also make it possible for non-experts to use the power of state spaces for answering questions by querying the state space of a CPN model. This could be obtained using the occurrence graph tool [6] of Design/CPN via a CGI script. It will be immediately possible to explore state spaces from a web page using the approach described in this paper. In Design/CPN it is just a matter of saving a CGI script after generating code for the occurrence graph tool instead of the code for the simulator.

Another interesting area is interactive simulation control via Java applets [11] embedded in HTML documents. It will also be possible to implement a domain specific GUI giving interactive control of the simulator of Design/CPN using a Java applet. Java applets are small Java programs that are automatically downloaded from a web server when a user requests an HTML document referring to the Java applet. When the Java applet is downloaded it is automatically started within the browser using a Java interpreter on the client machine. Using Java applets the simulator on the web server and the Java applet on the client machine can communicate during a simulation. By using Java applets it is possible to obtain interactive simulations via the web browser. To be able to use Java applets there are some extra requirements for the simulation tool related to communication between the Java applet and the simulator. It requires TCP/IP communication between the applet in the browser and the simulator residing on a server.

Finally, future work may also include developing auxiliary functions for generating templates of code for HTML forms and template code for batch scripts. In particular template code for retrieving data from input fields in forms would be easy to generate automatically. The CPN expert could annotate the relevant variables in the CPN model and then the Coloured Petri Net tool could automatically generate a HTML form including fields for the annotated variables. Furthermore, it could also create functions for parsing the fields of the form and for setting the variables in the CPN model to the values entered in the form by the application user. Experiments will show whether creating such template code will be useful in practice. However, one thing that will be gained from generating both the HTML form and the functions for parsing the form is consistency between the HTML form and the CGI script, i.e. it will be possible to avoid some errors due to inconsistency between the names of the fields in the HTML form and the names referred to by functions for retrieving data from a form.

In conclusion, this paper has described a technique for making simulation of CPN models usable for people without interests in the technical details of CPN models. Thus, the paper has opened for using CPN simulators behind services on the Internet.

7 Acknowledgements

The ideas presented in this paper have been developed during a project conducted in cooperation with Søren Christensen, University of Aarhus, Lee W. Wagenhals, Insub Shin and Daesik Kim from George Mason University, Fairfax, VA, USA. We want to thank these people for valuable discussions during the development of the CGI approach. In particular, we want to thank Lee W. Wagenhals for his involvement in writing early versions of this paper. Finally, we also thank the CPN Group at University of Aarhus for valuable comments on this paper.

References

[1] Design/CPN Online
Online: http://www.daimi.au.dk/designCPN/.

[2] Gnuplot
Online: http://www.cs.dartmouth.edu/gnuplot_info.html.

[3] Shishir Gundavaram. *CGI Programming on the World Wide Web*. O'Reilly & Associates, Inc., 1996.

[4] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.

[5] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 2, Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.

[6] Kurt Jensen, Søren Christensen, and Lars M. Kristensen. *Design/CPN Occurrence Graph Manual*. Department of Computer Science, University of Aarhus, Denmark, 1996.
Online: http://www.daimi.au.dk/designCPN/man/.

[7] Bo Lindstrøm and Lisa M. Wells. Batch scripting facilities for Design/CPN. In K. Jensen, editor, *Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN, DAIMI PB–541*, pages 79–97. University of Aarhus, Department of Computer Science, 1999. Online: http://www.daimi.au.dk/CPnets/workshop99/.

[8] Bo Lindstrøm and Lisa M. Wells. *Design/CPN Performance Tool Manual*. Department of Computer Science, University of Aarhus, Denmark, 1999. Online: http://www.daimi.au.dk/designCPN/man/.

[9] Bo Lindstrøm and Lisa M. Wells. *Tool Support for Simulation Based Performance Analysis using Coloured Petri Nets*, 1999. Department of Computer Science, University of Aarhus, Denmark.

[10] Lawrence C. Paulson. *ML for the Working Programmer, 2nd edition*. Cambridge University Press, July 1996.

[11] Sun
Online: http://developer.java.sun.com/.

[12] W3C, http://www.w3.org/TR/html/. *Hyper Text Markup Language (HTML), W3C Recommendation*.

[13] Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl, 2nd edition*. O'Reilly & Associates, Inc., September 1996.

High-level Petri Nets for a Model of Organizational Decision Making

Sven Heitsch, Michael Köhler, Marcel Martens, and Daniel Moldt

University of Hamburg, Department for Computer Science
Vogt-Kölln-Str. 30, D-22527 Hamburg
{5heitsch, koehler, 5martens, moldt}@informatik.uni-hamburg.de

Abstract. This paper introduces a new direction in formalizing sociological models. Sociological theories are a field of application for computer science, hence sociologists describe a theory in informal ways. These theories are transformed into computational models which can be studied and investigated with formal methods. We have chosen to formalize a common model of sociological theory, the Garbage Can Model of Organizational Choice. This model refers to organizations as organized anarchies. Bounded rationality leads to ambiguous decision situations.
Colored Petri nets offer formal semantics, graphical representation, means to model concurrency, and immediate executability and, thus, seem to be well suited to present complex sociological dependencies. Our approach uses a special formalism of high-level Petri nets, called Reference nets, which is applicable to present the individual parts of the model.
By the executable Petri net model of the Garbage Can theory, central notions like concurrency, conflict, and confusion – known in the Petri net theory – could be directly expressed and presented to sociologists and lead to new insights to the sociological theory.

Keywords: Garbage Can Model of Organizational Choice, Net Instances, Object-oriented modeling, Reference Nets, Synchronous Channels

1 Introduction

"Socionics" as understood in this context stands for an interdisciplinary research field between sociology and computer science. Research aims at the question how models of the social environment can contribute to the development of intelligent computer technologies. The modern society delivers a rich reservoir of ideas for the modeling of multi agent systems. Vice versa sociology may benefit from computer science by using computer science techniques for the validation and discussion of their terms, models, and theories. Finally future applications of "hybrid communities" are a special challenge to both disciplines (see [Mal99]).

Finding a formal description for sociological theories is a problem. Sociologists formulate their theories usually only in a textual form. This does not allow the application of more formal methods to validate their theories. One of the main ideas presented here is to formalize the theories by building Petri net models. A model which can be executed by computers and validated by sociologists is desirable. This paper focuses on the sociological theory of the Garbage Can Model of Organization Choice by Cohen, March, and Olsen ([CMO72]). Here a Fortran program was used, later Masuch and LaPotin in [ML89] presented an implementation with a functional language. The main drawback is that these models can be interpreted by computers, but not by many sociologists. The Garbage Can Petri net model is a first step to develop a special Petri net formalism which allows sociologists to directly identify their concepts.

Besides notions of multi agent systems from Distributed Artificial Intelligence Petri net modeling seems to be a promising approach. Object-oriented concepts in the context of programming languages (e. g. [Lou93]) and the Unified Modeling Language (UML) ([JBR99]) combined with high-level Petri nets are the techniques used in this paper.

Petri nets provide concurrency, conflicts, confusions, active and passive views (see [Pet81], [Rei92]). Sociologists use similar terms to describe their models. Object-orientation is used for finding the overall structure because objects represent real-world phenomena. [Val96] introduced the notion of "nets in nets", [Mol96] presented a general approach using Object-oriented Colored Petri Nets. Adding net references and integrating concepts of object-oriented programming resulted in a high-level Petri net formalism called Reference Nets (see [Kum99]). Theory was put into practice by the implementation of Renew, the Reference Net Workshop (see [KW99]).

In this paper the application of Reference Nets to the Garbage Can Model of Organization Choice ([CMO72]) is featured. [CMO72] is a fundamental paradigm of behaviouristic organizational theory. The article has been and still is widely received among sociologists. An illustrative, but semantically precise model of the sociological theory which can easily be simulated is the socionical contribution of this work. An early version of the work in progress was presented in [HMM99].

In the scope of the socionics project at the University of Hamburg different formal approaches to sociology need to be considered. This paper focuses on the application of one selected formalism to one sociological theory. In section 2 the interconnection between Petri nets and object-orientation is described. Section 3 gives an insight to the Garbage Can Model of Organizational Choice of Cohen, March, and Olsen ([CMO72]). A reference net model for a generalized version of the theory is introduced in section 4. Then this version is extended to a larger model which captures all aspects of [CMO72] in section 5. Section 6 discusses the Petri net approach, concludes by evaluating the results, and provides ideas for future work.

2 Basic Notions

The basic concepts of object-orientation that are relevant for this paper are shortly introduced. Then the basics of Petri nets are explained. Both concepts are then integrated in the object-oriented Petri nets. Further extensions lead to Reference nets which are used as the modeling technique.

2.1 Object-Orientation and Petri Nets

Object-oriented analysis is the here chosen method for transforming sociological texts into a formal descriptions. Petri nets are used as the formal language. To bridge the gap between object-oriented methodology and Petri nets as description technique, concepts of the former are adopted in the latter.

Object-oriented analysis The notions of objects and classes of this paper follow closely the ideas of Louden in [Lou93]. Diagrams follow UML (see [JBR99]). In sociology it is not common to structure models or even theories in that way. Understanding the main ideas of the sociological model and capturing the notions in terms of objects and classes is one aim of this work. The structure of the Petri net models follows the ideas of encapsulation and partitioning common to object-orientation.

Petri Nets A detailed description of the historical development – given by Jensen – can be found in [Jen92]. The basic concepts are concurrency and conflicts, active and passive parts, and the movement of tokens. It is important to provide a visual technique to sociologists since there is a common rejection of formal notions. The few concepts of active (transitions) and passive (places) parts of a system with the restricted relation between them is straightforward and easy to understand.

A first Petri net version of the Garbage Can Model has been developed by Valk. It has been used in lectures of a sociology course at the sociology institute of the University of Hamburg. Sociology students had few problems to understand the model. It even turned out that the Petri net was faster to understand than a natural text, which was taught in previous years.

However, due to the very basic constructs of elementary Petri nets larger models become difficult to handle. This general phenomenon is not specific to sociology. Many high-level dialects of Petri nets have been developed in the last years.

Synchronous communication On the one hand objects can communicate asynchronously by message passing, on the other hand they can communicate synchronously. For example, in programming languages function calls can be used for synchronization and communication. Christensen and Hansen combined this mechanism in [CH94] with Petri nets by introducing typed communication through synchronous channels for Petri nets. Synchronous channels allow different transitions to be synchronized and exchange data.

Object-Oriented Petri Nets (OOPN) Besides the concepts of object-orientation (like object, class, inheritance, and association) the partitioning of large models into special parts has been tackled by computer science (see the work of Lakos in [Lak95], Sibertin-Blanc in [SB94] and Moldt in [Mol96]). Various approaches how to combine object-orientation and Petri nets have been made. Here the idea that an object is a net is followed. The net as an object has a specific interface which allows access to the methods and has an unique identity. Similar objects can be folded to classes. Thus, object nets can be seen as instances of net classes (or templates).

An object-oriented Petri net is a Petri net with a special net structure. For a more compact notation usually Coloured Petri nets are used (see [Jen92]). The structure ensures that a certain kind of interface of the object is implemented. Each method is represented by a transition (with some kind of inscription), each variable can be represented by a place with a certain color set. By the appropriate marking the state of the variable (and of the object) is represented. The identity is ensured by the net structure itself. When Colored Petri nets are used, these ordinary nets can be folded and then the color determines the right assignment of each net element. The tokens are related by appropriate kinds of tuples.

Object Petri Nets (OPN) As objects are instances of a certain net class each object has an unique identity. This identity can be referenced by other objects an arbitrary number of times. Thus, the reference to a net is the token of another net. Due to the reference semantics[1] the nets are objects which are located somewhere and the tokens are only references.

This leads to the question, whether nets could be regarded as tokens and to the topic of self reflexivity in the Petri net theory. The instrument of "nets as tokens in a net" of [Val98] defines system nets which provide the environment for object-nets to move and communicate. This idea is illustrated in fig. 1, where one can see a system net having an object net as its token. The object net behaves like a token, so if the transition t_1 fires it removes the object net from place s_1 and outputs two nets – one in place s_2 and one in s_3. This is quite different from the meaning of reference semantics – illustrated in fig. 2 – where two references to one single object net would have been placed in s_2 and s_3.

As discussed in different works (cf. [Val96] and [Köh99,Köh00]), we obtain problems, if we regard these tokens as a whole net with its own marking (as in fig. 1) and also, if we regard these tokens as references to the original net (as in fig. 2).

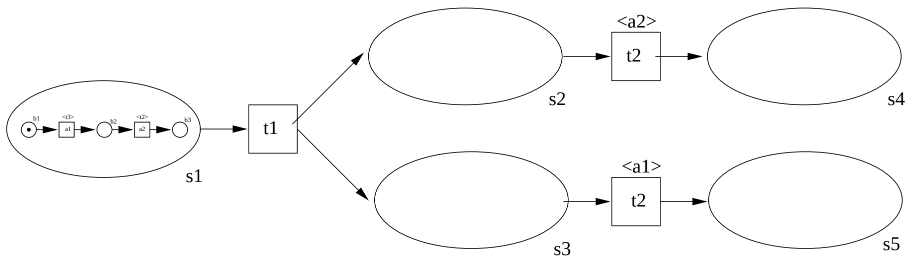


Fig. 1. An object net as a marking of the system net

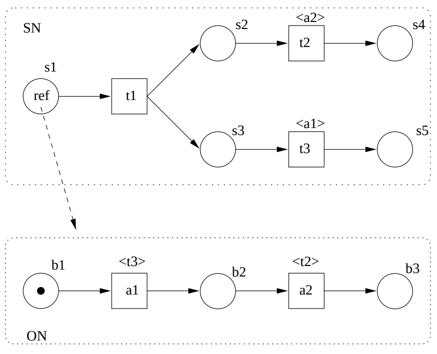


Fig. 2. An object net as a reference token

[1] Reference and value semantics for Petri nets should be understood analogous to the concepts in programming languages.

We use a conceptional notation – besides the concrete syntax of the Renew conventions – for synchronous channels. A transition t could be labeled with another transition, like $\langle a, b \rangle$. This means, that the transition t must fire synchronously either with transition a or with transition b. In our example the transition t_2 is labeled with $\langle a_2 \rangle$, so t_2 and a_2 must fire synchronously.

The benefit to use references to nets is efficiency of implementation and the well known behavior as used in usual programming languages.

From a formal point of view, the reference semantics destroys the Petri net paradigm of locality. As we have decided to use the reference semantics for our implementation of the garbage can problem, our design has to take care of the loss of the locality argument. The solution here is to use only one reference to a net to overcome this design problem. This is a special case of the approach in [Köh00].

Have a look at figure 2, where we have one object-net and one system net. In figure 3 one can see the a firing sequence, which is possible under the assumption of reference semantics.

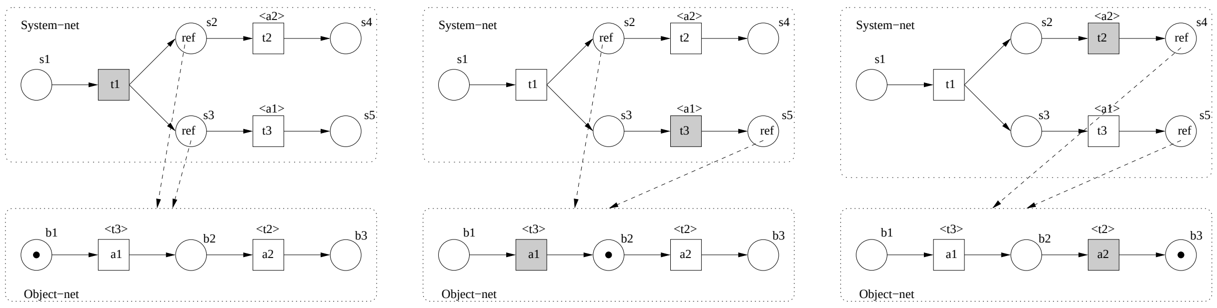


Fig. 3. Snapshots, reference semantics

You see, that our intuition to have two independent copies of our object net is defeated. Instead, firing of a transition changes the marking of a place somewhere else in the net.

If we try to fire this sequence again with a value-oriented semantics we run into trouble, as one can see in figure 4. In the situation of the right net in figure 4 no transition is activated, since the two copies of the object net are unrelated in their markings.

On the other hand, one can show, that a value oriented semantics – in such a simple way, as presented here – also has its disadvantages. To overcome these problems, Valk has introduced a "process semantics" for markings – an idea, which is not going to be discussed here.

2.2 Reference Nets

In this paper the formalism of reference nets is used which incorporates the concept of Valk in [Val98]. Reference nets – as implemented by the Renew tool – are a special high-level Petri net formalism that provide dynamic creation of net instances, references to other net

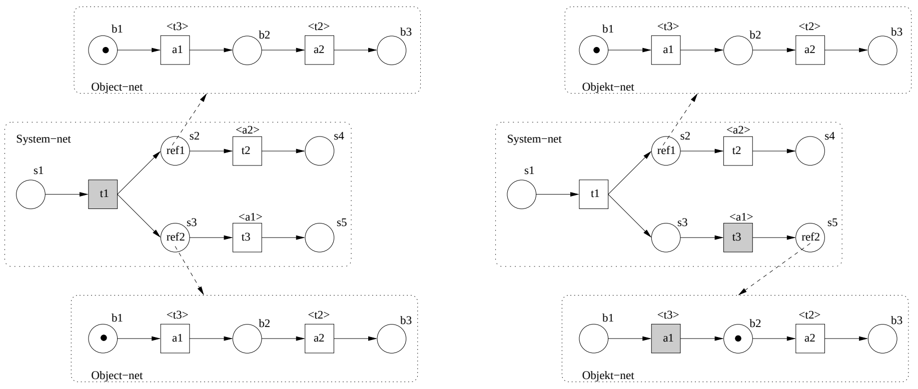


Fig. 4. Snapshots, value semantics

references as tokens, and communication via synchronous channels ([Kum98]). Java is used as the inscription language.

Basic elements Reference Nets (as Petri nets) consist of three types of elements: places, transitions, and arcs. Semantic inscriptions can be added to each net element. Places can have a place type and arbitrary number of initialization expressions. On creation of a net instance the initialization expression are evaluated and lead to the initial marking of the net. Arcs can have arc inscriptions. The arc inscriptions are evaluated when a transition fires and the results determine the consumption and creation of tokens. Transition may carry diverse inscriptions. There are expression inscriptions which are performed when the transitions fires. Guard inscriptions are preconditions to the transitions, i.e. the transition is only activated if all attached guard expressions evaluate to true. Action inscriptions start with the keyword action and are only evaluated when the transition fires. Creation inscriptions (consisting of a variable name, a colon, the reserved word new and the name of a class net) create new instances of nets. Synchronous channels synchronize two transitions which atomically fire at the same time.

Synchronous Channels Synchronous channels synchronize two transitions which both fire atomically at the same time. Both transitions must agree on the name of the channel and on a set of parameters before they can synchronize. This concept is generalized by allowing transitions in different net instances to synchronize.

The initiating transition must have a special inscription, a so-called downlink, which makes a request at a designated subordinate net. The syntax for a downlink consists of the name of the net reference, a colon (:), the name of the channel, and an optional list of arguments. The requested transition must have an uplink as an inscription which serves requests from every other net instance. Every time a synchronous channel is invoked, the channel expressions on both sides are evaluated and unified.

An uplink is specified as a transition inscription `:channelname(expr, expr,...)`. A downlink is specified as a transition inscription `netexpr:channelname(expr, expr,..)`.

This syntax illustrates the semantic difference between uplinks and downlinks because the invoked object net must be known before the actual synchronization can begin. A transition can have an arbitrary number of downlinks, but at most one uplink and a transition with an uplink can not fire without being requested explicitly by another transition with a matching downlink.

Net Instances and Net References When a net is constructed, it is a template without any marking that is used to create an arbitrary number of net instances during the simulation. These instances have got a marking, which can change over time.

A net instance is created by using the reserved word `new`, a variable and a net name in the following syntax `var:new netname`. This means that a new instance of the template `netname` is created and bound to the variable `var`.

Whenever a simulation is started, new net instances are created by transitions that carry the above mentioned creation inscriptions. For any further access on those new net instances now their references, which are tokens, are used.

The powerful formalism allows to model technical applications as well as business applications, especially workflow systems. In this paper the applicability to a sociological example is challenged. To find a reasonable starting point an example from organizational theory has been chosen.

2.3 Workflow concepts

To describe the control flow within net models capturing the organizational structure of a company the workflow nets of van der Aalst [Aal97] can be considered as a kind of standard. Within this contribution organizational aspects are of central interest, however, the main emphasis is not the economical point of view, but the sociological one. Even so the modeling requires to adequately present the control flow. Therefore, the structure of the nets is designed to fulfill the definition of Wil van der Aalst from a structural point of view. The extension to colored Petri nets and especially reference nets can briefly be described as providing no dynamic violation of the workflow net criteria, even if the structural design with respect to places, transitions, and arcs violates it. This is due to the net inscriptions.

Looking at the examples in the next chapter will reveal that the control flow could even be designed in a very rigorous way for the Garbage Can model. Nevertheless, the aspects of workflow are not discussed here any further. A more intensive discussion can be found in [MV00] and [AMVW99]. There the relations between object-orientation and nets as tokens and workflows are discussed. Furthermore, an architecture for a workflow engine is presented.

3 A Garbage Can Model of Organizational Choice

In sociology decisions are seen as one of the main outcome of organizations (see Luhmann in [Luh88]). This section introduces the Garbage Can Model of Organizational Choice by Cohen, March, and Olsen (see [CMO72]). Then a generalized version of the original work is presented. This will be the basis for the executable Reference net model of the following section.

The article of Cohen, March, and Olsen is a fundamental and often cited contribution to behaviouristic organization theory ([Imh99]). The model combines empirical characteristics, theory, and simulation aspects. It also deals with the essential sociological task how organizations can survive in an ambiguous and complex environment. The Garbage Can Model represents a criticism to common rational choice theories because decision making is seen as an ambiguous situation. It is argued that the behavior of at least parts of any organization can be described with this model.

The Garbage Can Model considers organizations as organized anarchies where decision situations are characterized by three general properties: problematic preferences, unclear technology, and fluid participation. It is argued that a decision is the outcome or interpretation of several relatively independent streams within an organization:

– A stream of problems: Problems are determined by inner and outer organizational circumstances and require attention of participants. Problems are looking for situations in which they might be raised.
– A stream of energy from participants: Participants come and go. It is assumed that they provide energy for organizational decision making.
– A stream of solutions: Participants of the organization produce solutions. Solutions move around, actively looking for questions to which they might provide an answer.
– A stream of choices: Choice opportunities represent the point of time when a decision is required by the organization. Each choice opportunity can be seen as a garbage can into which diverse problems and participants are dumped.

Organizations can be viewed as collections of choices, problems and participants. Participants and problems migrate between the different garbage cans. If a participant meets a choice under the right circumstances, a decision can be made. If there is at least one problem attached to the choice, the making of a decision leads to a rational outcome (decision by resolution), the problem is solved. Or the making of a decision takes too long and no problems are solved (decision by flight). If the decision is made so quickly that no problem has the chance to come up, it was made by oversight.

Masuch and LaPotin provide in [ML89] a metaphorical view on the basic Garbage Can processes of decision making:

> "... reconsider the finale of the James Bond movie 'A view to kill'. Agent 007 balances on the main cable of the Golden Gate Bridge, a woman in distress clinging to his arm, a blimp approaching for rescue. In terms of the Garbage Can Model, the blimp is a solution, Agent 007 a choice opportunity, and the woman a problem. In the picture's happy ending, the hero is finally picked up, together with the woman, and a solution by resolution takes place; the problem is solved. Now imagine numerous blimps, women, and heroes, all arriving out of the blue in random sequence. Heroes take their positions on the main cable. Women cling to heroes, blimps hover above the scene. Heroes may or may not be able to hold an unlimited number of women, but the blimps carrying capacity is limited; heroes with too many women cannot be

rescued. Blimps are retrieving rescuable, i.e., not-too-heavy, heroes. Women in distress are aware of that and switch heroes opportunistically, choosing the hero closest to retrieval. As women, as well as blimps, make their choices independently of each other, a light hero, on the verge of rescue, may suddenly find himself overburdened. Heavy heroes, in turn, may become rescuable all of a sudden as their women desert them."

This coming and going is the mechanism called fluid participation. Women may not be saved at all if they change between heroes disadvantageously and all of their heroes of choice turn out to be too heavy; then, these problems are not solved. Heroes may be saved when all women just have left; this is called a decision by flight. Also, heroes can be rescued before any distressed women was able to hold on to him; then, a decision by oversight has occurred.

Let's come back to the grounds of organizational theory and sum up the terminology: the bridge is an organization, heroes are choices, women are problems, and blimps are solutions. Choices attract problems and solutions. A choice is made if there is an appropriate solution to its problems. Three styles of decision making may appear, but only one of them solves problems.

One might wonder where the participants have gone. In this rather generalized version by Masuch and LaPotin in [ML89] participants do not appear. They remain backstage and have an indirect impact on the organization. They produce solutions and throw them into the scene. Because the participants are mentioned explicitly in the original Garbage Can Model by [CMO72], they will appear instead of solutions in the following Petri Net models for a better understanding of the decision making process.

4 The Executable Reference Net Model

The results of the above mentioned prototyping approach are three different Petri net versions of the Garbage Can Model.

1. The first version by Valk, regarding the metaphorical view of Masuch and LaPotin ([ML89]), is a non-executable Place/Transition net because it uses a special kind of arc which allows to remove all tokens of place atomically without knowing the exact number of tokens. This Petri net version is very generalized and reduced. It has been developed to give an impression how to apply Petri nets to sociological theories. It visualizes the essentials of the Garbage Can decision making processes and inspired the following models.
2. The second one is an executable reference Petri net which was created by folding the Place/Transition net of Valk in order to overcome the constrains of the first version. Colored Petri nets as a higher Petri net level are used. Still, this is a generalized version which illustrates more of the basic behavior of the Garbage Can model without taking care of all the details.
3. The third version is an executable reference net model (see [HM99]) consisting of eight large collaborating nets which captures all aspects of the original Garbage Can Model by

[CMO72]. It deals with several modeling problems which occured during the prototyping process. Presenting the whole model would certainly be far out of scope in this context.

Since the second version of the Garbage Can Model is presented to full extend in this paper, there will be two nets (Organization and Choice) picked out exemplarily and compared to the complex third version in order to discuss the prototyping approach and the modeling experience. In the following chapter it will be described, how structures of the nets could be reused in more complex models.

In this section the second version of the reference net model is presented. The results of an object-oriented analysis are transformed into object nets. The structure and interaction of these nets are described. Characteristics of the presented model are discussed.

An object-oriented analysis of the generalized Garbage Can Model – as described in the above-cited text by Masuch and LaPotin ([ML89]), but translated into the terms of the original model – leads to the identification of classes and associations (Fig. 5). In terms of Valk (in [Val98]), where Petri nets are used as token objects of other nets, a net which provides the environment and the control for the others is called system net. Tokens of the system net are simply named object nets. In this context there are one system net and three object nets. The methods of the objects are already listed in the class diagram, however, they are explained later in the context of the single classes.

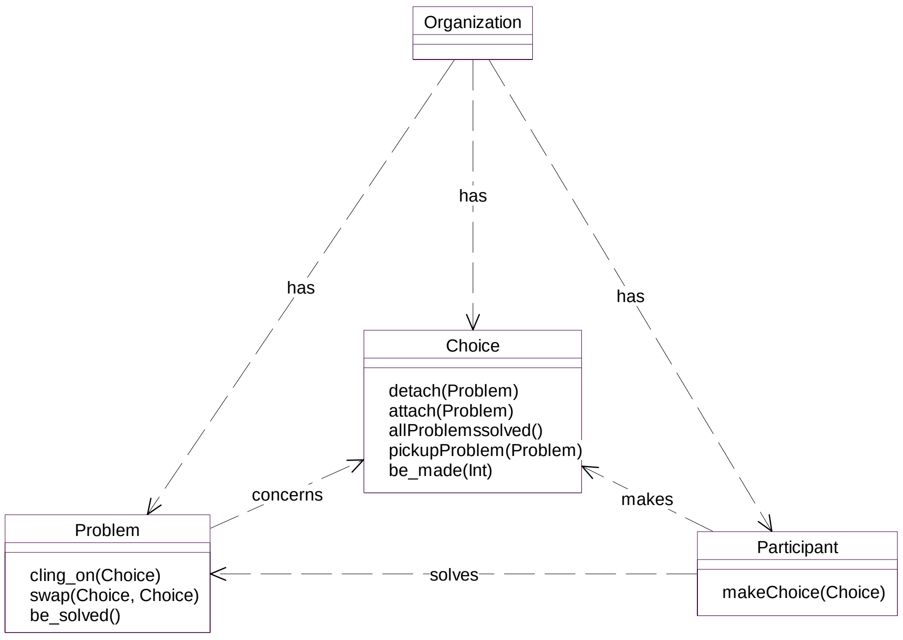


Fig. 5. Class diagram in UML notation

The Garbage Can Reference Net consists of five net classes:

1. the Organization which keeps track of the net instances involved and which controls the interactions among those instances,

2. the Choices which are the crucial elements of the decision making process and which represent the link between problems and participants,
3. the Storages, which are a special construct of the choices supporting the administration of the problems
4. the Problems which attach themselves to choices and are solved eventually, and
5. the Participants which bring relief to the distressed situation and lead to decision making.

The storage is intentionally not mentioned in the class diagram because it is just an auxiliary net for the choice and has nothing to do with the Garbage Can model.

There will be one instance of the net class Organization which is the system net (Fig. 6). The organization is responsible for instantiating and controlling all other objects involved.

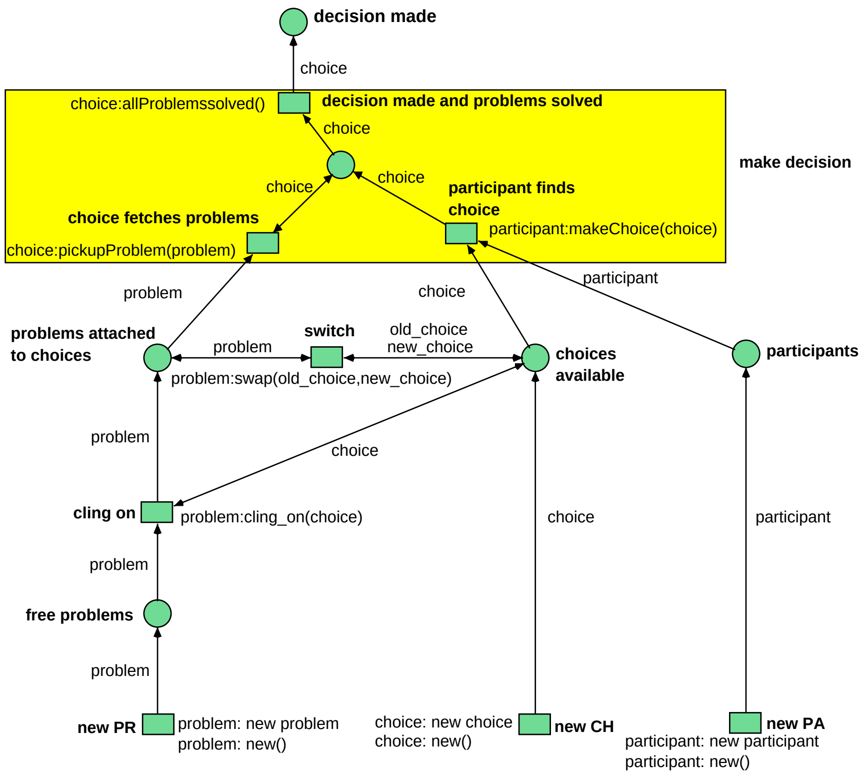


Fig. 6. System net Organization

Choices, problems, and participants are instantiated by calling their "new"-methods. Each transition of the system net calls methods of referenced object nets.

For example, when the transition "cling on" of the system net fires, one instance of problem (precisely, one reference to an instance of a class is meant) and one instance of choice are selected, and the problem's method "cling_on(choice)" is called.

The problem is put into the place "problems attached to choices" and the choice is returned to its previous place. Once again in different wording: transition "cling on" of the system net Organization is synchronized with transition "cling_on(choice)" of one instance of object net Problem. At the same time a synchronous channel called "cling_on" is agreed on by Organization and Problem. A reference to a choice is passed through this channel.

The transition "switch" tells the problem to leave its old choice and cling to a new one. One problem and two choices are referenced. The latter are passed to the problem net via the synchronous channel "swap". The large gray-shaded box in Fig. 6 can be seen as one large transition which represents the making of a decision. Due to characteristics of the Petri net formalism it has to be split into three single transitions ("participant finds choice", "choice fetches problem", and "decision made and problems solved"). When a participant has been generated and a choice is "available", the transition "participant finds choice" is enabled. After firing the choice moves on in order to solve all the problems attached to it. Eventually, the attached problems are being fetched, i.e. solved. After all problems of the choice have been removed, the decision making process can be completed (decision made and all problems solved).

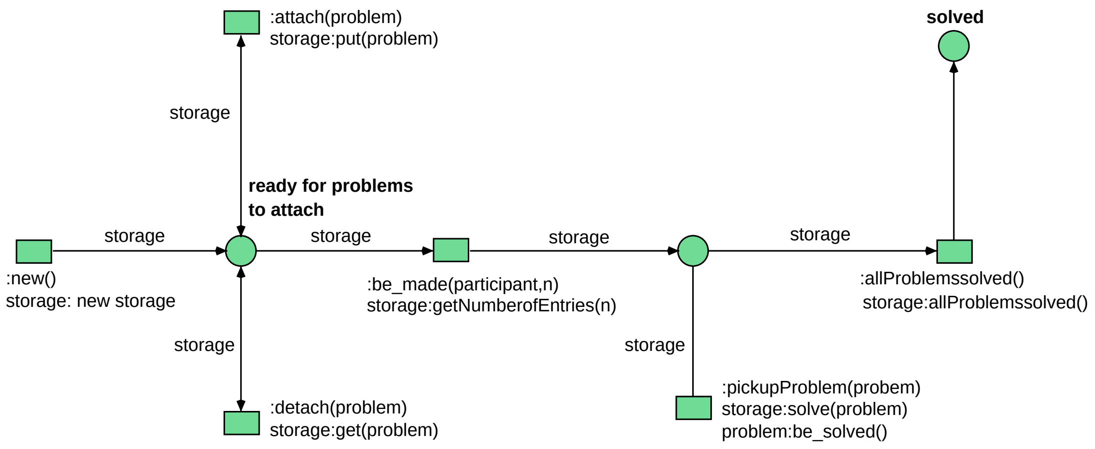


Fig. 7. Object net Choice

Each new instance of the net Choice (Fig. 7) also produces a new instance of the net Storage (Fig. 8) and the reference to this instance is returned and bounded to the variable storage. It is put in the place "ready for problems to attach" at first. Problems may cling to an instance of choice or switch between those. The choice is informed about these changes by the methods "attach" and "detach" and delivers this information to its storage via synchronous channels (put and get), where the amount of attached problems is saved. As soon as a decision has been made with the help of a participant (it has received the "be_made(n)" message where "n" represents the number of problems attached to the choice which is passed to the participant through the synchronous channel :be_made) it can neither be left by any of its problems nor can it be clung to by new problems. It is now in the state "decision made" and its "attach" and "detach"-methods cannot be called anymore. The choice's "pickupProblem(problem)"-method is called and returns the attached problems one by one. After all problems have been reported and "solved" by the storage, the transition "allProblemspickedup()" is enabled. It fires when the decision making process is complete.

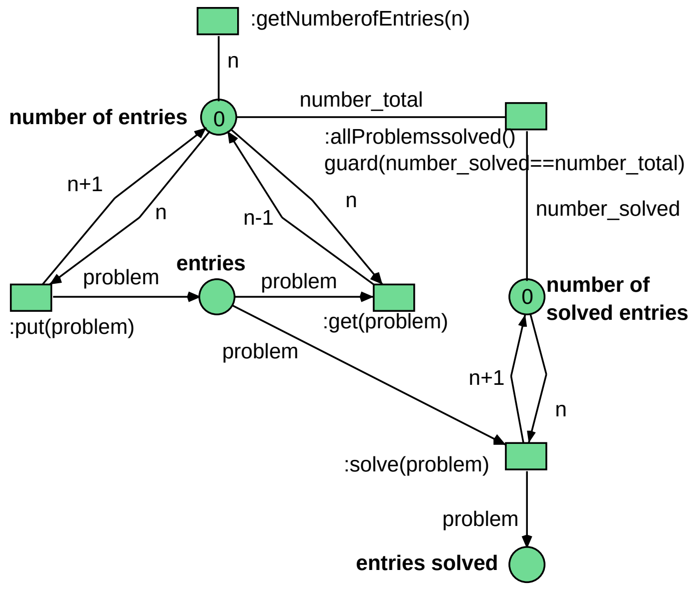


Fig. 8. Object net Storage

The object net Storage is a special construct to support the Choice with its administration of the attached problems. Basically it serves as a memory for each choice and stores the actual references of the attached problems ("entries") and the number of those references ("number of entries"). Furthermore the storage can move problems to the state entries solved and keeps track of this with a counter of saved problems. By calling the "getNumberofEntries(n)"-Method this number "n" of problems is returned. The Methods put(problem) and get(problem) add and remove entries and change the number of entries simultaneously. The "solve"-Method removes solved problems from the place "entries" and puts them into the place "entries solved" so that is not possible for solved problems to switch any more. The transition "allProblemssolved()" is enabled, when there are as many problems solved as there are stored.

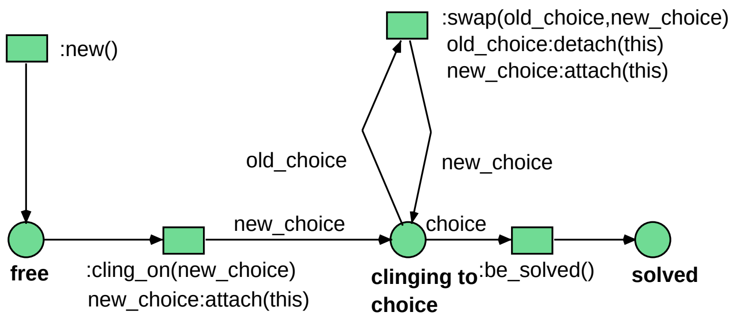


Fig. 9. Object net Problem

After the Problem net (Fig. 9)has been instantiated it will be "free" and looking for a choice at first. Then it can cling to a choice, swap between choices several times and will finally be solved.

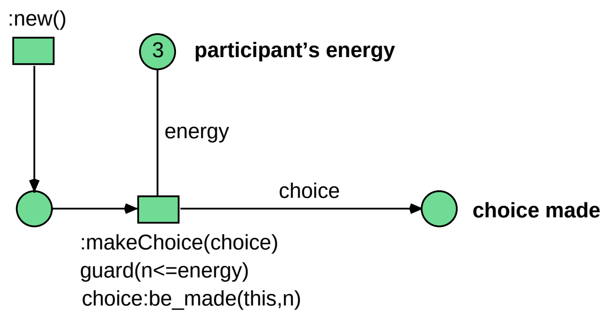


Fig. 10. Object net Participant

An instance of the Participant net (Fig. 10) approaches the scene and will eventually be relevant to one of the choices. If the choice's number of problems does not exceed the participant's energy, the choice can be made.

The Petri net model puts this version of the Garbage Can Model into a computer executable form. Each object of the sociological theory, except for the storage, can be found and analyzed separately. Also sociologists were satisfied with the visualization and readability of the results.

The basic phenomena of the original Garbage Can Model appear in this simulation. Especially the three decision styles can be observed. They can easily be visualized by firing sequences of the net during the simulation. Intentionally additional features do not appear in this version which should be seen as an illustrative introduction to the basic behavior of the Garbage Can Model.

5 Extensions of the reference net model

In this section the further development of the Garbage Can reference net model (see section 4) will be discussed and illustrated by comparing two nets from the generalized model with the corresponding two nets of the extended model (namely choice and choice (large), organization and organization (large)). Similarities and differences between the net structures will be examined.

The above mentioned reference net model has been extended in [HM99]. The result is a complex reference net model which captures the whole functionality of the Garbage Can Model of Organizational Choice [CMO72], which is far more complex than the metaphorical approach of Masuch and LaPotin ([ML89]).

The focus during the modeling process was to find basic and reusable structures. This approach was chosen in order to be able to add all the features of the original model without having to change the nets that has been developed so far.

The most important features, which refer to the problems and participants, are the strategy of finding the most attractive choice before clinging on it and the organizational structures which constrain the access to choices (see [CMO72]).

As mentioned in section 3 there is a stream of energy from the participants that is necessary to solve organizational problems. This energy is called "energy available". It characterizes the participants ability of solving problems. The more "energy available" a participant provides, the more skilled he is.

There is an opposite tendency in an organization, which behaves complementary towards the energy available. This is expressed in the "energy required", which characterizes the difficulty of the problems. The more "energy required" a problem has, the harder it is to solve it.

Choices attract participants and problems. So a choice receives both types of energy, the "energy available" and the "energy required". A decision can be made if there is as much "energy available" as "energy required". A decision can be made if there is at least as much "energy available" as "energy required". Finding the most attractive choice is done by calculating the difference between those energies: Only if the difference is zero or positive, a decision is made.

One can recognize this behavior in the net structure of Fig. 12 looking at the places which store the energies and the transitions which update and compare them. In [HM99] the afore mentioned organizational structures are also considered. These extensions, strategy and organizational structures, have been modeled by using additional reference nets (i.e. a net called oracle which has information about all object nets involved and which answers questions, or a net called multiset, which administers IDs, energies and checks the access to choices). Since not all details can be presented here, the focus is on the choice and the organization reference nets.

Moving from Fig. 6 to Fig. 11 there is not very much added to the net Organization (large). One can easily recognize the same structure of the net by looking at it. What has been changed is the number of participating objects which is limited following [CMO72], which means that there are ten participants (new PA), ten choices (new CH) and twenty problems (new PR).

Now every instantiation of new objects is reported to the above mentioned object net oracle. Before clinging to a choice or before swapping to another choice problems have to ask the object net oracle for the most attractive choice. Before spending their energy available, participants follow the same strategy as the problems and also ask the oracle for the most attractive choice. There are several more details in this model which can not be explained in the scope of this paper. Participants can spend their energy several times on different choices before they leave the organization by calling the "leave()"-method. The solution process of Fig. 11 is very similar to Fig. 6 if one compares the gray-shaded boxes ("make decision") and the inscriptions of the transitions involved. When all problems are solved there is a statistic module which analyses the style of decision making.

Comparing Fig. 7 and Fig. 12 again one can point out the same basic structure of the nets. The behavior is basically similar apart from additional places for the energies, the counters and some transition inscriptions.

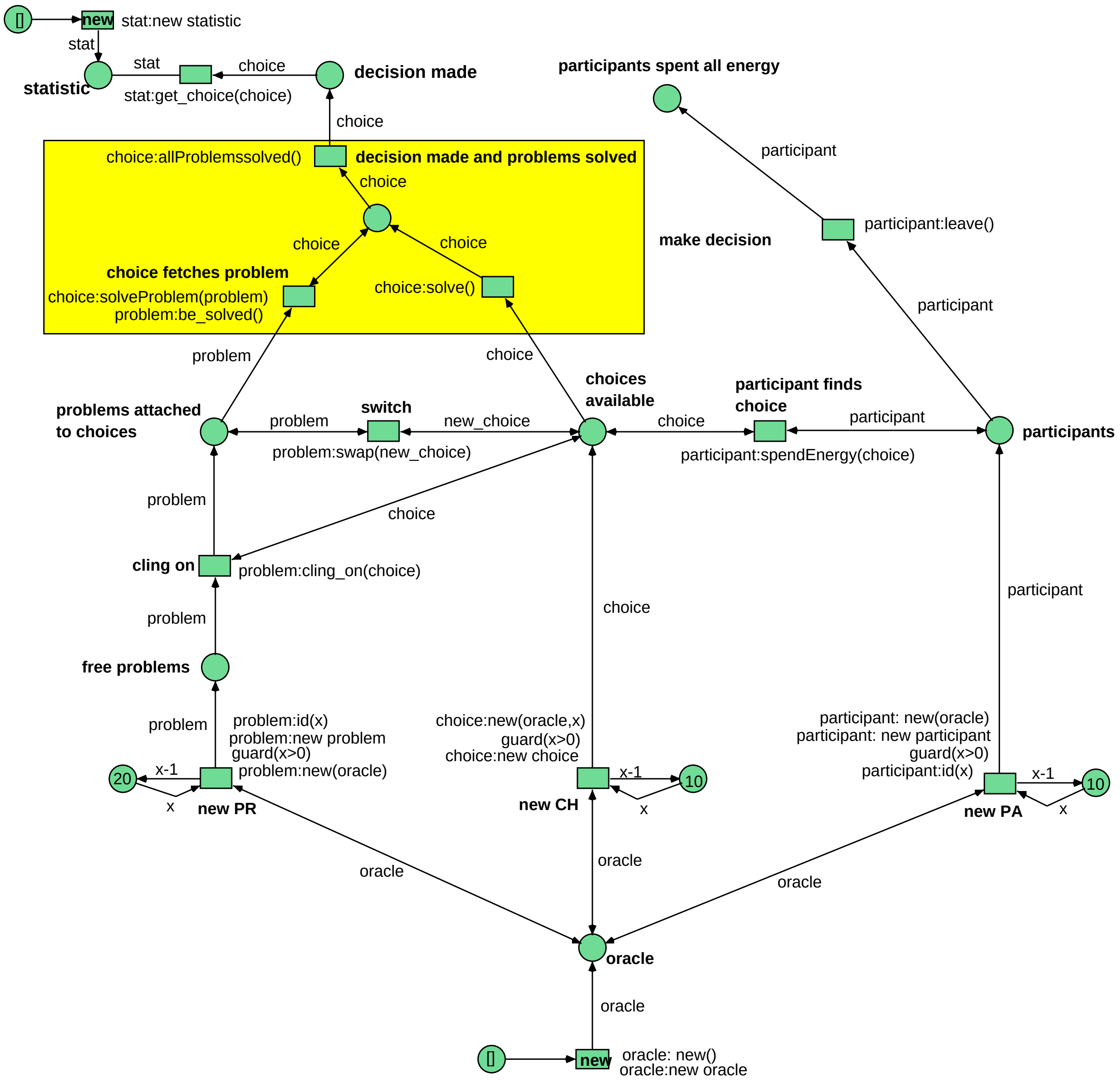


Fig. 11. Object net Organization

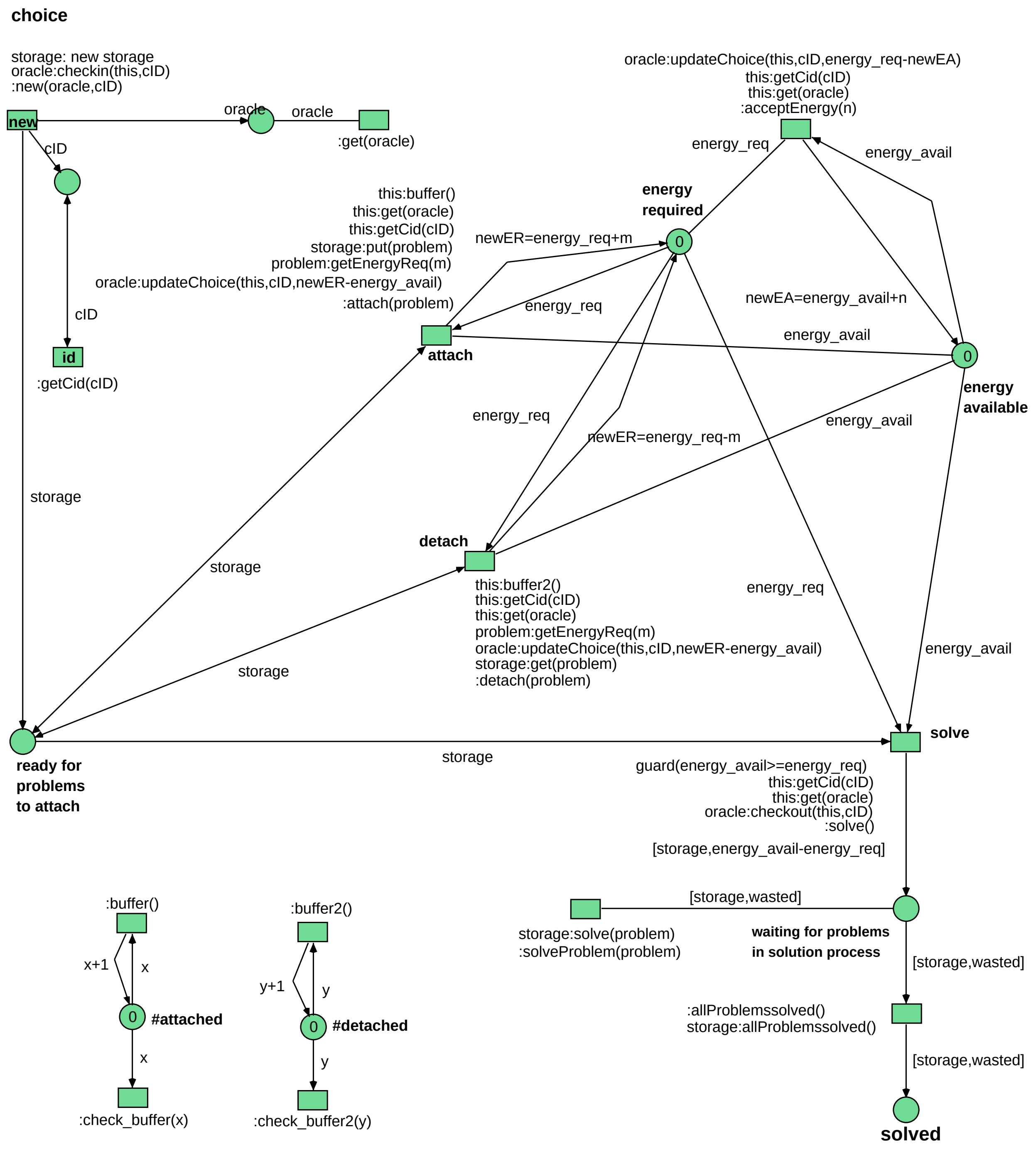


Fig. 12. Object net Choice

The choice produces a reference to its storage while being instantiated and remains ready for problems to attach until problems start attaching and detaching. The choice has two places for energies ("energy required" and "energy available"). It can accept energy with the :acceptEnergy(n)-Method. It has an ID in order to be identified and a reference to the oracle. The number of attachments and detachments is stored in the two counters #detached and #attached which is relevant for the statistics at the end of the simulation. Finally, if choices have more energy available than energy required a decision can be made and the storage is emptied gradually (waiting for problems in solution process) until all problems are solved. The detailed interaction with the other nets involved unfortunately can not be explained here.

6 Conclusion and Outlook

Starting from the problem of formalizing sociological theories a concrete one has been chosen and successfully modeled. By applying Petri nets an operational semantics could be given to the Garbage Can Model of Organizational Choice.

Petri nets and in our case Reference nets have been successfully applied and showed the advantages of its underlying concepts. These are:

- The generally well known advantages of Petri nets, especially the explicit expression of concurrency, conflicts, and confusion.
- The object-orientation to identify objects in the sociological application area, even if advanced concepts like inheritance or polymorphism have not been used.
- The concept of nets as active tokens to separate environment, active elements, and their interaction.
- The synchronous channels and the net instances of the net templates allowed a clear separation of the model and a clear dynamic behavior.

It should be noted that the success of the project so far crucially depended on the tool Renew, without which the modeling and execution of the model would not have been possible in such a compact way. By treating actors and active parts of the sociological theory as objects for sociologists an intuitive model could be developed.

As a result of the modeling process and the discussion about it new questions concerning the sociological theory could be found for and by the sociologists in our group. These are:

- Many aspects of the sociological model remain implicit and are not well structured.
- Global Knowledge is assumed.
- Structural dependencies of the relations between the actors of the model exist.
- Linkage to other theories, which are faded out by the original Garbage Can model, are unreflected.
- Confusion can be seen as a basic phenomena.
- Concurrency and causal dependencies need to be handled.
- Different context levels in the sense of environment and the restrictions of communication exist.
- The main principle of the Garbage Can model as a chaotic system is questioned, since local rationality is observable.

– It is necessary to formalize action oriented aspects more detailed than in the original model.

Overall it was interesting to see that the incremental prototyping approach was very successful and that the extension was relatively easy due to the overall object-orientation and the internal organization of the objects in a special kind of scenario nets or workflow nets as described in [Mol96] and [MV00].

The gained improvements and especially the results were inspiring the sociologists in our group to reformulate some parts of the theory. This shows the weaknesses of the traditional view and also allows to extend the Garbage Can theory by some other theories, especially those related to behavioral theories. The structures and processes induced by the additional aspects will lead to better models of organizational choices and will allow our group to apply this new theory to organizational units within public institutions. Especially, we plan to look at decision procedures within universities.

The means to express these new theories shall be done on the basis of the "nets as active tokens" concept. This should allow for a separate adaptation of the environment and its objects acting within it, capturing the aspects of local changes and of mobility. Furthermore the concepts of agents with some sociological properties can be developed according to [MW97].

With the here presented model a new application area has been tackled the first time. The high-level Petri net formalism could be applied by the computer scientists to a "practical" problem of sociology. Within our project we will extend this even further to other theories. Doing so the results shall be used to enhance agent architectures and shall lead to generalized sociological theories.

7 Acknowledgments

We would like to thank Daniela Hinck and Rolf von Lüde for the excellent sociological input, Rüdiger Valk for the basic model which was the starting point of the Petri net modeling, and Olaf Kummer and Frank Wienberg for their fruitful comments on and support with the Renew tool.

References

[Aal97] Wil van der Aalst. Verification of workflow nets. In Azéma and Balbo [AB97], pages 407–426.

[AB97] Pierre Azéma and Gianfranco Balbo, editors. *Application and Theory of Petri Nets 1997*, number 1248 in Lecture Notes in Computer Science, Berlin Heidelberg New York, 1997. Springer-Verlag.

[AMVW99] Wil van der Aalst, Daniel Moldt, Rüdiger Valk, and Frank Wienberg. Enacting Interorganizational Workflows Using Nets in Nets. In Jörg Becker, Michael zur Mühlen, and Michael Rosemann, editors, *Proceedings of the 1999 Workflow Management Conference Workflow-based Applications, Münster, Nov. 9th 1999*, Working Paper Series of the Department of Information Systems, pages 117–136, University of Münster, Department of Information Systems, Steinfurter Str. 109, 481149 Münster, 1999. Working Paper No. 70.

[CH94] S. Christensen and N.D. Hansen. Coloured Petri nets extended with channels for synchronous communication. In Rober Valette, editor, *Application and Theory of Petri Nets 1994, Proc. of 15th Intern. Conf. Zaragoza, Spain, June 1994*, LNCS, pages 159–178, June 1994.

[CMO72] M.D. Cohen, J.G. March, and J.P. Olsen. A garbage can model of organizational choice. *Administrative Science Quarterly*, 17:1–25, 1972.

[HM99] Sven Heitsch and Marcel Martens. Soziologische Modellbildung mittels Referenznetzen. Studienarbeit, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, November 1999.

[HMM99] Sven Heitsch, Marcel Martens, and Daniel Moldt. Petri Nets with Synchronous Channels Applied to a Sociological Example. Work in Progress Presentation at CPN'99 in Aarhus, see URL: `http://www.daimi.au.dk/CPnets/workshop99/`, 1999.

[Imh99] P. Imhof. Social studies of social simulation. URL: http://www.tu-harburg.de/tbg/Deutsch/Mitarbeiterinnen/Peter/Scientific.html, October 1999.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process: UML; The complete guide to the Unified Process from the original designers.* Addison-Wesley object technology series. Addison-Wesley, Reading, Mass., 1999.

[Jen92] Kurt Jensen. *Coloured Petri Nets: Volume 1; Basic Concepts, Analysis Methods and Practical Use.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin Heidelberg New York, 1992.

[Köh99] Michael Köhler. Algebraische Strukturen von Objektnetzen. Master's thesis, Universität Hamburg, 1999.

[Köh00] Michael Köhler. Distribution references and undecided markings. Technical report, University of Hamburg, Dept. for Computer Science. FBI-HH-M-293/00, 2000.

[Kum98] Olaf Kummer. Simulating synchronous channels and net instances. In J. Desel, P. Kemper, E. Kindler, and A. Oberweis, editors, *Forschungsbericht Nr. 694: 5. Workshop Algorithmen und Werkzeuge für Petrinetze*, pages 73–78. Universität Dortmund, Fachbereich Informatik, 1998.

[Kum99] Olaf Kummer. A Petri net view on synchronous channels. *Petri Net Newsletter*, (56):7–11, 1999.

[KW99] Olaf Kummer and Frank Wienberg. Renew homepage. URL: `http://www.renew.de`, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, 1999.

[Lak95] C.A. Lakos. From Coloured Petri Nets to Object Petri Nets. In *16th International Conference on the Application and Theory of Petri Nets*, number 935 in Lecture Notes in Computer Science, pages 278–297, Torino, Italy, 1995. Springer.

[Lou93] Kenneth C. Louden. *Programming languages: principles and practice.* PWS-Kent, Boston, 1993.

[Luh88] N. Luhmann. Organisation. In W. Küppers and G. Ortmann, editors, *Mikropolitik. Rationalität, Macht und Spiele in Organisationen*, pages 165–186. WDV, Opladen, 1988.

[Mal99] T. Malsch. Erforschung und Modellierung künstlicher Sozialität. URL: `http://www.tu-harburg.de/tbg/SPP/spp-antrag.html`, August 1999.

[ML89] M. Masuch and P. LaPotin. Beyond garbage cans: An ai model of organizational choice. *Administrative Science Quarterly*, pages 38–67, 1989.

[Mol96] Daniel Moldt. *Höhere Petrinetze als Grundlage für Systemspezifikationen.* Dissertation, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, August 1996.

[MV00] Daniel Moldt and Rüdiger Valk. Object-oriented Petri Nets in Business Process Modelling. In Wil van der Aalst, Jörg Desel, and Andreas Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, number 1806 in Lecture Notes in Computer Science, pages 254–273, Berlin, Heidelberg, New York, 2000. Springer-Verlag.

[MW97] Daniel Moldt and Frank Wienberg. Multi-agent-systems based on coloured Petri nets. In Azéma and Balbo [AB97], pages 82–101.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall Inc., Englewood Cliffs NJ, 1981.

[Rei92] Wolfgang Reisig. *A Primer in Petri Net Design.* Springer Compass International. Springer-Verlag, Berlin Heidelberg New York, 1992.

[SB94] C. Sibertin-Blanc. Cooperative nets. In Robert Valette, editor, *15th International Conference on the Application and Theory of Petri Nets*, number 815 in Lecture Notes in Computer Science 815, pages 471–490, Berlin Heidelberg New York, 1994. Springer-Verlag.

[Val96] Rüdiger Valk. On Processes of Object Petri Nets. Fachbereichsbericht FBI-HH-B-185/96, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, June 1996.

[Val98] Rüdiger Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In Jörg Desel, editor, *19th International Conference on Application and Theory of Petri nets*, number 1420 in LNCS, Berlin, 1998. Springer-Verlag.

Specification and Validation of a Concurrent System: An Educational Project

Gérard Berthelot[1] and Laure Petrucci[2]

[1] CEDRIC-IIE, Institut d'Informatique d'Entreprise,
18 allée Jean Rostand, F-91025 EVRY Cedex
berthelot@iie.cnam.fr
[2] LSV, CNRS UMR 8643, ENS de Cachan
61 avenue du président Wilson, F-94235 CACHAN Cedex
petrucci@lsv.ens-cachan.fr

Abstract. Since several years, we are in charge of a course on specification and validation of concurrent and reactive systems. At the end of this course, the students must carry out a project on a model railway. They have to specify the railway, to validate their model and finally to translate it into a program running the model railway with up to five trains.
In this paper, after presenting the problem, we describe how it is specified and checked, step by step, by the students. We also explain how the analysis results lead to a policy for the switches management. Finally, we give some hints about the implementation.

1 Presentation of the Problem

In this paper, we report about a teaching experience for a group of twenty graduate students, during their second year in engineer school. Their background consist of two years studies in mathematics and physics, followed by one year in computer science. During the previous year they learned the basics of computer programming, operating systems (in particular Unix processes) and concurrent programming concepts. Our course is optional and is composed of several topics: communicating automata, coloured Petri nets, temporal logics, tools for specification and verification (Design/CPN, Spin, Esterel), real-time systems. After 80 hours of lectures and exercise courses, the students have to carry out a project by themselves. They spend roughly 70 hours of personal work, to complete the project from specification to hardware implementation. The students choose among the three previous tools. Here, we will focus on the use of Design/CPN only.

1.1 Goals of the Project

Teaching programming of concurrent systems is not so obvious, as the students are used to programming in a sequential style. In order to alleviate this difficulty, we decided to have them manage a system with intrinsic parallelism. A model railway, allowing several trains, is altogether well-known, convenient, inexpensive and appealing to students. Moreover, such a system can be used to tackle real-time concepts and tools. Therefore, we decided to create a course including all aspects of parallel programming, from the specification phase to the real-time programming and implementation on the physical model. Nevertheless, this course is still being further developped, we have not yet addressed time aspects. This can be done in the future using interval timed coloured Petri nets as in [dAO94].

1.2 What is Asked from the Students

The students' project was not only designed as an approach to parallel programming, but also to emphasize the benefits of specification and validation prior to programming. In particular, the students were asked to produce a graphical model, having the same aspect as the physical railway. It should be pointed out that this is not demanded for esthetic reasons but it helps a lot to understand whether a configuration of the railway is normal or not. This facilitates a boring and error-prone effort to synthesize a long firing sequence. It represents an important benefit for debugging.

So far, the hierarchical coloured Petri nets ([Jen92]) have been chosen as a model, due to their support of hierarchies, simulation, occurrence graph requirements. Hierarchies allow a structured design, where the top-level net sticks to the hardware layout. The use of high-level nets allows both to capture several cases in a single transition and group the parameters of trains and tracks sections into one place. The use of an ordinary net leads to unreadable intricate graphics. The tool used to support this model is Design/CPN ([CPN96]).

Two views were considered in order to exploit the physical railway.
The first view, which will be named "real railway view", is meant to operate as a real railways, with the same rules. Effectively, the students are asked to design a system where each train is assigned a route. They must prove that some security requirements are satisfied (no collision, no more than one train per section) as well as efficiency (no deadlock). Once the model has been proved correct, the students must figure out how to translate the net into a set of processes, synchronized using semaphores, which can be run on the model railway. The transformation must be systematic, its soundness and the preservation of properties proved must be justified.
The second view is not intended to mimic a real railway system. Rather, it is related to an adaptative routing system, and henceforth will be named hereafter the "adaptative routing system view". The behavior of trains must be adapted to local conditions. Namely, at each switch, their route can be chosen among several tracks and a train may even go back when impossible to continue forward. Although surprising at first glance, such behavior of trains offers several complex routing possibilities and thus appears as a challenge. The students must design a routing policy so that the same security and efficiency requirements as

before are fulfilled. Then they have to deduce from the net the program of a controller, and implement it on the railway system.

2 The Model Railway

The model railway is depicted in figure 1. It consists of about 15 meters of tracks, divided in 16 sections (blocks B1 to B16) plus 2 sidetracks (ST1 and ST2), linked using 4 double or triple switches and one crossing. The way the trains can circulate on the switches and the crossing is indicated by the arrows in figure 1. The traffic on all tracks can go both ways. Although one can notice that switch 1 (and also switch 2) is composed of two elementary ones, it is managed as a single unit, due to the short distance between the two physical components. The railway is connected to a computer via a serial port which allows to read information from sensors and send orders to trains through the tracks or directly to switches. A section is equipped with one sensor at each end, thus allowing to detect the entrance or exit of a train. The orders sent to trains can be either stop or go forward/backwards at a given speed.

3 Specification Using Coloured Petri Nets

In this section, we describe the different steps encountered by a typical student[1] following the second view, i.e. the adaptative routing system described at the end of section 1). We will give hints about the differences with the real railway view in section 3.6.

The student is asked to proceed step by step in the construction of the coloured net representing the model railway. At first, the model only describes the main loop, i.e. the railway without the 2 sidetracks and the crossing. The analysis of this net is performed for three, four and five trains. At first there is an obvious deadlock which is corrected. Then another one arises when adding an extra train. This is repeated until we obtain a model which is satisfactory for five trains. All the corrections made correspond to a policy for managing the four switches and moving between contiguous sections. Then the two sidetracks are added and the analysis is done directly for five trains. Finally, the crossing is added. Due to the large amount of memory (and time spent) to generate the occurrence graph and check properties, the verification is first performed for four and then for five trains.

3.1 Design of a Hierarchical Coloured Petri Net

One of the requirements of such a project is to obtain a model which can easily be understood. This allows, as we will see later on, to facilitate the understanding of errors encountered when analyzing the model with the occurrence graph. Their correction also becomes much more natural.

[1] in fact, each project is conducted by a pair of students.

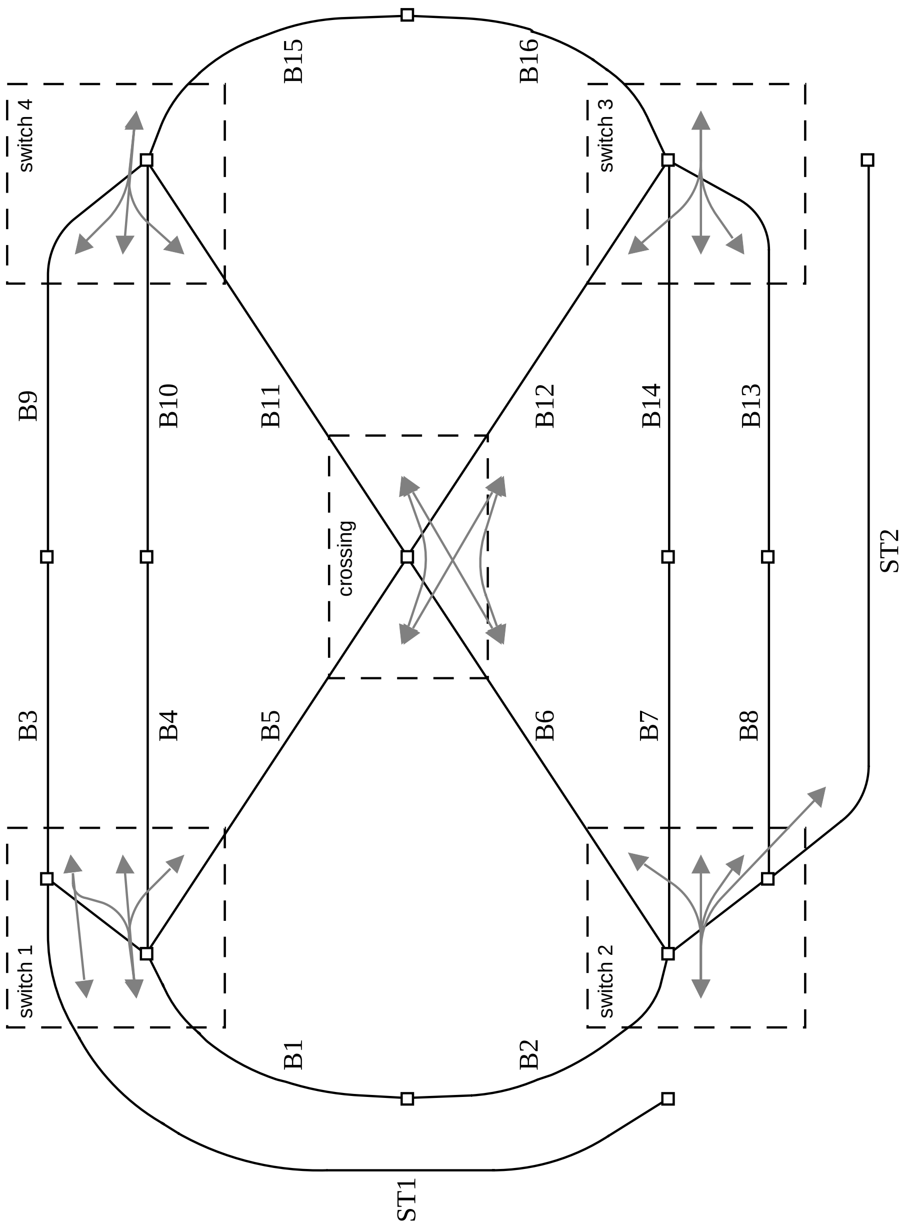


Fig. 1. The tracks of the model railway.

To do so, the use of page hierarchies was required: the prime page represents the whole railway, without any consideration of the policy used to move from one section to the next. This policy is described in subpages, corresponding to the different switches and moves between contiguous sections. A single look at this prime page shows the current state, i.e. where the different trains are located. The similarity between the physical railway model (figure 1) and the prime page (figure 2 for a partial representation and figure 7 for the full one) is easily noticed. The places represent the sections (they have the same names in both figures), while the transitions indicate the possible moves.

3.2 Model of the Main Loop

The students were asked to design their model step by step. Therefore, they started with just the main loop, i.e. the railway without the two sidetracks nor the crossing in the middle.

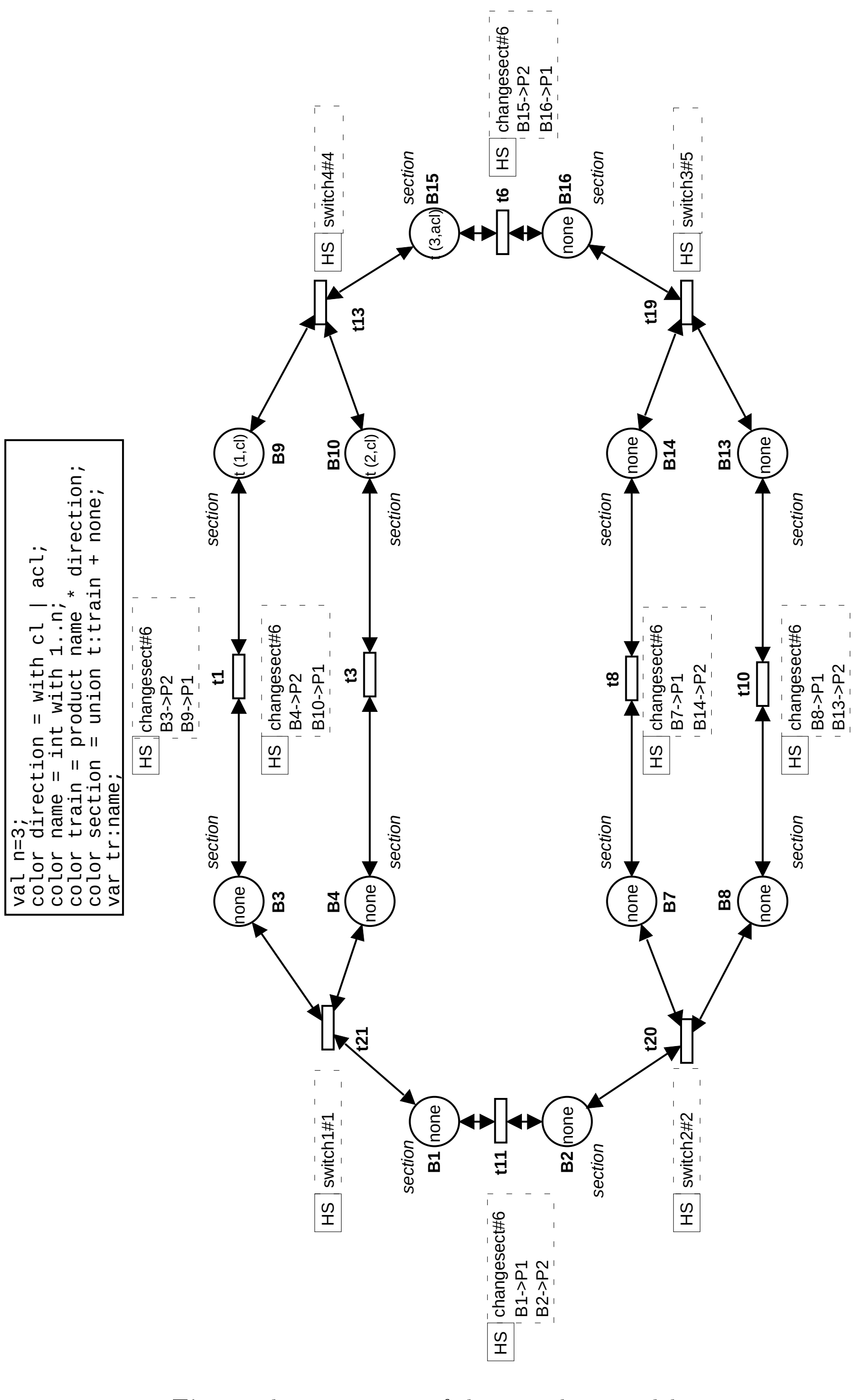
The prime page of the first net is shown in figure 2. Each place represents the railway section with the same name. It always contains one token, with a value characterizing the state of the section, i.e. either a train is in the section, or the fact that there is none. This is expressed with the union type:

```
color section = union t:train + none;
```

To stick to the real system as much as possible, the student generally chooses to describe trains with both a name and the way they are running, namely clockwise (`cl`) or anti-clockwise (`acl`).

All the transitions are substitution transitions, i.e. they must be substituted by a net with the same input and output places. This can be seen for e.g. transition t1 of figure 2, which has a box next to it with the name of the subpage to be substituted to the transition (changesect) and the correspondence between the names of input and output places of both nets (place B3 of figure 2 corresponds to place P2 of figure 3). The transitions are of two kinds:

– those allowing to move between two contiguous sections with no choice; a reasonable hypothesis is that all transitions of this kind have the same behavior. Therefore, all these substitution transitions will refer to the same subpage. Initially, we will assume that a train can move to the next section if this section is empty, as depicted in figure 3.
– the switches which can permit either to move from a unique section to two different ones or vice-versa. This is described in figure 4, where the net corresponds to one possible direction of trains (switch 1). Another page describes the other way round. These nets could have been merged into a single one, with a place containing the direction used by the switch. That choice would have lead to a less intuitive interpretation of the prime page, and to a unique page used by all the switches. Moreover, when enhancing the model to add the other parts of the railway, the switches become different. Similarly, some students have used two instances of the net in figure 3 to model the simple switch, but then, when changing the policy for moving



```
val n=3;
color direction = with cl | acl;
color name = int with 1..n;
color train = product name * direction;
color section = union t:train + none;
var tr:name;
```

B1
B2
B3
B4
B7
B8
B9
B10
B13
B14
B15
B16
none
t (1,cl)
t (2,cl)
t (3,acl)
section
t1
t3
t6
t8
t10
t11
t13
t19
t20
t21
HS switch1#1
HS switch2#2
HS switch3#5
HS switch4#4
HS changesect#6 B1->P1 B2->P2
HS changesect#6 B3->P2 B9->P1
HS changesect#6 B4->P2 B10->P1
HS changesect#6 B7->P1 B14->P2
HS changesect#6 B8->P1 B13->P2
HS changesect#6 B15->P2 B16->P1

Fig. 2. The prime page of the main loop model.

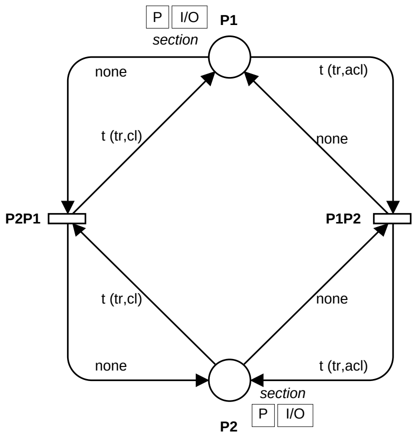


Fig. 3. Moving between two contiguous sections.

from one section to the next, the management of switches was also modified. This is often undesirable.

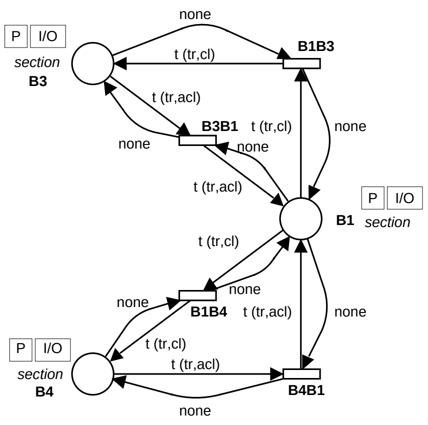


Fig. 4. A simple switch.

Once the net is designed, the students starts examining its behavior. It is obvious that the initial marking (with 3 trains) given in figure 2 is a deadlock. Thus, the student must design a less simple policy for switches, depicted in figure 5.
The switches are (for the moment) composed of two tracks arriving from the

same direction (such as B3 and B4) and one from the other side (as B1). When 3 trains arrive altogether on the switch in a deadlock situation, the new policy consists in having the train on the single track side (e.g. the train on B1) go backwards. Such a possibility takes sense only in the adaptative routing system, but not in a real railway system. This is modeled by transition `retB1` in figure 5.

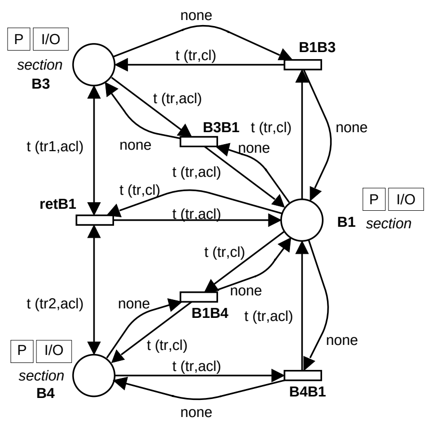


Fig. 5. New policy for switches.

It is easily noticed that the initial marking with three trains will enable only the new transition, and thus only the train going anti-clockwise will change direction. Thereafter, all the trains will go in the same direction, i.e. clockwise. To have interesting results, the student adds an extra train, and analyses his new net.

3.3 Verification with Four Trains

Now, the new policy for switches is used and a fourth train is added, by changing the initial marking of place B3 into `t (4,acl)`, and the value `n=4` in the declaration node.

The analysis of the model is performed by means of Design/CPN occurrence graph tool ([Jen94], [CPN96]). Once the graph is generated, a standard report and some additional properties are checked. This allows to discover nodes satisfying undesired properties, as explained later. To facilitate the understanding of the trains situation, the feature provided by Design/CPN which allows to transform a state of the occurrence graph into the corresponding marking in the simulator is used. Thus, the prime page is updated with the unwanted marking, and the user can see in a glimpse the locations of trains.

Analysis Results After a single occurrence graph generation, faced with the huge number of states, the student often guesses that the trains names highly contribute to the state space explosion, although they are useless, from a verification point of view. Effectively, in the problem considered, it is not necessary to know where a particular train is, but only that there is a train in a particular section, and also where it is heading. This could be formalized using symmetries, but we lack time to teach the theoretical and syntactical aspects. Moreover, the marking equivalence derived from this symmetry is totally obvious.

Occurrence Graph with Trains Names The occurrence graph obtained with four named trains contains 21.574 nodes and 72.026 arcs. It is computed in 334 seconds[2]. The strongly connected components (SCC) graph has 1.243 SCCs and 6.666 arcs. It is calculated in 18 seconds.

Occurrence Graph without Names When removing the trains names, i.e. only the direction the train goes is kept in the token value, the occurrence graph obtained, still with four trains, has 2.166 nodes and 7.157 arcs. It takes 5 seconds to obtain it. The SCC graph has 237 nodes and 1.245 arcs. It is computed in 1 second.

The following step in the project is the verification of the net. The student must formalize what does a correct behavior of the system mean, and indicate the properties that should be satisfied. Having some experience, through exercises, of the properties provided by the standard report of Design/CPN, the student tries to figure out how they can be used in order to prove the correctness of his model. This function allows to obtain in a file a textual result for the usual net properties (e.g. bounds, dead markings, liveness, …). It turns out that most properties are safety properties and can be checked using only the standard features.
The results and their interpretation are the following :

- all the lower and upper best integer bounds are 1. Thus *there is always exactly one token in each place*;
- all the best upper multi-set bounds are `1‘t(cl)++ 1‘t(acl)++ 1‘none`. This property, together with the previous one, shows that *each section can contain either exactly one train going one way or the other, or none*;
- there are *6 dead markings.* At first, the student is surprised. The evaluation of function `ListDeadMarkings();` provides him with the list of all dead markings. Then, he visualizes each of these markings on the prime page, using the feature of Design/CPN which puts a given marking of the occurrence graph into the simulator. Three of the dead markings have a train going clockwise in `B15` and a train going anti-clockwise in `B16`. The three other ones are similar, with sections `B2` and `B1`.

The student comes to the conclusion that the policy to move between two contiguous sections with no choice has to be improved.

[2] all the results in this paper were obtained on the same machine: a Linux PC Pentium II 450 MHz with 256 Mb of memory. The computation times are those given by Design/CPN.

Policy Adopted In order to avoid the previous situation, the student decides to have both trains go backwards, as modeled by transition `retboth` in figure 6, giving the following arguments:

– if both trains return where they come from, they will not travel in the same direction, thus such a policy should limit or delay the possibilities for all trains to go in the same direction;
– both trains are treated in the same manner, there is no priority.

The new model is then analyzed for four and five trains. The results with five trains will now be discussed.

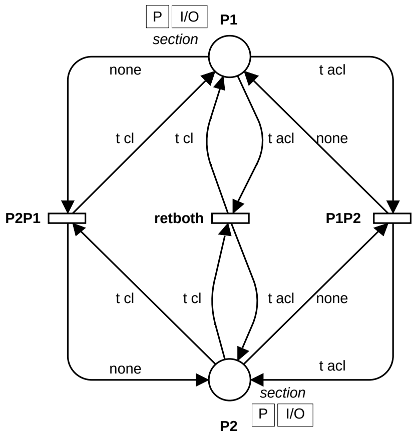


Fig. 6. New policy to move between two contiguous sections.

3.4 Verification with Five Trains

A fifth train going anti-clockwise is added in place `B2`.

Analysis Results The occurrence graph obtained for five trains has 24.556 nodes and 97.020 arcs. It is computed in 430 seconds. Its SCC graph has 615 nodes, 3.128 arcs and was calculated in 18 seconds. The properties obtained from the standard report, show that the bounds are the same as previously, *there is no dead marking nor home marking.*
This last property is quite intriguing for the student. He decides to have a closer look to the terminal SCCs. Therefore, he used the following functions (where the result is given after the arrow):

```
PredAllSccs SccTerminal;        -> [~615,~534]
length(SccToNodes(~615));       -> 792
length(SccToNodes(~534));       -> 792
```

The first function gives the list of all terminal SCCs. They are two, numbered ~615, and ~534. Then, the number of nodes in each of these terminal SCCs is 792. After looking at a marking of each component, the student concludes that this is a normal situation: in SCC ~615, all trains go clockwise while in SCC ~534, all trains go anti-clockwise. The two SCCs remain separated as, considering the routing rules adopted, a train cannot go backwards if it does not meet a train gong in the opposite direction. So, such situations are acceptable w.r.t. the initial requirements.

3.5 The Complete Railway

When the main loop works correctly, the student enhances his model by adding the two sidetracks and the crossing.

Final Model The prime page of the model of the complete railway (see figure 7) has 6 additional places corresponding to the 2 sidetracks and the 4 inner sections, and one extra substitution transition which models the crossing.

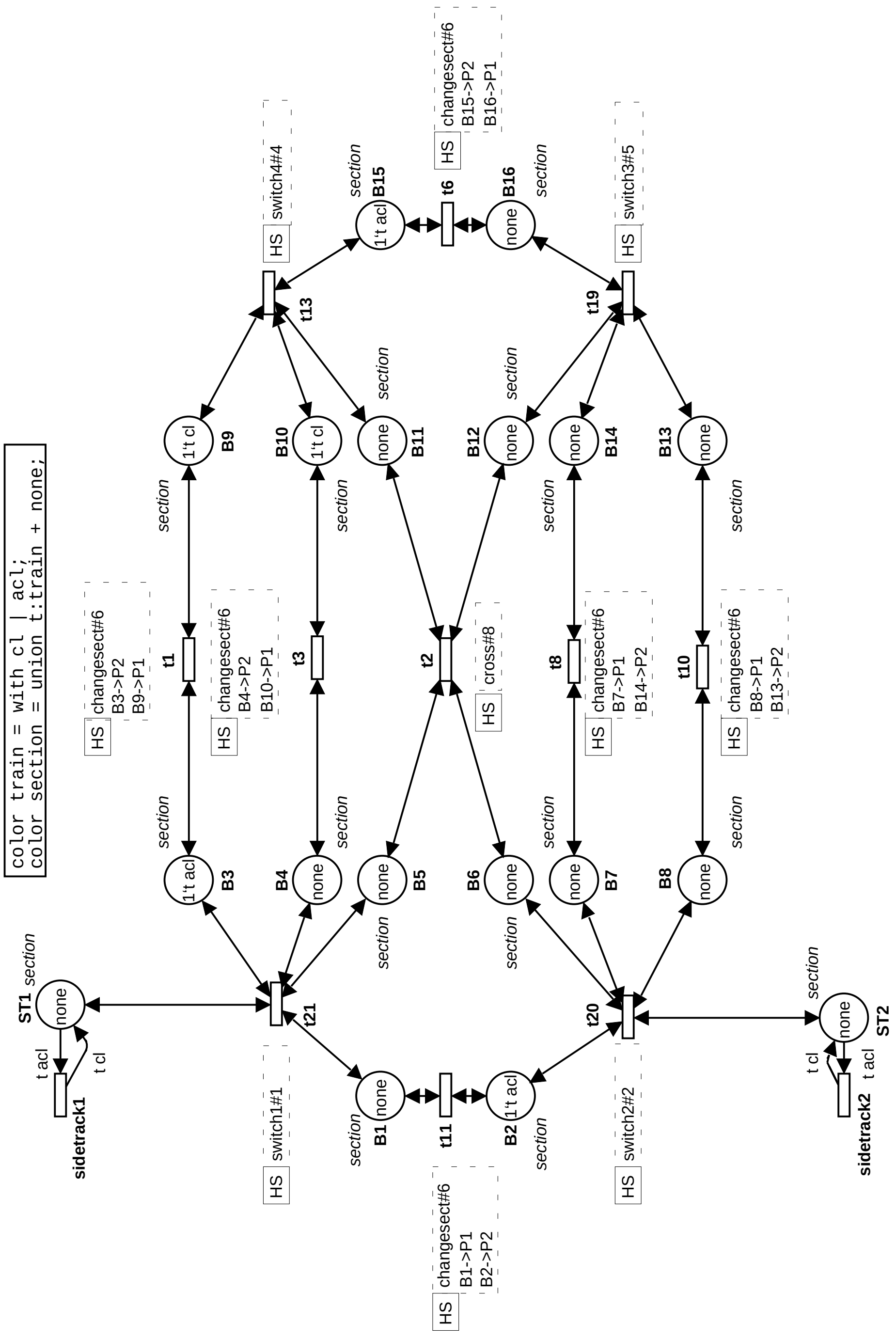
A subpage, presented in figure 8, is added to represent the inner crossing in the railway. It takes into account all the possible moves on the inner crossing, as described in section 2. Moreover, the case where four trains want to enter the crossing – and will then be blocked – is treated by forcing the train on `B5` to go backwards.

The four switches subpages are modified to take into account the new possible movements, but their general policy remains the same, just taking into account more possibilities for a train to enter and exit a switch. The nets in figures 9, 10, and 11 give an idea of the complexity of the complete model.

The fourth switch can easily be deduced from the third one, by changing the place names and direction of trains.

Analysis Results As a lot of time is needed to build the occurrence graph of the full model, it was first done with four trains and then with five.

The occurrence graph with four trains contains 48.957 nodes, 228.790 arcs and was calculated in 2.991 seconds (50 minutes). Its SCC graph has 1 node, no arc and is obtained in a bit less than 2 minutes.
The standard report gives us the same integer and multi-set bounds as before. *All the reachable states are home markings.* This is interesting, because it shows that any reachable train distribution can always be reached again. There is no deadlock. But *transition `retB2` of subpage `switch2` is dead.* This means that with these four trains, the situation were a train going anti-clockwise on section `B2` is blocked can never occur. When looking at the design of the railway, the student notices that this is normal as there are 4 possibilities for this train to go forward, and only three can be occupied by the other trains. The question now is: what happens with a fifth train?



```
color train = with cl | acl;
color section = union t:train + none;
```

Fig. 7. The prime page of the model railway hierarchical coloured Petri net.

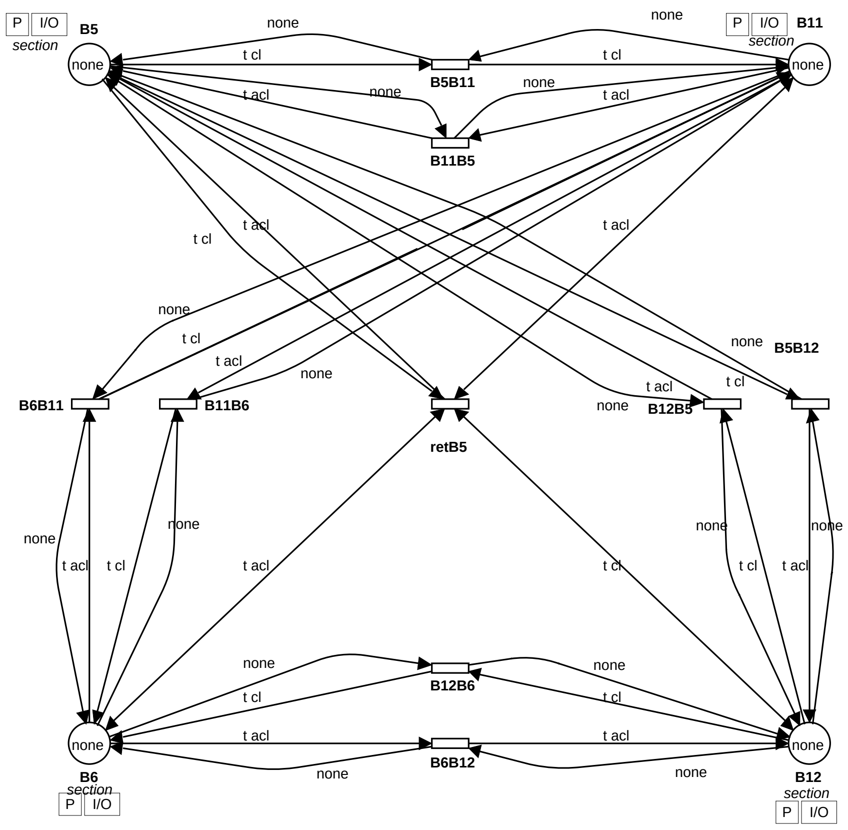


Fig. 8. The subpage modeling the inner railway crossing.

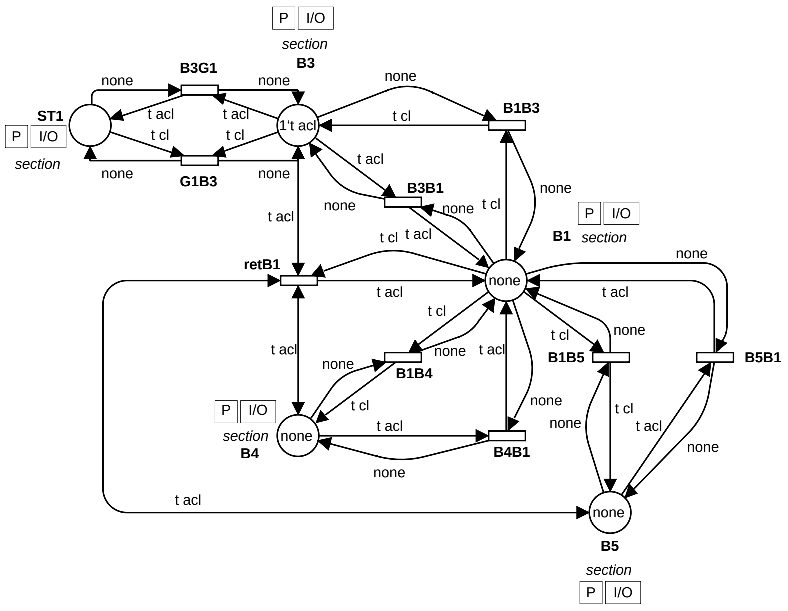


Fig. 9. The first switch (transition t21 of figure 7).

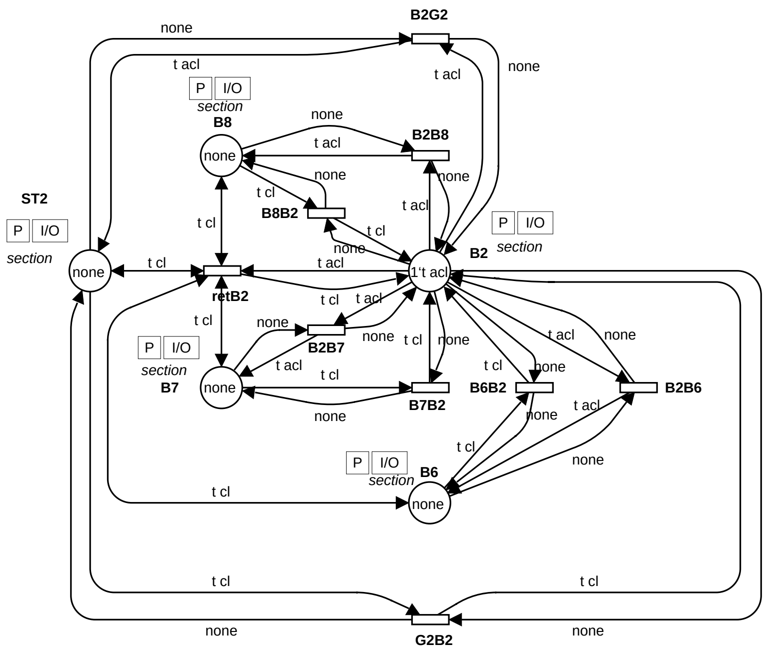


Fig. 10. The second switch (transition t20 of figure 7).

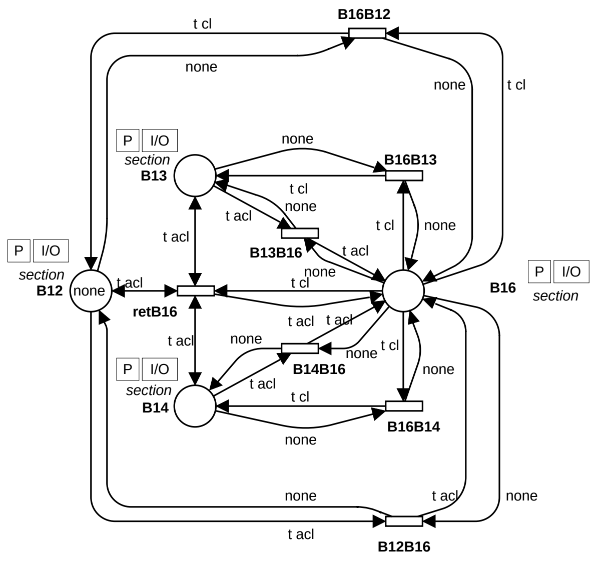


Fig. 11. The third switch (transition t19 of figure 7).

The Occurrence Graph with Five Trains has 274.082 nodes, 1.500.384 arcs and was computed in 103.221 seconds (1 day, 4 hours and 40 minutes)[3], approximately the same time was necessary for the SCC graph which finally contains 1 node and no arc. This informs the student that, as for four trains, all the reachable states are home markings. There is still no deadlock and *all the transitions are live.* Thus the model meets the initial requirements : allow up to 5 trains, avoid collisions and deadlocks.

3.6 Modeling the "Real Railway System"

When the first view (i.e. mimicking a real railway system) is considered, a route must be assigned to each train. This has three consequences: First, when a train enters a switch, it must exit it as specified in its own route, and not by a randomly chosen exit. As a second consequence, trains must keep their identity and cannot become anonymous. Finally, a train may have to book several sections in advance in order to avoid deadlocks: e.g. if a train in B7 wants to enter B2, it should also make the reservation for B1, otherwise it can be blocked by a train coming the other way round. As concerns modeling, a transition must not only be connected by the arcs to places representing the next and/or previous resource, but also to the places representing the resources necessary one or more steps later. Hence, the net is graphically less close to the physical railway, and also it is more difficult to create a hierarchical net as there are special transitions

[3] with intensive use of the 1Gb swap space

associated with each train.
The number of states in the occurrence graph strongly depends on the length and the complexity of the routes designed by the students. Another big difference with the second approach is that verification mainly consists in deadlock detection between trains. It is obvious that complex routes and the simultaneous reservation of four sections can lead to deadlocks very difficult to foretell. To conclude, the complexity of the model and the possibility to fully check it depends a lot on the options chosen by the student. This is the reason for explaining the other view.

4 From the Specification to the Implementation

When the model of the net has been analyzed, and its desired properties proved, the student has to translate it into a `C` program, having a behavior as close as possible to the net's one. To do so, he has roughly two solutions: the first one consists in writing a controller (a Petri net simulator), the second is to split the net into a set of synchronized processes. The student naturally tends towards the first solution, which is closer to sequential programming and automata theory, but, after a deeper insight, some of the students accept to try the second solution. We shall now describe both in more detail.

In the simulator approach, the color sets and associated operators are translated into `C` data structures and functions. Then, each transition is splited into a boolean precondition function and a post function. The former allows to check if the transition is firable, while the latter is called when firing to change the marking and transmit orders to the hardware. The core of the simulator is an endless loop which, at each step, scans the transitions list to see which ones are firable, then fires some of them. It should be noticed that, contrary to what happens in a general simulator, the binding of variables of transitions is quite easy since each place contains exactly one token. Nevertheless, when several transitions are enabled, a fair choice must be provided, using for instance a random number generator. A companion process is in charge of reading the information from the hardware, and update accordingly the state variables.

In the second solution, which is more natural when using the "real railway view" because the trains are seen as independent entities which travel along a route, the set of transitions of the net must be partitioned in order to build up processes which have to synchronize using semaphores. The most intuitive and logical way to do so is to associate each train with a proper process. With this approach, each part of the railway must be considered as a critical resource. The student can directly apply the classical Dijkstra method([Dij65]) to access resources. Each resource is described by a set of state variables. Processes must request resources using a general mutual exclusion semaphore. If a process is blocked, it leaves the critical section and hangs up on a private semaphore until it is awaken by another process. With this approach, no fairness problem appears at first, provided that semaphores have FIFO queues. However such a problem arises when a train arrives at a fork with several possible exits, and several are compatible with the route. The student must add a mechanism to choose randomly which transition to try. As in the previous approach a companion process handles the hardware inputs.

The program obtained is then downloaded on the PC which monitors the train model and the student may conduct some experiments. The program is often rapidly effective, after some tuning (speed of trains), addition of hardware commands (switches, crossing), and detail improvements (traffic lights). The student can see that no real problem arises, except from bad hardware management (if a sensor does not "see" a train, this one becomes a "crazy train" which must be physically removed as soon as possible). However, most of times a long run results in a periodic behavior, even if the theoretical analysis predicted a less rigid scheme. This is due to the fact that even with a fair program, first come trains are also first served ones at critical points, and the delay to run through a block depends mainly on its length. So, the model train can adopt only a subset of behaviors of its theoretical model. The exact analysis would require to take into account time aspects, but it was not tried yet.

5 Conclusion

In this paper, we have reported a teaching experience using a train model and high-level nets. A similar experience for modeling a train system is described in [HURK98], but the students could not achieve verification.
In spite of its childish aspect, the train model forces students to touch, master and manage all the notions rendering parallel programs error prone: critical section, deadlock, fairness, time and combinatorial explosion. Moreover, it is a very strong evidence that such programs cannot be developed from scratch (bottom-up style) as students tend to do, but must be rigorously modeled and analyzed before implementation. An additional benefit from this project consists in tackling a process control problem.
From students' opinion, it appears that handling a real system helps and motivates both to model the system and to formalize the properties that it should satisfy. However, they are disappointed by the state space explosion problem which arises quite early in the verification. Thus only restricted or more abstracted problems can be validated.

A further step would be to manage time aspects. An automatic code generation feature would have helped a lot for the last step of the project, i.e. the implementation part.

References

[CPN96] META Software and Aarhus University. *Design/CPN 3.0*, 1996. Also available as: `http://www.daimi.au.dk/designCPN`.

[dAO94] W. M. P. Van der Aalst and M. A. Odijk. Analysis of railway stations by means of interval coloured Petri nets. *Real-time systems*, 9:1–23, 1994.

[Dij65] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, september 1965.

[HURK98] W. Hielscher, L. Urbszat, C. Reinke, and W. Kluge. On modelling train traffic control in a model train system. In *Proc. 1st CPN Workshop*, DAIMI PB 532, pages 83–101. Aarhus University, 1998.

[Jen92] K. Jensen. *Coloured Petri Nets: Basic concepts, analysis methods and pratical use. Volume 1: basic concepts.* Monographs in Theoretical Computer Science. Springer, 1992.

[Jen94] K. Jensen. *Coloured Petri Nets: Basic concepts, analysis methods and pratical use. Volume 2: analysis methods.* Monographs in Theoretical Computer Science. Springer, 1994.

Modelling and Analysing a Distributed Dynamic Channel Allocation Algorithm for Mobile Computing Using High-Level Net Methods

Leo Ojala, Nisse Husberg, and Teemu Tynjälä
Helsinki University of Technology,
Laboratory for Theoretical Computer Science,
P.O. Box 9700, FIN-02015 Espoo, Finland
Email: Leo.Ojala@hut.fi, Nisse.Husberg@hut.fi, Teemu.Tynjala@hut.fi
http://tcs.hut.fi

Abstract

A Distributed Dynamic Channel Allocation algorithm has been proposed in [PSS95]. In this paper the algorithm is modelled using Predicate/Transition nets. The same model can be used for any number of cell and channel configurations. The Maria reachability analyser has been used to analyse the protocol for some configurations and these have been deadlock-free and are shown to satisfy the requirement that the same channel never is allocated to two neighbouring cells. The suitability of high level nets for the modelling and analysis of distributed algorithms is discussed.

1 Introduction

The development of mobile computing over the last decade has raised many requirements on wireless communication networks. The radio spectrum is limited so frequencies and channels must be reused to accommodate for the ever growing user population and their increasing demands on service. In the scope of mobile computing hand-held terminals, laptops, and mobile phones connect to a telecommunications network (and possibly to the Internet) via a radio frequency interface.

Mobile computing faces challenges in the form of coping with low bandwidth, high bandwidth variability, heterogeneous networks and security risks [FZ94]. The future networks must provide the user with fast access, information on-demand and foolproof security and do all this cost effectively. Excellent channel management and reuse are a prerequisite for the success of mobile computing.

The development in this area is fast and new systems appear all the time. For some time it will be the case that new and better systems must be created to cope with the tremendous increase. This requires tools which rapidly can verify the correctness of new protocols and algorithms. Traditional testing is too slow and costly and does not guarantee the correctness in all cases.

The Maria reachability analyser [Mar] is designed to verify industrial size systems in an easy way. This work is also a test of the applicability of the design concepts, mainly the adequacy of the input language of the analyser. A first version of the model was presented earlier [OHB99], but it was not complete (lacking timestamps and threshold) and was not analysed in Maria (which was not implemented at that time). The concrete analysis also brought some changes to the model.

In this paper, however, we present the DDCA (Distributed Dynamic Channel Allocation) algorithm [PSS95] using high level Petri Nets, specifically Pr/T (Predicate/Transition) nets [Gen87]. *Inhibitor arcs* are used as a shorthand notation in the model which is general, non tool-specific, allowing for analysis using any high level Petri net tool. The net model used in the analysis, however, contains no inhibitor arcs.

Section 2 presents the DDCA algorithm in general and Section 3 the complete high level Petri Net model of the algorithm. Section 4 presents the analysis results produced by the Maria analyser. Section 5 shows the direction of work to be undertaken in the future.

2 The Distributed Dynamic Channel Allocation Algorithm

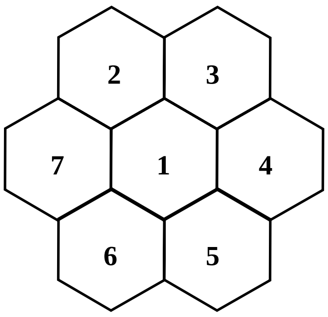
The mobile computing world is normally seen as a collection of base stations (BS) connected together via a fixed network. The area covered by each BS is referred to as a *cell*. Each cell has a set of neighbouring cells. The users carry mobile hosts (MH) which connect to the fixed network via the cells.

There are two ways of handling the reuse of channels. The first way is called fixed channel allocation and the second one is called dynamic channel allocation [Lee93]. In fixed channel allocation the reuse patterns of radio channels are determined a priori based on the geographical layout of the cells. This approach is fast (it can be hard coded) but it is not very maintainable or scalable.

In dynamic channel allocation, channels are assigned according to an algorithm run by a central control or by the base stations. The algorithms are based on determining the channel occupancies of neighbouring cells or on measuring co-channel interference directly. In performance measurements the algorithm measuring co-channel interference is found to be more spectrally efficient [CC96]. In this work we concentrate on traffic measuring based dynamic channel allocation presented in [PSS95].

The algorithm handles a set of $\mathbf{N_k}$ channels which are ordered. The channel with the lowest frequency is the first channel, and the channel with highest frequency is the nth channel where n is the number of channels. The set of all channels is denoted by *Spectrum*.

The cells in the network have a fixed set of neighbours Nbr. Normally these cells are *hexagonal* — that is, each has six neighbours (Figure 1). Our model does not assume anything about the number of neighbours. The channels allocated to a cell C_i are represented by the set *Allocated*$_i$. Each allocated channel is in one of three subsets: *Busy*$_i$, *Transfer*$_i$, or *Available*$_i$. The set *Busy*$_i$ contains all the channels that are currently used in active communication. The set *Transfer*$_i$ contains those channels that will be transferred to a neighbouring cell to support a communication session. If a channel is allocated to a certain cell but not busy or being transferred, it is in *Available*$_i$. Note that all these sets change dynamically. In the terminology used in this paper *free* channels are not allocated to any cell.



$$\mathbf{Nbr} = \begin{bmatrix} 2, 3, 4, 5, 6, 7 \\ 1, 3, 7, 0, 0, 0 \\ 1, 2, 4, 0, 0, 0 \\ 1, 3, 5, 0, 0, 0 \\ 1, 4, 6, 0, 0, 0 \\ 1, 5, 7, 0, 0, 0 \\ 1, 6, 2, 0, 0, 0 \end{bmatrix}$$

Figure 1: Hexagonal cell configuration.

The brief description of the original algorithm, as written in [PSS95] is as follows: "When a new communication request originates in C_i, one of the non-busy channels in *Allocate*$_i$ is assigned to support the communication session. If there is no such channel, then after a round of message exchange with the neighbours, a channel that is in the *Spectrum*, but not in *Allocate* set of the cell or any of its neighbours is added to *Allocate*$_i$ as well as *Busy*$_i$. This channel is used to support the session. If such an attempt fails, C_i tries to transfer a non-busy channel from the *Allocate* set of its neighbours to *Allocate*$_i$. If such a transfer is not possible, the communication request is dropped. Otherwise, the communication is successfully completed."

3 Petri Net Model of the DDCA Algorithm

We present a complete Pr/T net model of the DDCA algorithm in this section. The presentation is divided into four phases corresponding closely to the steps of the algorithm as given in [PSS95]. A first version of the Pr/T net model of DDCA algorithm was given in [OHB99], but this version contains so many changes and additions that it is necessary to present the whole net model.

In the model there are $\mathbf{N_c}$ cells connected to neighbouring cells as described by the *neighbourhood relation* Nbr: i $\rightarrow \mathbf{2}^{N_c}$, where $1 \leq \mathbf{i} \leq \mathbf{N_c}$. An example of a hexagonal configuration using a matrix can be found in Figure 1. This relation assigns a set of neighbour cells C_{i_j} to each cell C_i. Here $1 \leq j \leq 6$ (if the cell layout is hexagonal), and $i \neq i_j$ for all j. The number of channels which may be in each cell is represented by the constant $\mathbf{N_k}$. Changing the configuration of the net is simply changing the neighbourhood relation.

There are two very important places in the Pr/T net model. The first of these, **Pch** (referred to in all parts of the net), contains *always* $\mathbf{N_c} \times \mathbf{N_k}$ tokens. The tokens are triples of the form $< \mathbf{k}, \mathbf{i}, \mathbf{s} >$. Here **k** is a *channel* number ($1 \leq \mathbf{k} \leq \mathbf{N_k}$) and **i** is a *cell* number ($1 \leq \mathbf{i} \leq \mathbf{N_c}$). The third element, **s**, represents the *status* of the channel **k** in the cell C_i. The status **s** can have four distinct values, $\mathbf{s} \in \{\mathbf{a,b,f,t}\}$; **a** denotes that a channel is *available*, **b** that it is *busy*, **f** that it is *free* and **t** that it is in *transfer* status.

The other important place in the Pr/T net model is **Pcl** in Figure 2 which keeps track of the connection attempts. This place contains a token **i** for each cell C_i. When a connection attempt originates from the cell C_i, the token **i** is removed from the place **Pcl**. When the attempt either succeeds or fails the token **i** is replaced in the place **Pcl**.

In the *initial state* of the net, all the channels are *free*, i.e. all tokens in place **Pch** have the status field **f**, and the place **Pcl** contains exactly one token **i** for each cell C_i.

In the original algorithm, the sets $\mathit{Allocate}_i$, Busy_i, and $\mathit{Transfer}_i$ are extensively used in set operations. For example, the Free_i set is calculated from the above sets (when *Spectrum* is known). In our Pr/T net model we have avoided the necessity of set calculations by having a token for each channel in each cell in the place **Pch**. **Pch** can be seen as a global database for channel information but note that the cells can *not* read the status of the channels in other cells in this model because it would destroy the distributed nature of the algorithm.

The spontaneous off-hook, on-hook behaviour is modelled by transitions **Tin** and **Tout** in Figures 2 and 6 respectively. In the analysis we had to limit the number of off-hook transitions in order to get a finite reachability graph.

To make it more readable the net model has been divided into five figures. Places with the same name may appear in several different figures, but these multiple "instances" of the same place represent only *one* place in the whole net. Places which can be found in several figures are drawn with **thicker** border lines. In Figures 4 and 6 arcs extend from the word **Prqt** and to the word **Pcl**. These represent connections from/to the *places* named **Prqt** and **Pcl** and are denoted in this way only to save space.

3.1 Phase 1 - Available Channel in the Same Cell

When a connection attempt originates from cell C_i, we first check whether there are any available channels in that cell (Figure 2). This is modelled by the off-hook transition **Tin** which places a token **i** in place **Pin** when it fires. If an available channel is found in the same cell, the k' channel will be put into *busy* status. This is modelled by the transition **Tha** with the condition **condh** which selects the highest available channel and may be expressed as "*the channel k' satisfying* $\forall k_n \exists k' \; k' \geq k_n$". The transition **Tha** takes all tokens with cell index field **i** and status field **a**, i.e. all channels which are *available* in cell C_i, from the place **Pcl** and returns them unchanged except for the token with **k'** which will have a status field with the constant **b**. This is handled by the token

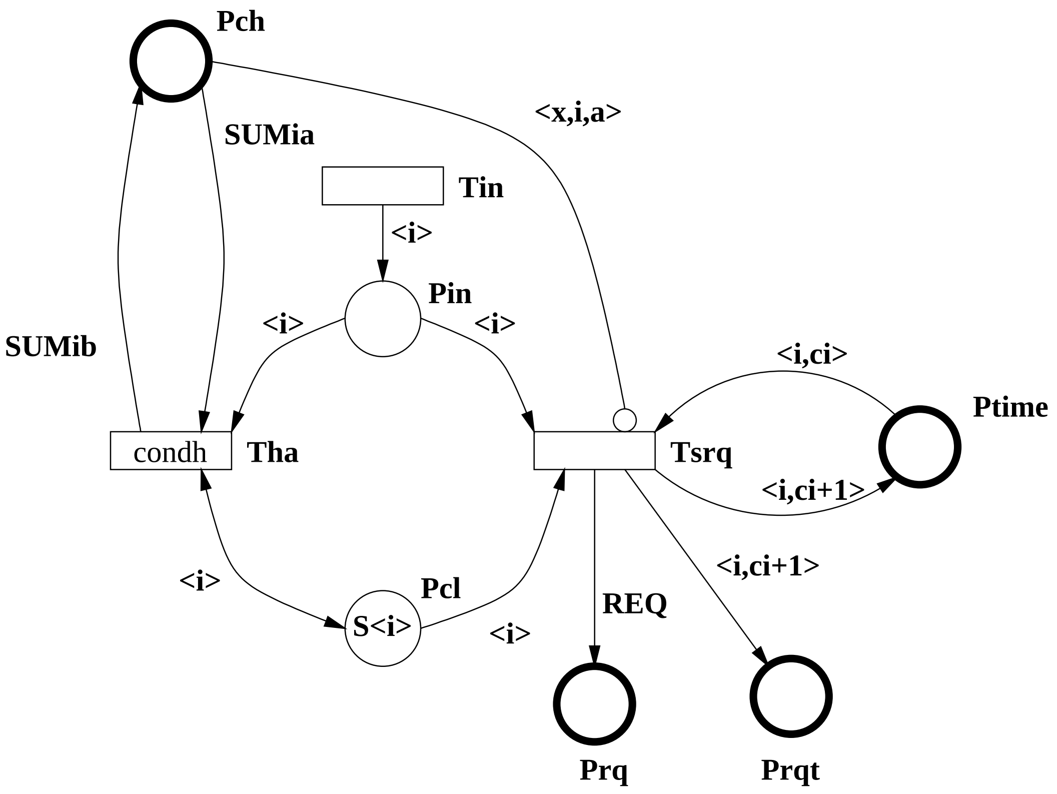


Figure 2: The internal phase.

sums **SUMia** $= \Sigma_{n \in N_k} < \mathbf{k_n}, \mathbf{i}, \mathbf{a} >$, where **a** is the status constant **a** (for *available*) and **SUMib** $= (\Sigma_{n \in N_k} < \mathbf{k_n}, \mathbf{i}, \mathbf{a} >) - < \mathbf{k'}, \mathbf{i}, \mathbf{a} > + < \mathbf{k'}, \mathbf{i}, \mathbf{b} >$, i.e. only the status of the highest order channel is changed into *busy*. This ends the connection request cycle.

If there are no available channels in cell C_i, i.e. no tokens $< \mathbf{k_n}, \mathbf{i}, \mathbf{a} >$ for any n, the transition **Tsrq** will fire. Here, the inhibitor arc with the arc expression **<x,i,a>** will be active only if there is *no* such token in place **Pch**. Thus the transitions **Tha** and **Tsrq** are complementary.

The firing of transition **Tsrq** is synonymous with sending a *request* for a new channel to all neighbours of the cell C_i. This is modelled in the Pr/T net by sending a set of tokens **REQ** $= \Sigma_{j \in Nbr_i} < \mathbf{i}, \mathbf{j}, \mathbf{ci} + \mathbf{1} >$ to place **Prq**. The term 'ci' is actually a *timestamp*. The DDCA algorithm uses Lamport's logical clocks [Lam78] for timestamps which here are modelled by tokens in place **Ptime**. Also, the transition **Tsrq** inserts a tuple $< \mathbf{i}, \mathbf{ci} + \mathbf{1} >$ into place **Prqt** and takes a token $< \mathbf{i} >$ away from place **Pcl**. The place **Prqt** is used as a bookkeeping place for pending requests from a cell C_i. This information is necessary in the next step of the algorithm, when the request with the lowest timestamp is served first.

The places **Prq** and **Prqt** represent the "communication medium" between the different cells. The neighbouring cells may read requests for a new channel from these places. The marking $\mathbf{S} < \mathbf{i} >$ of the place **Pcl** is equivalent to the sum of tokens $< \mathbf{i} >$ for $1 \leq i \leq \mathbf{N}_c$.

3.2 Phase 2 - Requesting a New Channel

Figure 3 shows the second phase of the algorithm — namely, how the requests for a new channel are received by neighbours and how the replies are returned. The transition **Trc** represents the reception of request-messages by the neighbours C_j of cell C_i. It simply forwards the tokens $< \mathbf{i}, \mathbf{j}, \mathbf{ci} >$ to the place **Prcrq** and updates the timestamps according to Rule 2 in [PSS95]. Because high-level nets are used this place represents the folding of all individual cells. In the high-level model the tokens are tagged with the number of the individual cells (**j** in the token $< \mathbf{i}, \mathbf{j}, \mathbf{ci} >$).

The replies are returned through two transitions, **Trep** and **Trep2**. The transition **Trep** ensures that the request containing the lowest timestamp gets served first, i.e. if C_j and C_i both have pending requests, C_j sends its local information first to C_i if its timestamp is higher than C_i's. If there are no pending requests from any of the neighbours C_j, the transition **Trep2** fires. This represents the sending of C_j's local information to the cell C_i. This occurs through the place **Prep** and transition **Trcrep**. The local channel information is contained in the sum $\mathbf{SUMj} = \Sigma_{n \in N_k} < \mathbf{k_n}, \mathbf{j}, \mathbf{s_n} >$.

The reply tokens sent by transitions **Trep** and **Trep2** all go to the place **Prep** because all the cell models are folded. Thus a method to distinguish the destination of the reply tokens is needed. The solution is to tag each reply token with the index number i of the cell which sent the requests. Thus the sum **SUMji** is constructed from **SUMj** as follows: $\mathbf{SUMji} = \Sigma_{n \in N_k} < \mathbf{k_n}, \mathbf{j}, \mathbf{s_n}, \mathbf{i} >$. Informally this means: "Send the channel information from cell C_j to cell C_i".

The transition **Tint** collects all the reply tokens from the neighbours, and once all are received, combines them with the tokens modelling the channel information of cell C_i. This corresponds to the algorithm step A.4 in [PSS95], where unions of the *Allocate*$_i$ set and the *Allocate* sets received in the replies are stored in the *Interfere*$_i$ sets.

The token sums are forwarded to place **Pint**, which is an "interference place" used to avoid interference between channels in neighbouring cells, i.e. the same channels may not be busy in adjacent cells. The token sum representing the replies from the neighbours C_j of cell C_i is $\mathbf{SUMjiCh} = \Sigma_{j \in Nbr_i} \Sigma_{n \in N_k} < \mathbf{k_n}, \mathbf{j}, \mathbf{s_n}, \mathbf{i} >$ whereas the local channel information is $\mathbf{SUMi} = \Sigma_{n \in N_k} < \mathbf{k_n}, \mathbf{i}, \mathbf{s_n} >$. Simultaneously the token $< \mathbf{i} >$ is placed into **Pfree**.

3.3 Phase 3 - Choosing a Channel in the Neighbourhood

Once the replies from Phase 2 have been received, the algorithm chooses a channel which will be used to support the communication session. Figure 4 shows the Pr/T net model of this phase of the algorithm.

When there is a token $< \mathbf{i} >$ in place **Pfree** one of two transitions may fire. Transition **Thf** fires if there is a token sum **SUMf** in **Pint**, i.e. when there is a channel that is free in cell C_i as well as in all its neighbours as in step A.5 in the algorithm. This channel must not be allocated to any cell in the neighbourhood. The tokens in place **Pint** model both the sets *Interfere*$_i$ and *Free*$_i$ in the algorithm, because it is more efficient to store also the tokens corresponding to the sets *Free*$_i$ in place **Pint**.

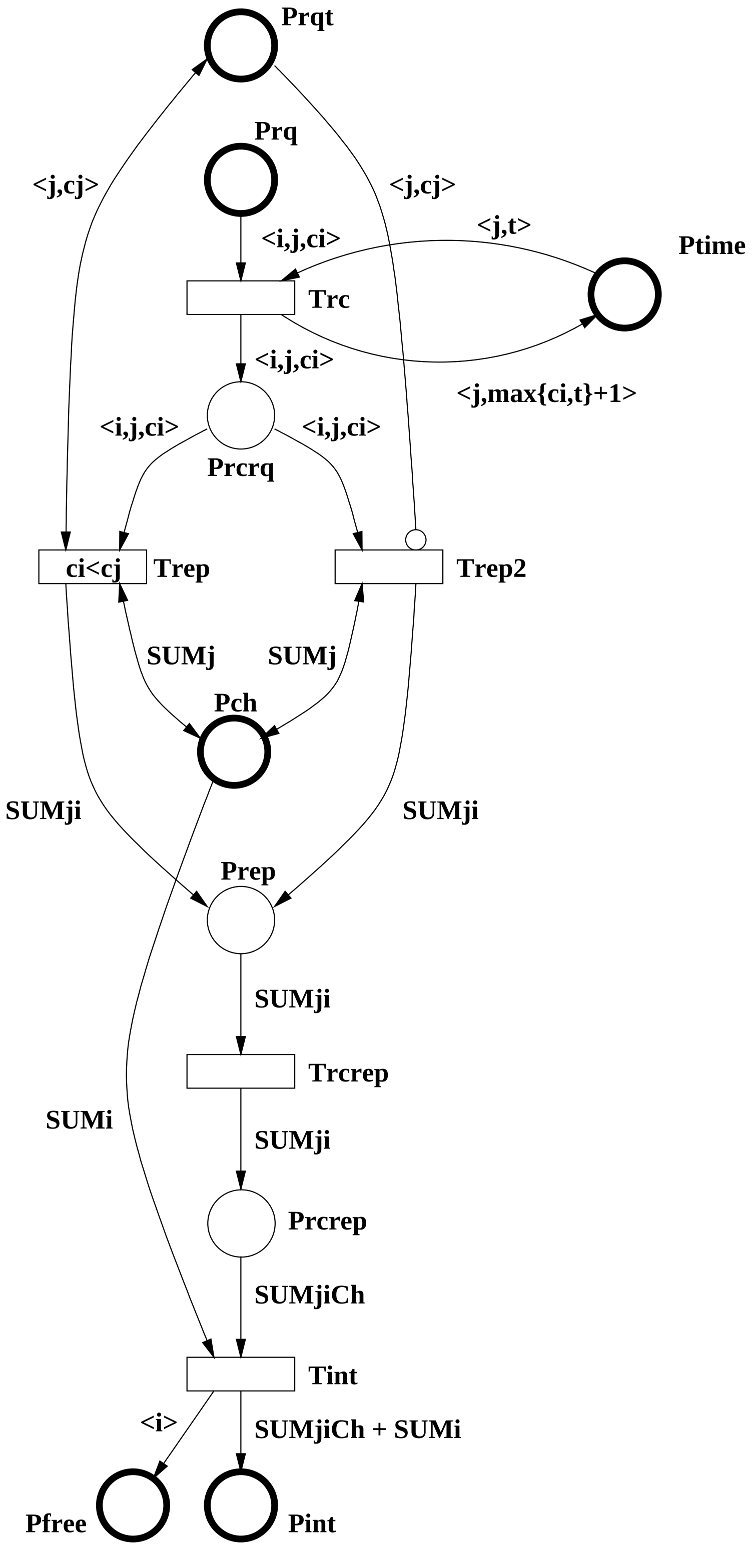


Figure 3: Sending channel requests and replies.

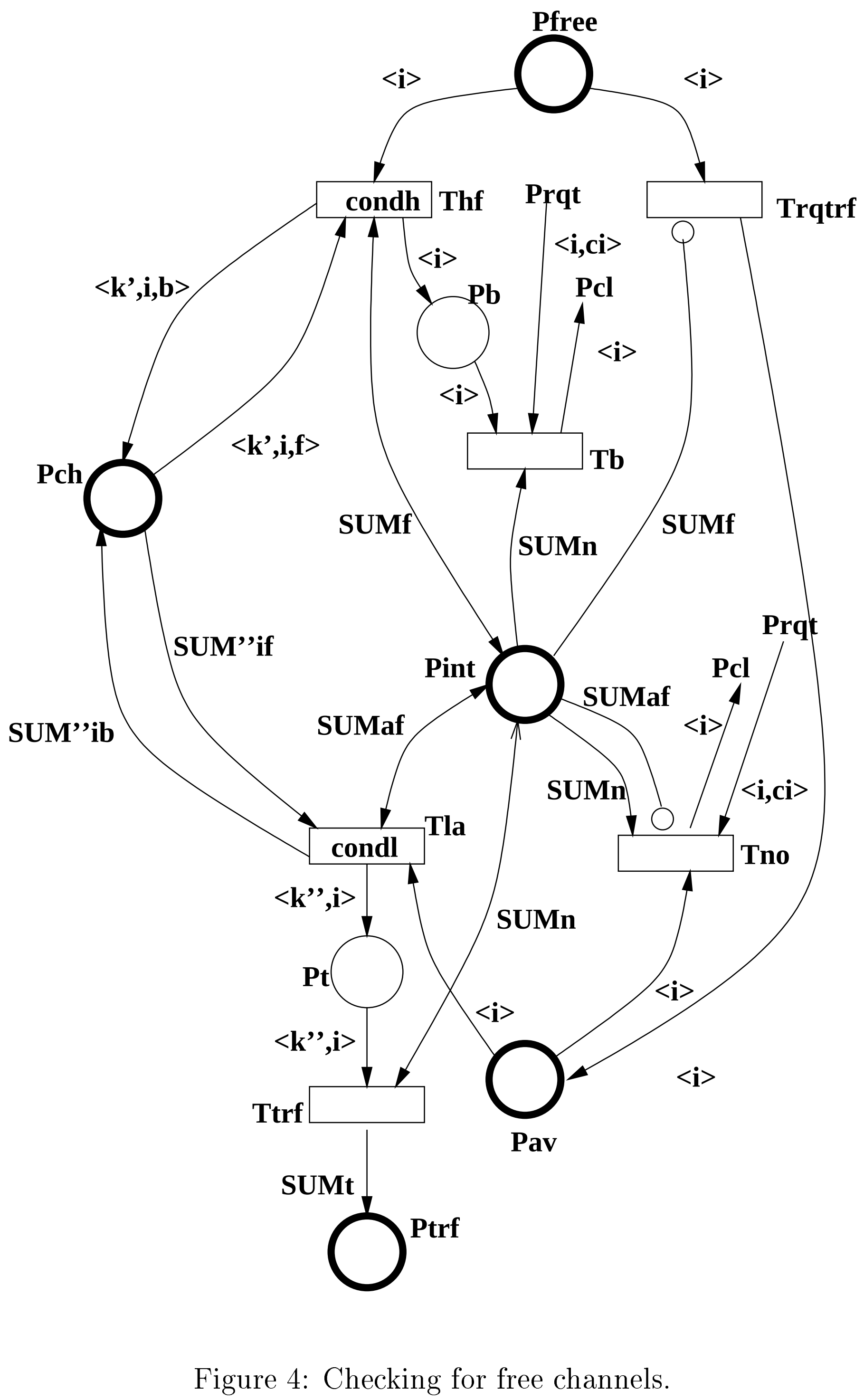


Figure 4: Checking for free channels.

Note that the tokens in place **Pint** are constructed in one transition (**Tint**) only in the model, while the sets *Interfere*$_i$ and *Free*$_i$ are calculated twice in the algorithm (also in step A.6). This is not necessary in the net model because all necessary information is available in place **Pint** and can be obtained by using suitable arc expressions.

Thus it is possible to use the token sum **SUMf** together with the transition condition **condh** denoting $\forall k_n \exists k'\ k' \geq k_n$ to select k' as in Phase 1. The testing if there is a channel which is *free* in the whole neighbourhood is handled by the arc expressions $\mathbf{SUMf} = \Sigma_{j \in Nbr_i} \Sigma_{n \in N_k} (< \mathbf{k_n}, \mathbf{j}, \mathbf{f}, \mathbf{i} > + < \mathbf{k_n}, \mathbf{i}, \mathbf{f} >)$ from place **Pint**. If the transition can fire the token $< \mathbf{k'}, \mathbf{i}, \mathbf{f} >$ is read from place **Pch**.

A successfully chosen channel token $< \mathbf{k'}, \mathbf{i}, \mathbf{b} >$ is returned to place **Pch** (note that the status changes from *free* to *busy*). The token sum **SUMf** is returned to place **Pint** to allow for an easier clean-up operation. Also, as a result of the firing of **Thf** a token $< \mathbf{i} >$ is inserted into place **Pb**.

The transition **Tb** ends the (successful) cycle for channel request by cleaning up place **Pint** (removing all tokens connected to the request), removing a pending request from place **Prqt** (token $< \mathbf{i}, \mathbf{ci} >$) and allowing a new request to be handled in cell C$_i$ by returning the token $< \mathbf{i} >$ into place **Pcl**. The sum involved in the clean-up of place **Pint** is $\mathbf{SUMn} = \Sigma_{j \in Nbr_i} \Sigma_{n \in N_k} (< \mathbf{k_n}, \mathbf{j}, \mathbf{s_n}, \mathbf{i} > + < \mathbf{k_n}, \mathbf{i}, \mathbf{s_n} >)$.

The complement of the transition **Thf** is the transition **Trqtrf**, which fires when there is no token sum **SUMf** in place **Pint**, i.e. no channel that is free in cell C$_i$ and all its neighbours. In that case the algorithm tries to transfer an allocated channel from the neighbourhood to the cell C$_i$ and the net model forwards the token $< \mathbf{i} >$ to place **Pav**.

When the control (token $< \mathbf{i} >$) is in place **Pav**, one of two complementary transitions will fire. If there is no channel that is either *available* or *free* in cell C$_i$ and all its neighbours, the transition **Tno** will fire. **Tno** cleans up the places **Pint** and **Prqt** as described above and returns the token $< \mathbf{i} >$ to place **Pcl**. In this case the connection attempt fails. The token sum modelling the checking of available or free channels in the neighbourhood is $\mathbf{SUMaf} = (x = a) \vee (x = f) \Sigma_{j \in Nbr_i} \Sigma_{n \in N_k} < \mathbf{k_n}, \mathbf{j}, \mathbf{x}, \mathbf{i} > + < \mathbf{k_n}, \mathbf{i}, \mathbf{x} >$.

If a channel is found that is either free or available in cell C$_i$ and all its neighbours, i.e. **SUMaf** exists in place **Pint**, the transition **Tla** will fire. The condition **condl** will select the lowest order of these channels (denoted by **k"**) by requiring that $\forall k_n \exists k''\ k'' \leq k_n$. The channel is subsequently assigned the *busy* status (by sending token $< \mathbf{k''}, \mathbf{j}, \mathbf{b} >$ to place **Pch**), and the control token $< \mathbf{k''}, \mathbf{i} >$ is forwarded to place **Pt**.

Finally, the algorithm will send *transfer* messages to all the neighbours of cell C$_i$ that have channel k'' *allocated* (ie. any status but *free*). In the model, however, transfer messages are sent to all the neighbours, because it is more efficient (a sum can be used over Nbr_i). The transition **Ttrf** takes tokens $< \mathbf{k''}, \mathbf{j}, \mathbf{s}, \mathbf{i} >$ from **SUMn** which have channel field $\mathbf{k''}$ and cell field $\mathbf{j}$ ($\mathbf{j} \in Nbr_i$) and constructs transfer messages from these. The transfer messages are represented by the sum $\mathbf{SUMt} = \Sigma_{j \in Nbr_i}\ if\ not(s = f)\ then < \mathbf{k''}, \mathbf{j}, \mathbf{t}, \mathbf{i} >\ else < \mathbf{k''}, \mathbf{j}, \mathbf{f}, \mathbf{i} >$. This token sum is forwarded to place **Ptrf** which is a folding of the places modelling the communication links between a cell and its neighbours. The constant **t** in the sum means that this is a transfer message. The transition **Ttrf** also cleans up the place **Pint** as described earlier. The tagging of tokens in **SUMt** with **i** is necessary to avoid confusion with other transfer attempts from the same neighbourhood.

3.4 Phase 4 - Double Checking with the Neighbours

Once a channel for transfer is found, we must double check the choice with the neighbours. It may happen that after a channel is chosen as a candidate for transfer, this channel switches to the *busy* status in some cell (before the transfer messages are received in the cell). In that case the transfer is no longer possible.

Figure 5 represents this double-checking with the neighbours. The transition **Trctrf** represents the reception of transfer messages by the neighbours of C_i. The sum in the input arc of the transition is denoted by $\mathbf{SUMtr} = \Sigma_{j \in Nbr_i} < \mathbf{k}, \mathbf{j}, \mathbf{s_j}, \mathbf{i} >$. This token sum is forwarded to place **Prctrf** which is a folded place representing all the neighbour cells but the tag j determines which cell gets the transfer request.

The two transitions **Tref** and **Tagr** take individual tokens (transfer messages destined to the cell C_j). **Tref** fires if there is a token $< \mathbf{k}, \mathbf{j}, \mathbf{p}, \mathbf{i} >$ in place **Prctrf** and a token $< \mathbf{k}, \mathbf{j}, \mathbf{s} >$ in place **Pch** such that $\mathbf{s} = \mathbf{b} \vee \mathbf{s} = \mathbf{t}$ (denoted by **s=b,t** in Figure 5). The value of the variable $\mathbf{p}$ in the arc expression $< \mathbf{k}, \mathbf{j}, \mathbf{p}, \mathbf{i} >$ is not important. This models the case that the channel k in a cell C_j ($j \in Nbr_i$) has status *busy* or *transfer*. In that case the transfer of the channel k from cell C_j is unsuccessful and the channel status in cell C_j will be unchanged (the token is returned unchanged to place **Pch**).

Tagr models the case when the channel k in a cell C_j is *available* or *free*. The annotation **s=a,f** in transition **Tagr** denotes the condition $\mathbf{s} = \mathbf{a} \vee \mathbf{s} = \mathbf{f}$. In that case, the transfer of the channel k from cell C_j is successful (the token $< \mathbf{k}, \mathbf{j}, \mathbf{s} >$ is returned to place **Pch** with status field $\mathbf{t}$).

The unsuccessful transfer of a channel k from cell C_j is coded into a token $< \mathbf{k}, \mathbf{j}, \mathbf{b}, \mathbf{i} >$ that is forwarded to place **Pstr**. Similarly, the successful transfer of a channel is coded into a token $< \mathbf{k}, \mathbf{j}, \mathbf{f}, \mathbf{i} >$ that is placed into **Pstr**. Thus the $\mathbf{b}$ in token $< \mathbf{k}, \mathbf{j}, \mathbf{b}, \mathbf{i} >$ does not mean the status *busy* but is just a convenient way of coding that the transfer request was unsuccessful in cell C_j ("refuse message"). Likewise the token $< \mathbf{k}, \mathbf{j}, \mathbf{f}, \mathbf{i} >$ means that the transfer request for cell C_j was successful ("agree message").

Once all the neighbouring cells have either agreed or refused a channel transfer, the transition **Tstr** can fire. This transition models the sending of answers from the all the neighbours (**SUMtr**) to the cell C_i (place **Ptrep**). The sum makes certain that all the cells have replied before the cell C_i will start evaluating the answers. Note that the interpretation of **SUMtr** is different from the token sums with the same name annotating arcs around transition **Trctrf** although the token structures are the same.

3.5 Phase 5 - Cleaning up

Once the "agree" or "refuse" messages have arrived from the neighbours of the cell C_i, i.e. the token sum **SUMtr** has arrived at the place **Ptrep**, one of two complementary transitions can fire (Figure 6). If there is even a single "refuse" message in the replies to cell C_i, i.e. the condition $\exists \mathbf{s_j}\ \mathbf{s_j} = \mathbf{b}$ (annotated by **exist b**) is satisfied meaning that

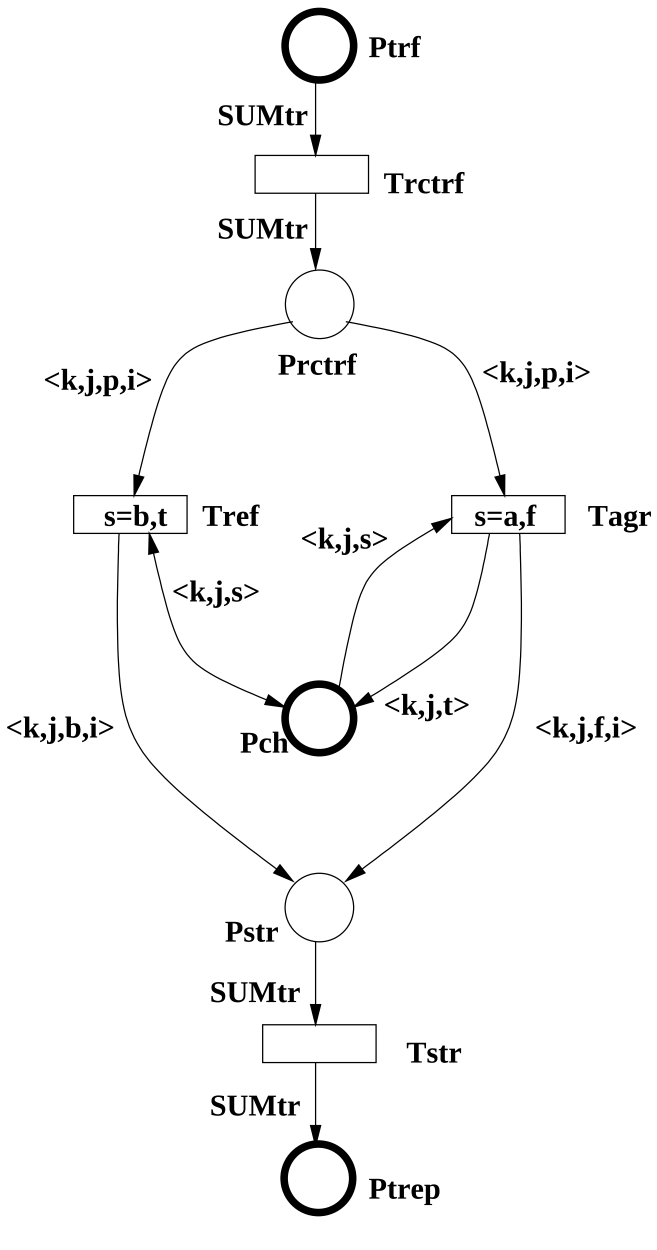


Figure 5: Double checking the chosen channel.

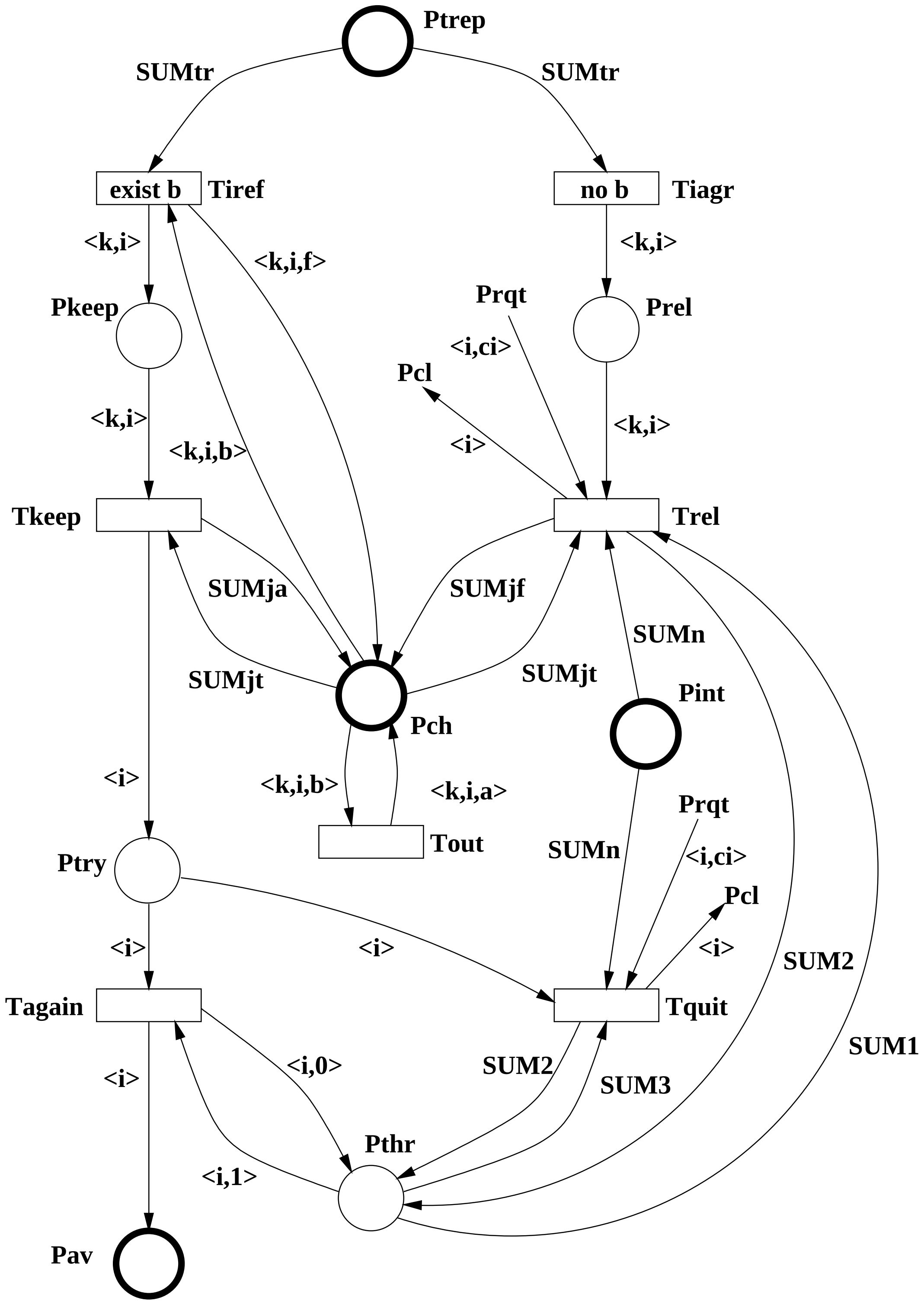


Figure 6: Cleaning up.

there is at least one token $<\mathbf{k},\mathbf{j},\mathbf{b},\mathbf{i}>$ in place **Ptrep**, the transition **Tiref** will fire. This transition sends a token $<\mathbf{k},\mathbf{i}>$ to the place **Pkeep**. Also, the channel k in cell C_i that was set to *busy* status will be assigned the *free* status (token $<\mathbf{k},\mathbf{i},\mathbf{b}>$ in place **Pch** is replaced by $<\mathbf{k},\mathbf{i},\mathbf{f}>$).

The place **Pkeep** models the fact that the cell C_i is sending "keep" messages to the neighbourhood. **Tkeep** sets those channels that are in the *transfer* status back into *available* status in the cells C_j. This is handled by the two sums **SUMjt** $= \Sigma_{j\in Nbr_i}<\mathbf{k},\mathbf{j},\mathbf{s_j}>$ and **SUMja** $= \Sigma_{j\in Nbr_i}\ if\ (s_j = t)\ then <\mathbf{k},\mathbf{j},\mathbf{a}>\ else <\mathbf{k},\mathbf{j},\mathbf{s_j}>$.

If no "refuse" messages are returned, i.e. the condition $\neg\exists\mathbf{s_j}\ \mathbf{s_j} = \mathbf{b}$ (annotated in transition **Tiagr** by **no b**) is satisfied meaning that no token $<\mathbf{k},\mathbf{j},\mathbf{b},\mathbf{i}>$ is found in place **Ptrep** the transition **Tiagr** will fire denoting a successful transfer of a channel. This transition sends a token $<\mathbf{k},\mathbf{i}>$ into place **Prel**, which models the sending of "release" messages from the cell C_i to the neighbourhood. The transition **Trel** cleans up the place **Prqt** and returns the token **¡i¿** to place **Pcl**. It will also set those channels that are in the *transfer* status in the neigbour cells C_j into the *free* status by replacing token sum **SUMjt** with **SUMjf** $= \Sigma_{j\in Nbr_i}\ if\ (s_j = t)\ then <\mathbf{k},\mathbf{j},\mathbf{f}>\ else <\mathbf{k},\mathbf{j},\mathbf{s_j}>$.

The original algorithm has a *THRESHOLD* parameter that tells how many times the algorithm will try to transfer a channel before giving up. This behaviour is modelled by places **Ptry** and **Pthr** and transitions **Tagain** and **Tquit** in Figure 6. The place **Pthr** initially contains n_{thr} tokens of the form $<\mathbf{i},\mathbf{1}>$ where $n_{thr} = THRESHOLD$. Each time an unsuccessful transfer takes place (and transition **Tagain** fires), a tuple $<\mathbf{i},\mathbf{1}>$ in place **Pthr** is replaced with $<\mathbf{i},\mathbf{0}>$ and control is given to place **Pav** in Figure 4 (a new transfer attempt is started).

If there are only $<\mathbf{i},\mathbf{0}>$ tuples left in place **Pthr**, the transition **Tquit** will fire ending the transfer attempt cycle. In this case we replace all the tuples of the form $<\mathbf{i},\mathbf{0}>$ (SUM3) with $<\mathbf{i},\mathbf{1}>$ (SUM2).

If the transfer attempt was successful (**Tiagr** fires) then **Trel** replaces all the tokens of the form $<\mathbf{i},\mathbf{r}>$ (SUM1) with tokens $<\mathbf{i},\mathbf{1}>$ (SUM2). The sums involved in this *THRESHOLD* part of the algorithm are:
SUM1 $= \Sigma_{j\in\{1\ldots n_{thr}\}} <\mathbf{i},\mathbf{r}>$,
SUM2 $= \Sigma_{j\in\{1\ldots n_{thr}\}} <\mathbf{i},\mathbf{1}>$, and
SUM3 $= \Sigma_{j\in\{1\ldots n_{thr}\}} <\mathbf{i},\mathbf{0}>$.

The transition **Tout** models the spontaneous on-hook behaviour. It may fire at any time.

4 Reachability Analysis

The general Petri net model shown here was analysed with the Maria analyser [Mar] and in that process a few changes had to be made to the model because the input language of Maria does not support inhibitor arcs and some expressions. The inhibitor arcs were used in the general model only to avoid that the figures would be too cluttered - inhibitor arcs can always be replaced by other constructs. In the Maria net model the complexity of the net is much larger than in this abstract version. This is mainly due to dividing some transitions. It does not add to the size of the reachability graph but makes the net model less readable. This is not so important because the net description language of Maria is designed as an intermediate language although it has very powerful high-level features and a complete type system. Also for Petri net modelling front-ends are planned which handle the interface to the user, for example using the well-known Design/CPN.

The main effort in this work is without doubt in the modelling (comprising many man-months and going through many versions), but of course it is also important to analyse some properties of the model (and the algorithm). This model was also used in the development work of the Maria analyser. The results given here are from the first experimental version of the analyser and are only representative for the modelling power of this first version.

Creating a Maria model was also useful to detect errors in the more general net model. Even before the analysis was performed we detected a few mistakes by just using the simulation facility of **marde**, the Maria debugging tool. Especially using high level nets which are heavily folded the inscriptions in the net may be so complex that it is difficult to see errors just by inspecting. In many respects the modelling process is analogous to program design and this is certainly the case for Maria which has a very developed data type system and a very powerful expression syntax.

As an example of the Maria input language (version 0.1) the transition **Trep** in Figure 3 is shown:

```
// Define a new place for the replies of type reqreply.

typedef struct{channel k; cell j; state s; cell i} reqreply;
place Prep reqreply;

trans Trep
{ channelstate s; }
in {
 Prcrq:  {i,j,ci};
 Prqt: {j,cj};
 Pch: channel k (k>=1): {k,j,s[k]};     // SUMj
}
out {
 Prqt: {j,cj};
 Pch:  channel k (k>=1): {k,j,s[k]};    // SUMj
 Prep: channel k (k>=1): {k,j,s[k],i};  // SUMji
}
gate ci<cj;
```

Note the coding of the sums. For **SUMj** the last line in the **in** arc definition means that from place **Pch** all tokens {k,j,s[k]} are taken for all channels k if $k >= 1$. The value of j is determined from the places **Prcrq** and **Prcrq**. The **SUMji** is constructed from the available values of k, j and s[k] with the addition of the value of i determined from place **Prcrq**.

Although Maria has many powerful constructs which make the modelling of distributed algorithms easy there are a few additional features which would be most useful like quantifiers. It is often necessary to express that a transition should fire (or not fire) if there *exists* a token of a certain kind or if *all* tokens have a certain property. Thus existential and universal quantifiers would be quite useful, especially in modelling protocols like this one.

In the Maria model of the DDCA algorithm we could not yet make the net completely parameterised, i.e. make its structure insensitive to changes in the cell configuration. Thus some transitions had to be split into several transitions - one for each cell. We hope to add features to Maria to make a completely parameterised model possible in the future.

An exhaustive reachability analysis for a complex configuration with a large number of channels and many concurrent channel requests (call attempts) is very difficult because the size of the reachability graph explodes. It is therefore necessary to check the function of the DDCA algorithm for relatively small examples. We analysed two special configurations, one with seven cells and one with three cells. In both cases the number of channels was only two, but the number of concurrent channel requests varied.

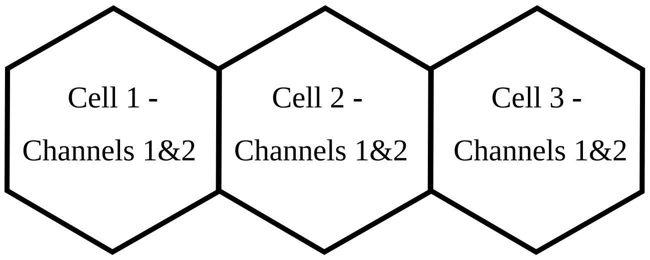


Figure 7: The three cell configuration.

Whereas the general model (presented in Figures 2–6) has 25 transitions the Maria model for three cells and two channels (Figure 7) has 66 transitions and 41 places. The model with seven cells in Figure 1 has 63 transitions and 46 places.

We considered a set of three cells each having two channels as in Figure 7. Initially, all the channels in every cell were *free*. The size of the reachability graph was limited by constraining the number of concurrent connection attempts to one, two and three respectively. Note that there could be a maximum of one connection attempt in any given time at a given cell (ie. no concurrent connection attempts from the *same* cell).

The results of the reachability analysis are given in Table 1. The analysis uncovered no deadlocks in the execution of the algorithm. Moreover, a Linear-Time Temporal Logic formula was verified ensuring that no neighbouring cells can use the same channel at the same time. In effect, the property was verified by considering the marking at place **Pch** in

	three cells		seven cells	
no. of concurrent requests	number of markings	number of arrows	number of markings	numbers of arrows
1	105	163	4551	19387
2	4557	11490	-	-
3	18528	48603	-	-

Table 1: Statistics for the reachability analysis.

each reachability marking. This was the key property that the protocol had to satisfy to be correct.

The corresponding analysis for seven cells generated a much larger number of states already with one request and was very slow. For two requests the analysis was stopped after a few hours. The Maria analyser is actively developed and new versions with better efficiency and new features appear at a steady pace. The most important improvement in the analysis will, however, be in using reduction methods and other techniques for handling large reachability graphs.

Making relatively small changes in the net model influenced the reachability analysis substantially and the figures in Table 1 should be seen as indicative only and bound to change.

5 Further Work

The basic algorithm given in [PSS95] has been since augmented to contain mobile base stations [PN99]. The natural continuation of this work is to extend the model to the case where the base stations are mobile. Work on this extension is already going on and a first version will be reported in [OHT].

The basic DDCA algorithm has been modelled and analysed in this work using the Maria reachability analyser, but it still seems to be possible to gain from small changes in the model in order to get a more efficient analysis.

Much also remains to be done in developing the power of the expressions in Maria and also in the analysis methods. The expressiveness is not only needed in order to keep models small and readable but also to make them parameterised, i.e. easy to alter for different configurations. The inclusion of quantifiers would be very useful in this kind of models and for the DDCA algorithm we would also need the possibility to choose “maximal” and “minimal” token from a place. This should be easy in Maria because the data types are totally ordered [Mar].

It is very difficult to analyse this model for big configurations, but already in these simple cases some mistakes in the model have been found and the use of an analyser, even in debugging (or simulation) mode, is considered very useful for the modelling process.

The further development of **marde**, the debugging tool therefore would be important. Especially the inspection of paths in the reachability graph is useful in checking if the model works as intended.

The reachability analysis results must also be investigated in order to check how the analysis can be made more efficient. The Maria analyser will soon have also reduction methods implemented (the same as in PROD) and the influence of these could be interesting.

6 Conclusions

An algorithm for distributed dynamic channel allocation for mobile computing has been successfully modelled using Predicate/Transition nets. The general high-level Petri net model is quite compact with *only 22 places and 25 transitions* and it can be used for *any number of cells and channels with any configuration.* It would be difficult to achieve without a very expressive high-level net language. It was a nontrivial undertaking to model the DDCA algorithm and this is the first complete model of the algorithm.

The algorithm has been analysed by the Maria analysis tool and has been found deadlock-free for some simple configurations. Moreover, the algorithm ensures that the same channel is never allocated in contiguous cells for the verified cases. Further analysis should be carried out, but the algorithm seems to be correct and implementable in real systems.

References

[CC96] M. Cheng and J. Chuang. Performance evaluation of distributed measurement-based dynamic channel assignment in local wireless communications. *IEEE Journal on Selected Areas in Communications*, 14(4):698–710, May 1996.

[FZ94] George H. Forman and John Zahorjan. The challenges of mobile computing. UW CSE Tech Report #93-11-03, University of Washington, Computer Science & Engineering, March 1994.

[Gen87] Hartmann J Genrich. Predicate/Transition nets. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency, Part I*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, Berlin, 1987.

[Lam78] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lee93] W.C.Y. Lee. *Mobile Communication Design Fundamentals.* Wiley, 1993.

[Mar] Maria - a Modular Reachability Analyser, Web Home Page. http://www.tcs.hut.fi/.

[OHB99] Leo Ojala, Nisse Husberg, and Simo Blom. Modelling a Distributed Dynamic Channel Allocation Algorithm for Mobile Computing Using Predicate/Transition Nets. In *Proc. of 1999 IEEE Int. Conf. on Systems, Man, and Cybernetics, October 12-15, 1999, Tokyo, Japan.* IEEE, 1999.

[OHT] Leo Ojala, Nisse Husberg, and Teemu Tynjälä. Modelling a Distributed Wireless Channel Allocation Algorithm for Cellular Systems with Mobile Base Stations using Predicate/Transition Nets. Accepted for presentation at SCI2000, Orlando, Florida, 23-26 July 2000.

[PN99] Ravi Prakash and Sanket Nesargi. Distributed wireless channel allocation in networks with mobile base stations. In *Proceedings of INFOCOM'99*, pages 592–600, March 1999.

[PSS95] Ravi Prakash, Niranjan G. Shivaratri, and Mukesh Singhal. Distributed dynamic channel allocation for mobile computing. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 47–56, 1995.

Modelling and Initial Analysis of the Resource Reservation Protocol using Coloured Petri Nets

María E. Villapol and Jonathan Billington
Cooperative Research Centre for Satellite Systems
University of South Australia
SPRI Building, Mawson Lakes, Adelaide SA 5095
Tel: 08 8302 3371 Fax: 08 8302 3873
email: maria@spri.levels.unisa.edu.au and email: jb@spri.levels.unisa.edu.au

Abstract

The *Resource Reservation Protocol (RSVP)* conveys Quality of Service information along the path of a data flow. It is intended to support the new emerging Internet applications which require a guaranteed level of service to achieve their functionality. The aim of this paper is to use Coloured Petri Nets to model some features of the protocol over a simple unicast network. Initial analysis of the model shows that it is working correctly, in other words, as described in RSVP specification document. However, further analyses are required to validate the CPN model.

1. Introduction

Traditionally, Internet applications, such as *File Transfer (FTP)* and *Telnet*, use a best-effort service with no service guarantees [6]. In the last decade, however, new applications have emerged. These applications, such as multimedia and real-time applications, generate not only data but also images, video and voice. They require different levels of *quality of service* (QoS) regarding, for example, delay and throughput. Thus, the *Internet Engineering Task Force (IETF)*[1], a volunteer organisation that discusses operational and technical problems of the Internet, has worked on extending the Internet architecture to support such new applications.

Although the original plan of the IETF was to create an unified model for the Internet, today several service models have been developed [2] [3]. One of the proposals is the *Internet Integrated Service Model (IntServ)* [3]. IntServ supports not only best-effort applications but also real-time applications. In addition, the *Resource Reservation Protocol (RSVP)* [4][7][18] conveys QoS parameters and sets up QoS information along the path of a data flow.

Formal methods provide techniques to support the design and maintenance of communication protocols [1]. They have already been applied to protocol engineering activities [15][16]. *Coloured Petri Nets (CPN)* [10][11] are a formal technique with a solid mathematical foundation which has been used for modelling many systems such as communication protocols [12].

The authors have found that formal techniques have been seldom applied to the Internet protocol engineering activities. In this paper, CPNs, with the aid of a software tool called Design/CPN [14], are used to model the operation of RSVP and to analyse it based on the RSVP specification [4]. This initial model includes some basic features of RSVP working on a simple network topology supporting unicast traffic (ie a sequence of packets travelling from a sender to a single receiver).

[1] See IETF home page at *http://www.ietf.org.*

The paper has been organised as follows. Section two presents an overview of RSVP which includes its characteristics, operation and relationship with other protocols. Section three includes a description of the CPN model of RSVP. In addition, the assumptions and requirements taken into account for this model are presented. The model is analysed in section four. It includes some simulation results and an initial state space analysis. In addition, previous CPN models of the protocol are compared in terms of the size of the occurrence graphs. Finally, section five concludes this paper.

2. Resource Reservation Protocol (RSVP) Overview

2.1 Characteristics

RSVP is a signalling protocol developed to create and maintain resource reservations on each link along the transport path. It is also used by a host to request a particular QoS for each application.

The design principles of RSVP are outlined in [18]. A detailed description of those principles is beyond the scope of this paper. However, the main characteristics of RSVP, which are closely related to those principles, are summarised as follows:

- **Receiver-based:** receivers initiate the resource reservation along the path between the source and destination of a data flow, since receivers know the resource availability and limitations [4][18].
- **Soft-state reservations:** the reservations along a path are considered non permanent, so they must be refreshed periodically. If a reservation is not refreshed before a timeout occurs, the reservation is cancelled, so, the reservations may adapt to dynamic routing changes and the QoS reserved for a flow may be changed at any time.
- **Flow oriented:** RSVP reserves resources on a flow basis. A *data flow* is a distinguishable packet stream which results from a single user/application activity and requires the same QoS.
- **Unidirectional:** RSVP reserves resources in one direction.
- **Heterogeneous receivers:** each receiver requests resources to support its own QoS requirements.
- **Support of multicast sessions:** RSVP makes resource reservations for both unicast and multicast applications.

2.2 Architectural Overview

Figure 1 shows the TCP/IP protocol stack extended to support QoS provision. It includes several protocols which support the transfer of data from the applications. RSVP is located on top of IP[1] at the level of a transport protocol. It does carry control information (ie RSVP signalling messages) intended to create, manage, and remove reservations associated with user data. A description of these protocols, apart from RSVP, is beyond the scope of this paper; however there is a wide range of literature related to the Internet protocols (eg see [6][8]).

[1] RSVP may also run on top of UDP as explained in RFC 2205 [4]. However, for simplicity, it is not shown in figure 1.

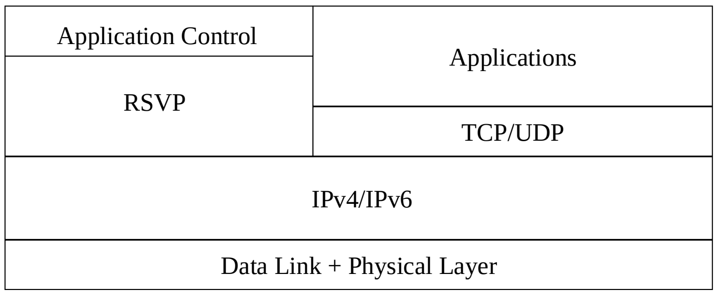


Figure 1: RSVP in the TCP/IP architecture.

Figure 2 shows the software architecture of a host and a router which supports RSVP. The host and router systems are the same except that the application block in the host is replaced by a routing block in the router.

An application must be able to interact with RSVP in order to communicate traffic characteristics of the data flow and their QoS requirements. RSVP signalling messages will be encapsulated into IP packets and travel hop-by-hop from the sender to the receiver(s). At each node, if the node supports RSVP, the message must be processed. Some nodes in the network may not be able to process RSVP messages, thus they will be forwarded without further processing [4]. At each router, RSVP interacts with a routing protocol to obtain the IP address of the next hop on the route of the data flow.

Traffic control, which includes the classifier, packet scheduler and admission control [3], is responsible for allocating network resources according to QoS information carried into RSVP messages. Thus, the *classifier* classifies IP packets according to a set of service classes and assigns them to different queues. The *packet scheduler* determines which of the set of IP packets will be served next. Finally, *admission control* decides whether there are sufficient resources available to grant the requested QoS for a data flow.

RSVP may also communicate with a *policy control* component which decides if the user requesting a reservation is permitted to do so. Policy control mechanisms may involve, for example, the identity of the user and application, traffic and data rate requirements, and security considerations [7].

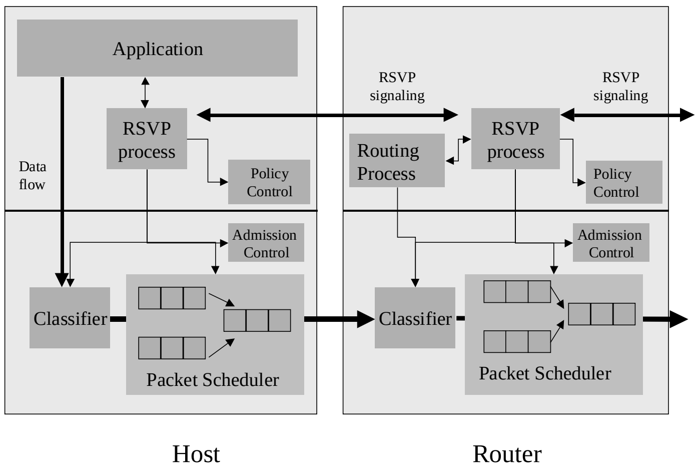


Figure 2: QoS software architecture.

2.3 RSVP specification

Sessions and dataflows

A *session* is a *data flow* with a particular destination and transport-layer protocol and is identified by an IP destination address (unicast or multicast) of the data flow, IP protocol ID, and destination port (optional) (eg UDP/TCP destination port field) [4].

Traffic and QoS parameters

The RSVP specification defines a reservation request in terms of a *filter specification (filter spec)* and a *flow specification (flow spec)* [4]. The former defines the sequence of packets or data flow to receive the QoS specified in a flow specification. A filter specification together with a session ID is used to identify a flow which will receive the QoS. The latter defines a desired QoS for the flow and defines its traffic characteristics. It includes a service class, a *Reservation specification (Rspec),* and a *Traffic specification (Tspec).* A *traffic specification (Tspec)* defines the traffic characteristics of the flow, for example, the peak rate. A *reserve specification (Rspec)* defines the reservation (ie. desired QoS) characteristics of the flow, for example, the service rate. The formats of a Tspec and Rspec are not defined by the RSVP specification.

A filter specification is used by the classifier to assign the data flow to a queue and a flow specification is used by the packet scheduler to allocate the corresponding QoS and to schedule packets based on their traffic characteristics.

Soft state

RSVP soft state reservations deal with occasional loss of RSVP messages and route changes at any point on the path of a data flow. Thus, reservation and path states set up by RSVP along the route of a data flow must be refreshed periodically, otherwise they will be removed. The *refresh timeout* determines when a refresh message must be generated, while the *cleanup timeout* determines the maximum period of time that a node waits to receive a refresh message, before it removes the associated state information.

RSVP operation

RSVP uses several messages in order to create, maintain, and release state information for a session between one or more senders and one or more receivers (see fig. 3). Sequences of packets travelling in opposite directions may follow different routes. In RSVP, reservation requests travel from receivers to the sender(s), in the opposite direction to the user data flow for which such reservation is being requested. *Path Messages* are used to set up a route for the reservation requests along the same path of the corresponding data flow. They set up and maintain path information (eg the IP address of the previous host and traffic characteristics of a data flow).

A path refresh is the result of either a state refresh timeout or the modification of a path state (as mentioned before). Once a path is established, a node periodically (ie every refresh timeout period) sends path refresh messages (ie *Path messages*).

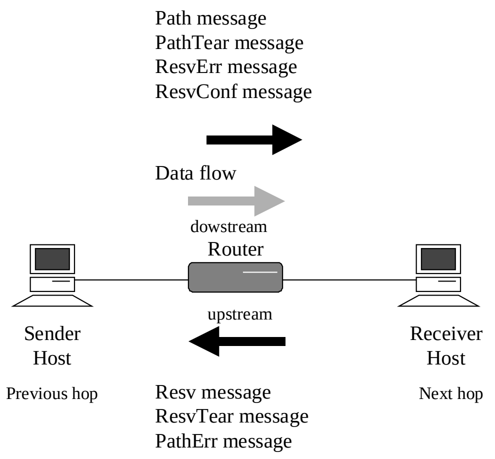


Figure 3: Flow of RSVP messages.

Resv messages travel upstream from the receiver(s) to the sender. They carry reservation requests (e.g. for bandwidth and buffers) used to set up reservation state information along the route of a data flow. At any intermediate node, a reservation request may be rejected by Admission Control because there are not sufficient resources to guarantee the requested QoS. Also, reservation requests which arrive at a router are merged. The aim of merging is to control the overhead of reservation messages by making them carry more than one flow and filter specification [4][18]. Thus, the effective filter and flow specifications, which are carried in a reservation message, are the result of merging reservations from several requests.

A reservation refresh is the result of either a state refresh timeout or the modification of a reservation state (as mentioned before). Like path states, reservation states need to be refreshed. Thus, a receiver periodically sends reservation refresh messages (ie Resv messages) to the sender.

RSVP tear down messages are intended to speed up the removal of path and reservation state information from the nodes. They may be triggered because a state timeout occurs (as explained before) or an application wishes to finish a session (ie service preemption). A *PathTear message* travels downstream from a sender to the receiver(s) and deletes any path state information and dependent reservation associated with the session and sender. A *ResvTear message* travels from a receiver to a sender and removes any reservation information state associated with one or more data flows.

In addition, there are two error messages, *Path Error* and *Resv Error*, which are used to report problems associated with processing or installing Path/Resv information or to report administratively defined constraints imposed on the setup of a reservation state [7]. They travel hop-by-hop from the point where the error was found.

Optionally, a receiver may ask for a confirmation for its reservation by including a RESV conformation object[1] in the Resv message (ie reservation request). A *ResvConf message* is used to notify the receiver that the reservation request was successful. In the simplest case, a ResvConf message is generated by the sender (see fig. 3)[2].

[1] A RSVP message comprises a message header and a set of objects. Objects contain information necessary to process the message at each RSVP node it arrives [4][7].

[2] For more information about ResvConf message see [4]

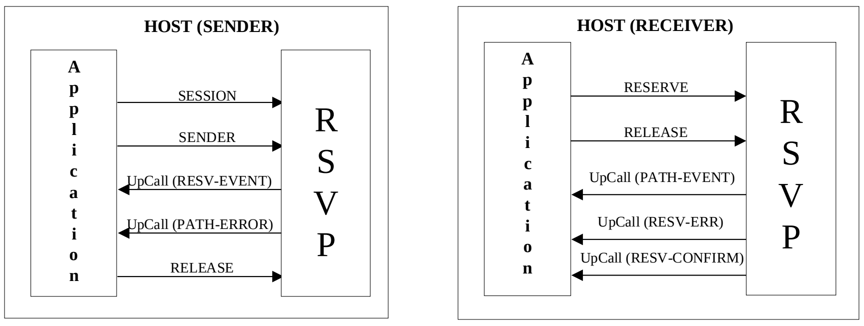


Figure 4: Application/RSVP interface.

2.4 Application/RSVP interface

An application which requires QoS guarantees from the network must communicate with RSVP and provide the QoS and traffic characteristics of the data flow. Braden et al [4] describes a generic interface between an application and RSVP which includes the following calls (see fig. 4):

1. **Session:** creates a RSVP session.
2. **Sender:** is used to define or to modify the characteristics of a data flow (eg peak data rate). The first call triggers RSVP to send Path Messages. Future calls will make RSVP send modified Path Messages (eg including a different TSpec).
3. **Reserve:** is used to create or to modify a resource reservation (eg the size of a buffer). The first call will initiate the transmission of Resv Messages. Future calls will change the existing reservations.
4. **Release:** is used by the sender application leaving a session. It removes any existing path and corresponding reservation state information. A receiver application uses this call to remove one or more reservations for the session.

Also, some upcalls have been defined. They indicate an error or event:

5. **Path_Event:** indicates that the first Path message for the session has been received or the path has been changed.
6. **Resv_Event:** indicates that the first Resv message for the session has been received or the reservation has been changed.
7. **Path_Error:** indicates a Path Error message has arrived or a local error has occurred.
8. **Resv_Error:** indicates a Resv Error message has arrived or a local error has occurred.
9. **Resv_Confirm:** indicates that a ResvConf message has been received.

3. CPN Model of RSVP

RSVP is a complex protocol whose features have been outlined in this paper. In order to facilitate the design and debugging of the CPN model, an incremental approach was adopted. Several versions of the model were created, gradually including more features. The last version of the model comprises the features of the protocol which have been modelled in this paper. It is described as follows. Then, section 3.4 describes briefly previous versions of the model.

3.1 Requirements and assumptions

The CPN model of RSVP has been developed based on the operation of the protocol given in section two and in the protocol specification [4]. The network topology comprises two hosts (sender and receiver) and a single router between them (figure 3). Also, the following assumptions have been made:

1. Just one session is necessary to study the functional behaviour of RSVP, since RSVP treats each session independently [4].

2. In order to simplify the modelling of RSVP, only one data flow may flow between the sender and receiver. Future models of RSVP will consider more than one application and/or sender, so more than one flow.

3. Traffic and QoS parameters of a data flow are carried in Path and/or Resv messages as RSVP message objects [4][7]. Any change in the value of those parameters must be propagated to all nodes along the way of the data flow. Each node makes the appropriate adjustment in the path and/or reservation states. However, for simplicity, the model presented in this paper is focused on RSVP messages which do not change installed states at the nodes but keep them on place. After checking that the current model is working as specified, it will be extended to considered traffic and QoS changes.

4. Since RSVP is running on top of IP, which is not a reliable protocol, messages may be lost. As mentioned before, RSVP Path and Resv refresh messages deal with occasional loss of RSVP messages. Although, message losses have not been considered in the model presented here, the mechanisms for dealing with that are modelled. In future, the model will be extended to include message losses.

5. An RSVP network may include several nodes (eg sender and receiver hosts and routers). The model presented in this paper is based on a simple topology in order to facilitate analysis of the model. Thus, route changes are not possible. However, RSVP Path refresh messages deal with route changes. Future models of RSVP will consider more complex topologies which may support route changes.

6. The RSVP specification [4] only suggests the basic functions which may be performed for an Application/RSVP interface. Protocol implementors may not only create real detailed interfaces but also define the sequence of Application/RSVP calls. Thus, it is necessary to make some assumptions about sequences of Application/RSVP calls (see section 2.4). In particular, it has been assumed that the receiver application will send a Reserve call (so the RSVP entity will start sending Resv messages) after it receives a PathEvent call indicating that the first path message has arrived.

7. Since there are few reasons why a Path must be rejected [4], it has been assumed that a path can not be rejected. Therefore, Path Error messages will not be required. This simplifies the initial analysis of the model. Future models will include Path Error messages.

8. This initial model of RSVP is intended to include only the compulsory features of the protocol. Thus, optional features, such as reservation confirmation, are not considered.

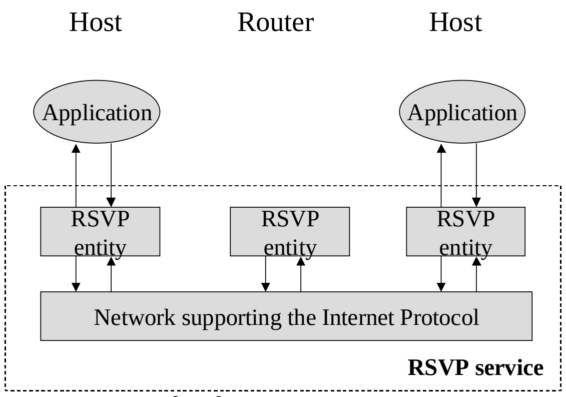


Figure 5: Top-level structure.

3.2 Top-level structure

Figure 5 shows the top level architecture of RSVP, where there is only one router in the network. An application may request a particular QoS by interacting with the RSVP entity (see section 2.4) which attempts to reserve the necessary resources in the router, by signalling to the router's RSVP entity and then to the sender's RSVP entity.

3.3 Detailed model

3.3.1 General structure

The detailed model of RSVP consists of ten pages (figure 6). The hierarchical view has been designed based on the network topology and the functionality of RSVP entities at each node. The subpages (eg Resv Setup) represent the more complex functions.

The top level model of RSVP shows the interaction between the RSVP nodes (figure 7). The three transitions of the top level of the model (ie Sender, Router, and Receiver) represent the RSVP entities at each node and are shown as hierarchical transitions.

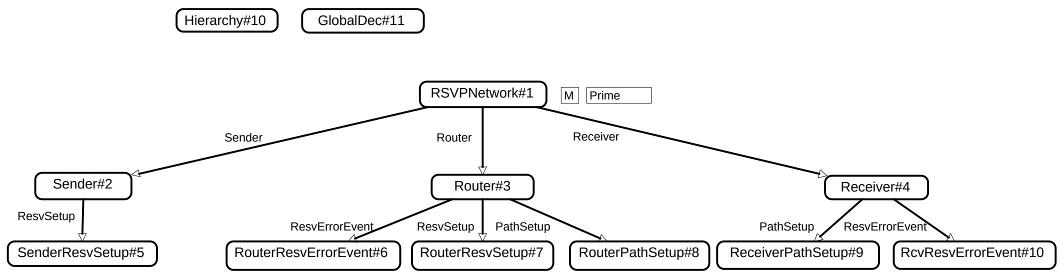


Figure 6: CPN hierarchy page.

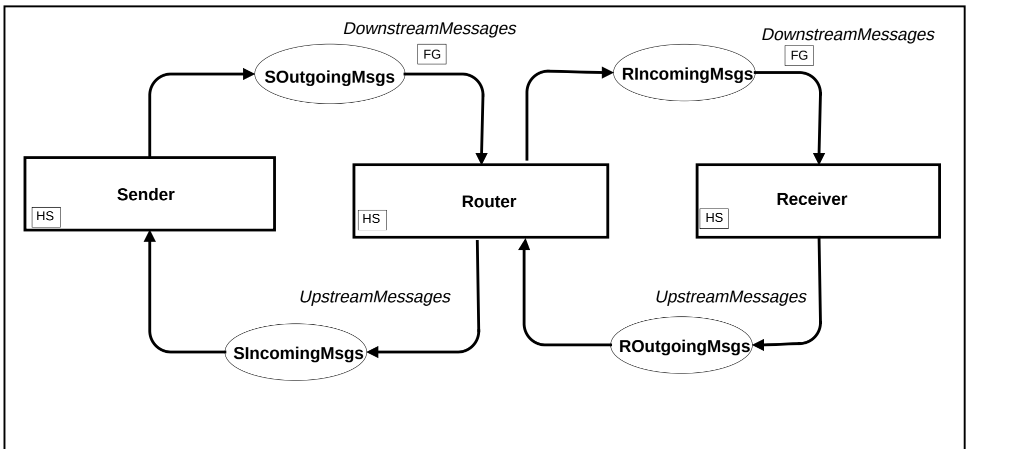


Figure 7: Top-level of the CPN model.

3.3.2 Global declaration

Figure 8 shows the colour sets, variables, and functions from the global declaration node. The colour *RSVPState* indicates the possible states of the RSVP entity. Those states have been chosen based on the description of RSVP state blocks given in [5]:

- CLOSED: when the sender is in this state, it has closed the session. If the receiver is in this state, the receiver application has torn down the existing reservation for the session.
- NOSTATEINFO: there is no state information in the node.
- WAITINGRESV: means that a path has been established but as yet no reservation request has been received.
- RESVREADY: means that a path and reservation have been established.

It may be noted that the CLOSED and NOSTATEINFO states are similar, however the former state is used to limit the model simulation to one Sender session and one receiver application's release call (see section 2.4) .

The subset *RouterState* indicates the possible states of the Router, which does not require the CLOSED state.

```
(* States of RSVP entities*)
color RSVPState = with CLOSED|NOSTATEINFO|WAITINGRESV|RESVREADY;
color RouterState =subset RSVPState with
[NOSTATEINFO,WAITINGRESV,RESVREADY];

(* RSVP Messages *)
color UpstreamMessages = with RESVMSG|RESVTEAR;
color DownstreamMessages = with PATHMSG|RESVERR|PATHTEAR;

(* Variables *)
var sta: RSVPState;

(* Functions *)
fun pathexists (s:RSVPState) = s= WAITINGRESV orelse s=RESVREADY;
fun resvexists (s:RSVPState) = s = RESVREADY;
```

Figure 8: Detailed model declaration.

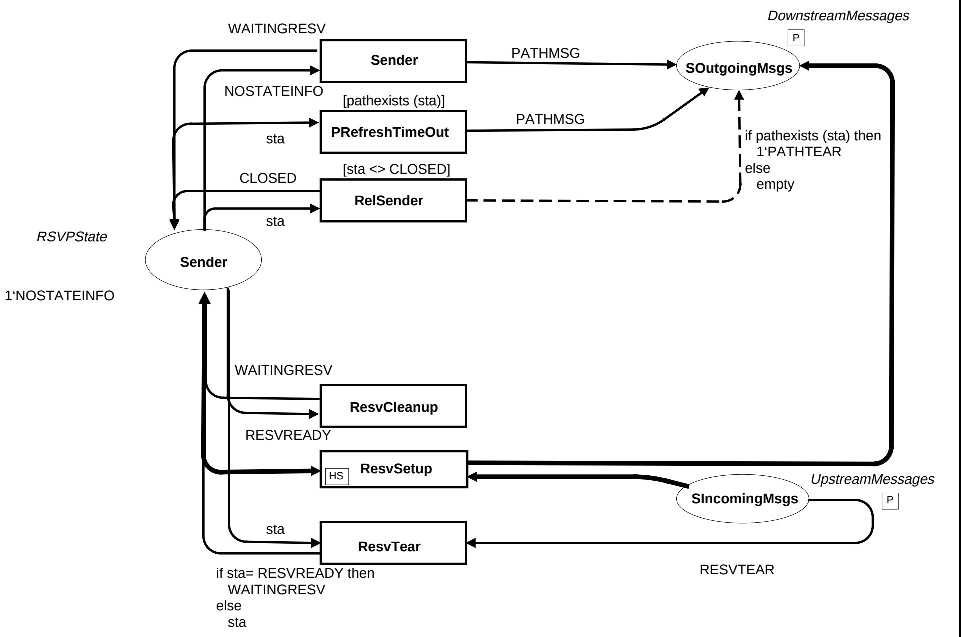


Figure 9: Sender CPN page.

There are five basic RSVP messages represented by the colour sets *UpstreamMessages* and *DowstreamMessages.* The former represents the messages that travel from the receiver to the sender, while the second represents the messages travelling from sender to receiver (see fig. 3). The colour set UpstreamMessages contains the following set of enumerated values:

- RESVMSG: represents a Resv message.
- RESVTEAR: represents a Resv Tear message.

The colour set DowstreamMessages represents the following set of enumerated values:

- PATHMSG: represents a Path message.
- PATHTEAR: represents a Path Tear message.
- RESVERR: represents a Resv Error message.

The variable *sta* is typed by any RSVP State. The functions are used to simplify guard inscriptions. A *pathexists* function means the path state has been established in the correspondent node (ie the RSVP entity – sender, router or receiver places - is in WAITING or RESVREADY state). A *resvexists* function means that a reservation has been established (ie the correspondent RSVP entity is in RESVREADY state).

3.3.3 Structure of CPN subpages

In this section, the three main component pages (ie Sender, Router, and Receiver pages) of the model and some subpages are described (figures 9 to 13).

Sender

In this section, the model of the Sender RSVP entity (see fig. 9) is explained. The Sender's RSVP entity begins in the "NOSTATEINFO" state – the Sender place has an initial marking of

1‘NOSTATEINFO. When the RSVP sender process receives a *Sender Call* from its application (modelled by the *Sender* transition), it will build and send a PATHMSG downstream to the router, and wait for a reservation request.

After a path has been established in a node (ie the RSVP entity is in the WAITINGRESV or RESVREADY state), it must be refreshed every path refresh period (see section 2.3). The transition *PRefreshTimeOut* models the action taken when a path refresh timeout occurs. It may be noted that this transition does not change the Sender state, it only sends a path refresh message (ie a Path Message) to the router.

A RESVMSG, generated by a router, may eventually get to the Sender. The *ResvSetup* subpage models the reservation establishment actions, such as reservation rejection, successful reservation setup or reservation refresh (if a reservation is rejected a RESVERR will be sent back to the router).

If the RSVP entity does not receive a RESVMSG before a reservation cleanup timeout occurs, the transition *ResvCleanup* will remove any reservation state information stored in the node.

The RSVP entity may receive a RESVTEAR from the router because of one of the two reasons explained in section 2.3 (ie state cleanup or service preemption). The transition *ResvTear* models the action taken when a node receives such a message.

Finally, a sender application, which wishes to finish a session, sends a *Release Call* to the RSVP entity. It is modelled by the *RelSender* transition. It will remove any path state and dependent reservation information and send a PATHTEAR to the receiver. It may be noted that the end of a session is modelled by the CLOSED state.

Router

Figure 10 shows the model of the Router’s RSVP entity. Like the Sender’s RSVP entity, the Router’s RSVP entity begins in the “NOSTATEINFO” state. The *PathSetup* subpage (figure 11) models the action taken when a RSVP entity receives a PATHMSG. The transition *NewPath* sets up new path information, and the transition *PathRefresh* refreshes any existing path state information.

In the Router page, the transition *PRefreshTimeOut* models the action taken when a path refresh timeout occurs.

When the RSVP entity does not receive a PATHMSG before a path cleanup timeout occurs, the transition *PathCleanup* will remove any state information stored in the router and send a PATHTEAR to the receiver. The entity may also receive a PATHTEAR from the sender, thus the transition *PathTear* will remove any existing path information and dependent reservation.

The router may receive a reservation request (ie RESVMSG) from the receiver. The *ResvSetup* subpage models the action taken by the RSVP entity to setup a reservation state (figure 12). The transition *AcceptNewResv* models the action taken when a new reservation request (ie RSVP entity is in WAITINGRESV state) is accepted by admission control (ie there is sufficient resources available to grant the requested QoS). Otherwise, the transition *RejectResv* rejects the request and sends a RESVERR to the receiver. The transition *ResvRefresh* refreshes reservation

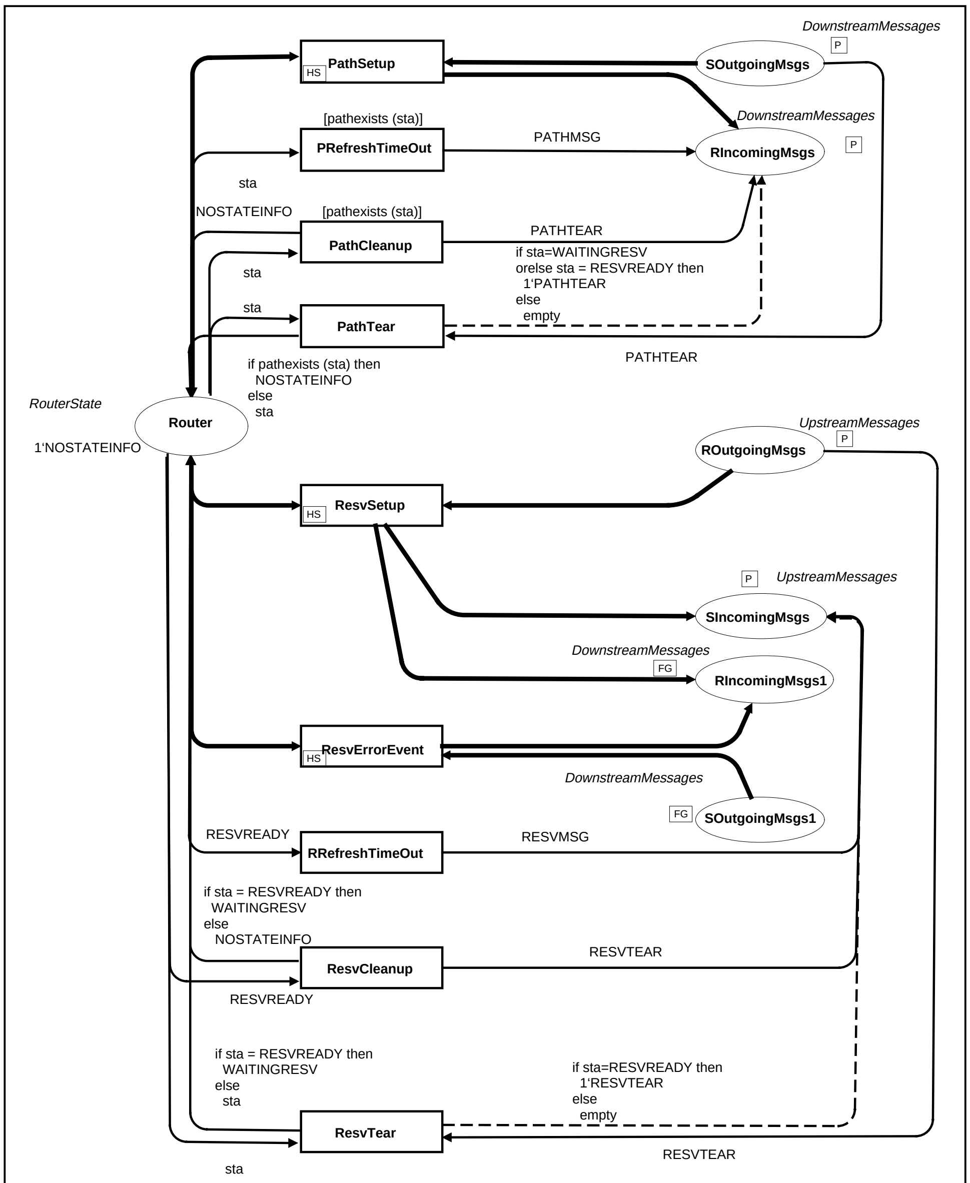


Figure 10: Router CPN page.

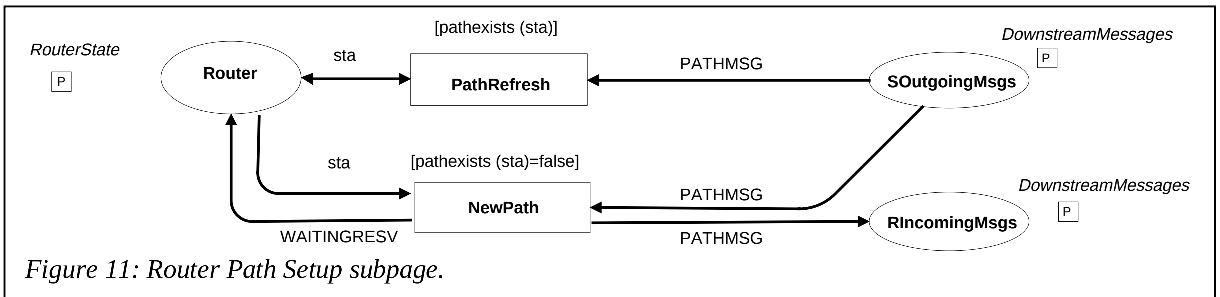


Figure 11: Router Path Setup subpage.

state information upon receiving a refresh message (a RESVMSG) from the receiver. The transition *NoPath* models the action taken when a reservation request arrives but there is no path information stored in the router. It sends a RESVERR to the receiver.

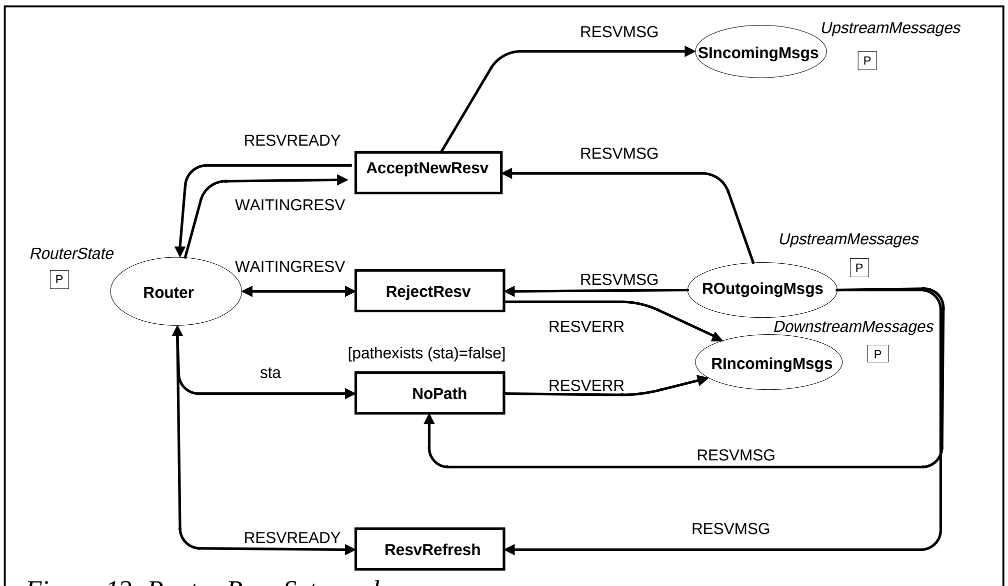


Figure 12: Router Resv Setup subpage.

The router may also receive an error indication (ie RESVERR) from the sender. The *ResvErrorEvent* subpage (in the Router page) models the actions taken in this case.

Once a reservation has been established in the router (ie the RSVP entity is in the RESVREADY state), the reservation state information (see section 2.3) must be refreshed, otherwise it will be removed. The transition *RRefreshTimeOut* models the action taken when a reservation refresh timeout occurs. It does not change the Router state, but sends a reservation message to the sender.

As pointed out before, if a reservation has been established in a router and the RSVP entity does not receive a RESVMSG before a reservation cleanup timeout occurs, the transition *ResvCleanup* will remove any reservation state information stored in the node. In addition, it generates and sends a RESVTEAR to the sender. The RSVP entity may also receive a reservation tear down request (ie RESVTEAR) from the receiver. The transition *ResvTear* will remove any reservation state and send a RESVTEAR to the sender.

Receiver

The Receiver page is shown in figure 13. The RSVP entity (ie Receiver place) begins in the NOSTATEINFO state, as the other two RSVP entities described before. Similarly to the Router page, there is a *PathSetup* subpage which models the path establishment actions, such as new path state setup or path refresh, taken when a PATHMSG arrives to the receiver.

Similarly to the Router's RSVP entity, when the RSVP entity does not receive a PATHMSG before a path cleanup timeout occurs, the transition *PathCleanup* will remove any state information stored in the receiver. The entity may also receive a path teardown request (ie PATHTEAR) from the router, thus the transition *PathTear* will remove any path state and dependent reservation state information.

After a path state has been established in a receiver, the application may send a *Reserve call* to RSVP, which is modelled by the transition *Reserve*. Thus, RSVP will start sending RESVMSGs

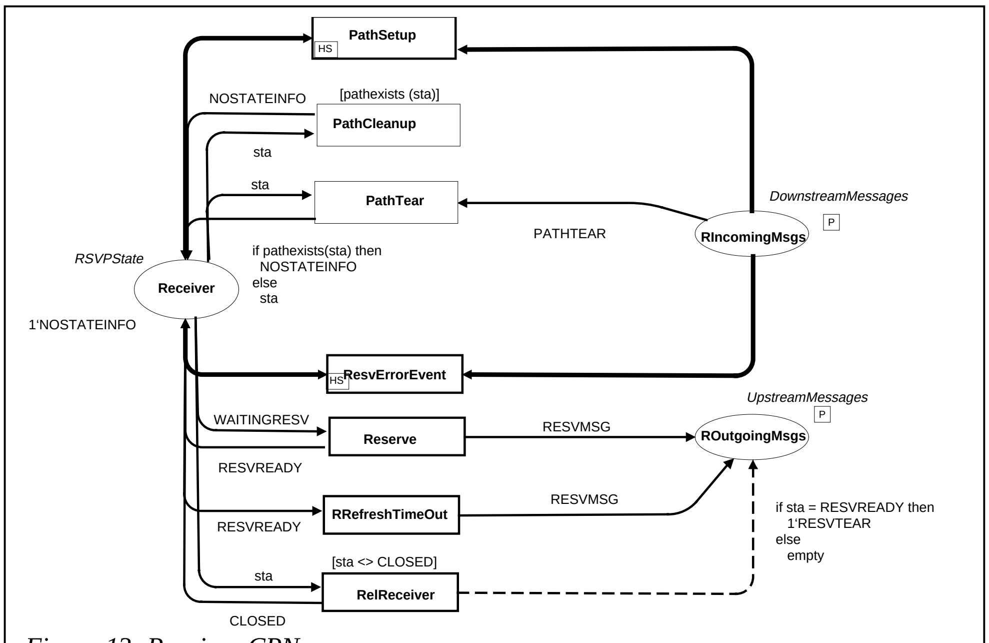


Figure 13: Receiver CPN page.

to the sender. The transition *RrefreshTimeOut* generates and sends RESVMSG every reservation refresh timeout period.

The subpage *ResvErrorEvent* will model the actions taken when the RSVP entity receives an error indication (ie RESVERR) because a reservation setup has failed in either a Sender or Router.

Finally, a receiver application, which wishes to tear down its current reservation, sends a *Release Call* to the RSVP process. It is modelled by the *RelReceiver* transition. It will remove any reservation state information and send a RESVTEAR to the receiver, if it is necessary (ie there is a current reservation).

3.4 Overview of the previous versions of the model

As mentioned previously, the model design was based on an incremental approach where the features of RSVP were included gradually, creating several versions of the CPN model. In this section, previous versions of the CPN model of RSVP are described in terms of the features of RSVP, already introduced before. They are presented incrementally from the simplest model, which includes very basic features of RSVP, to the most complex model which comprises all the features presented in section 3.3.

1. **ResvSetup model:** models path and reservation setup procedures. Neither path nor reservation refresh procedures are considered. Path and reservation requests may be rejected; however, that situation is not reported to either the sender or the receiver because error notification procedures are not modelled in this version.

2. **ResvSetupWithErrors model:** models the same features of RSVP as the ResvSetup model but it also models path and reservation error procedures by using Path and Resv Error messages, respectively.

3. **PathTear model:** models the same features as the previous versions. In addition, it models Path Tear Down procedures initiated by the sender application using a RelSender call.

4. **ResvTear model:** models the same features of RSVP as the previous versions. In addition, it models the Resv Tear Down procedures initiated by the receiver application using a RelReceiver call.

5. **Refresh model:** models the same features of RSVP as the previous versions except that neither path nor resv tear down procedures are considered. It also models path and reservation refresh procedures.

6. **RSVP model:** was described in section 3.3. It models the same features as the Refresh model but it also models path and reservation tear down procedures. In addition, the model is limited to one session. Thus, once a sender application closes a session it is not opened again. Once the receiver application releases its current reservation it is not established again either.

7. **RSVP with multiple sessions model:** is the same as the RSVP model except that a sender application may initiate a session which has been closed previously.

4. Simulation and analysis

Design/CPN was used to simulate and analyse the model. Interactive graphical and automatic simulations were used during its development. In this section, the results of the simulations are presented. Firstly, some results are shown using *the message sequence chart (MSC)* tool provided by Design/CPN. Then, the results of the state space (occurrence graph), calculated for the CPN model, are analysed briefly.

4.1 Message sequence charts

Message sequence charts (MSC) provide a mechanism to visualise the execution of selected traces of communication scenarios [9]. The MSC tool provided by Design/CPN has been useful to check and to visualise the sequence of RSVP events, which may occur during a single simulation run. A MSC for RSVP is given in figure 14. It shows the message interchange between the communicating entities (ie Sender, Router, and Receiver) intended to establish, refresh, tear down, and release a reservation. Vertical lines on the picture represent communicating entities. Small squares on vertical lines describe RSVP events (eg application calls and time outs). Finally, horizontal lines are used to represent message flow. Time increases from the top of the chart, to the bottom of the chart.

Figure 14 shows a simple RSVP message interchange. For simplicity, message overtaking has not been considered for this simulation. The first eight communicating events (represented by arrows and small squares) are concerned with reservation establishment (see fig. 14, event no 1-8), which was explained in section 2. After that, a possible refresh sequence is presented by the following eight events (event no 9-16). Next, a resv cleanup time out occurs as indicated by the small square (event no 17). Thus a reservation is torn down in both the router and the sender (event no 18). Following that, the receiver sends a resv refresh, which propagates to the sender and reestablishes the reservation state on both the router and sender host (event no 19-22). Finally, the sender application finishes a session using a *RelSender* call, and the sender sends a PathTear message which will propagate to the receiver (event no 23-25).

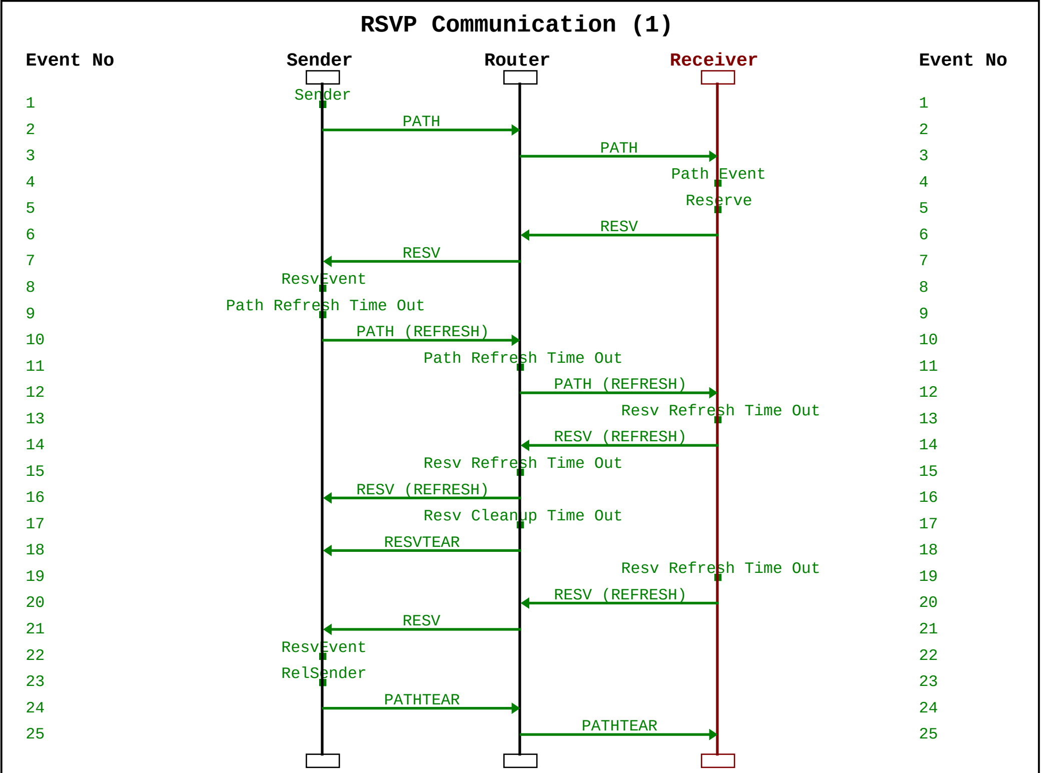


Figure 14: A representative MSC for RSVP.

Since, the RSVP specification does not include any MSCs [4], the MSC tool is useful for providing the sequences of protocol events between the different network entities.

4.2 Occurrence graph analysis

The state space (occurrence graph) tool of Design/CPN was used to investigate some dynamic properties of the CPN model such as boundedness, liveness, and home properties as well as to check the behaviour of the protocol. It was produced by Design/CPN on a Linux PC with 128 MB RAM.

4.2.1 Comparison of several versions of RSVP model

In section 3.4 , several versions of the RSVP model were introduced. Table 1 compares these models in terms of the size of the (full) occurrence graphs (number of nodes and arcs), the time it took to generate them, and the size of the *strongly connected component (SCC)* graphs. Each node of the SCC graph includes all markings, which are mutually reachable from each other [10][13].

It may be seen from table 1 that, in the first four versions of the model, the sizes of the OCC and SCC graphs are the same. Thus there are no loops in the state space graph as expected. The next three versions model RSVP's soft state feature, based on refresh messages and state timeout. Two things may be seen from table 1. Firstly, the sizes of the OCC and SCC graphs are not the same, so there are loops in the state space graph. Those loops are the result of the periodic state refresh and state cleanup. Secondly, the size of the state space has been increased. Those results are expected.

	OCC graph			SCC graph		
Model	**Nodes**	**Arcs**	**Secs**	**Nodes**	**Arcs**	**Secs**
ResvSetup	11	10	0	11	10	0
ResvSetup WithErrors	17	16	0	17	16	0
PathTear	67	105	0	67	105	0
ResvTear	182	400	0	182	400	1
Refresh	613	3584	6	10	379	0
RSVP	13570	96837	536	2601	43493	69
RSVP (with multiple sessions)	24576	199360	18105	1	0	988

Table 1: A comparison of several versions of RSVP model.

In the following sections, the occurrence graph for the RSVP model (the shadowed row in table 1) is analysed. This model includes all the features presented in section 3.3.

4.2.2 Behavioural properties of the model

A full state report for the RSVP model was generated. The statistical information about the size of the full state space and SCC graph is shown in table 1 (the shadowed row). Since there are less SCC nodes (2601) than OCC nodes (13570), it may be concluded that there are some loops. As mentioned before, those loops are due to two reasons. Firstly, the RSVP entities at each node have to send path and refresh messages to avoid the removal of the existing state information. Secondly, any path and reservation state information (represented by the WAITINGRESV and RESVREADY states) may be removed after the correspondent state cleanup occurs and established again by refresh messages.

The state report also shows the home and liveness properties (see table 2). Marking 6 is the only dead marking. This marking corresponds to the state where the sender has finished a session, the receiver has torn down any existing reservation and all messages have been removed from the communication places (see figure 15). Thus, for the initial marking, the protocol behaves as expected (ie no deadlocks). Since this marking is also a home marking the protocol always terminates correctly [10].

Home Markings	[6]
Dead Markings	[6]
Dead Transitions Instances	None
Live Transitions Instances	None

Table 2: Home and liveness properties.

Table 3 shows some information about integer bounds. The minimal number of messages in the communication places is zero, which corresponds of the terminal state to the system. The maximal number of messages is 3 for the Dowstream communication places (SOutgoingMsgs and RIncomingMsgs) and 2 for the upstream communication places (SIncomingMsgs and ROutgoingMsgs). Those results are in alignment with figure 3 except that ResvConf and PathErr

```
6
RSVPNetwork'SOutgoingMsgs 1: empty
RSVPNetwork'RIncomingMsgs 1: empty
RSVPNetwork'SIncomingMsgs 1: empty
RSVPNetwork'ROutgoingMsgs 1: empty
Sender'Sender 1: 1`CLOSED
Router'Router 1: 1`NOSTATEINFO
Receiver'Receiver 1: 1`CLOSED
```
Figure 15: Terminal marking state (node 6).

Places	Upper	Lower
RIncomingMsgs	3	0
RoutgoingMsgs	2	0
SIncomingMsgs	2	0
SOutgoingMsgs	3	0
Sender	1	1
Router	1	1
Receiver	1	1

Table 3: Upper and lower integer bounds.

Places	**Upper multi-set bounds**
RIncomingMsgs	1‘PATHMSG+ 1‘RESVERR+ 1‘PATHTEAR
ROutgoingMsgs	1‘RESVMSG+ 1‘RESVTEAR
SIncomingMsgs	1‘RESVMSG+ 1‘RESVTEAR
SOutgoingMsgs	1‘PATHMSG+ 1‘RESVERR+ 1‘PATHTEAR
Sender	1‘CLOSED+ 1‘NOSTATEINFO+ 1‘WAITINGRESV+ 1‘RESVREADY
Router	1‘NOSTATEINFO+ 1‘WAITINGRESV+ 1‘RESVREADY
Receiver	1‘CLOSED+ 1‘NOSTATEINFO+ 1‘WAITINGRESV+ 1‘RESVREADY

Table 4: Upper multi-set bounds.

messages have not been considered. The number of tokens located in the state places (Sender, Router, and Receiver) is always one, corresponding to each RSVP entity in a particular state.

Table 4 shows some information about multi-set bounds. It shows the messages which can be exchanged between the nodes, both downstream and upstream. The table also shows that all the RSVP entities can be in all the expected states (eg WAITINGRESV state). It means, for example, that a path state can be established in any node. Those results are expected.

4.2.3 Overtaking of Messages

Given the size of the model, visual inspection was just used for some part of the OCC graph. It was used to debug the model (eg finding errors in the arc inscriptions) and to check the behaviour of the protocol. For example, RSVP does not provide any mechanism to deal with message overtaking. Instead it uses refresh messages and periodic cleanups to solve that problem. RSVP nodes can reach an undesirable state when some messages arrive out of order. Checking the OCC graph may easily identify those states. In figure 16, the sender sends a Path message (transition Sender’Sender occurs). After that, it closes the session and sends a Path Tear message downstream (transition Sender’RelSender occurs). The Path Tear message arrives at the router before the Path message (transition Router’PathTear occurs). Once the Path message

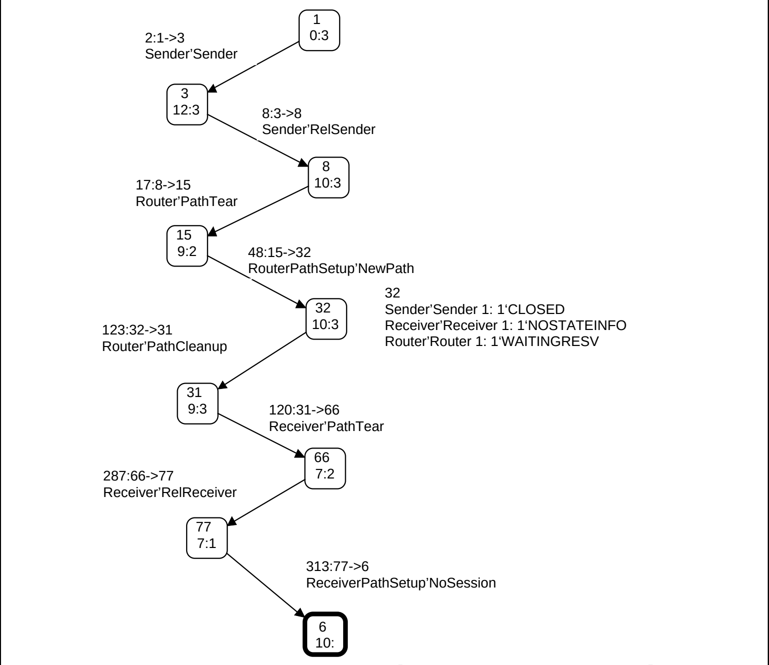


Figure 16: Occurrence sequence showing message overtaking.

arrives at the router, a new path is established (transition Router'PathSetupNewPath occurs). Marking 32 shows an undesirable state, since there is a path established in the router (Router is in WAITINGRESV state). Instead, it is expected that, after a sender has closed a session, all existing path and dependent reservations are removed by the Path Tear message and no further path requests are generated by the sender. Fortunately, later on, the transition Router'PathCleanup removes any state information in the router and the protocol is able to finish in an expected terminal state (marking 6, see figure 15).

5. Conclusions

In this paper, Coloured Petri Nets have been used to provide an initial model of RSVP based on a number of simplifying assumptions. The simplest network topology (one sender, communicating with a receiver via a single router) and unicast operation was assumed, and only the basic features of the protocol were modelled. The model was developed incrementally and checked at each stage to reduce the possibility of modelling errors.

The main problem found during modelling was the lack of a well-defined specification of RSVP [4], where only a narrative description is provided.

Interactive graphical and automatic simulations were used to examine behaviour and to debug the model. Firstly, the MSC tool was useful for providing a graphical overview of the sequence of RSVP events between different network entities. Secondly, the initial analysis of the model based on the state space shows that RSVP terminates correctly. However, the sequence of events described in figure 16 shows the existence of states which are undesirable, though temporary. Further analyses are required to validate the current model. Given the size of the state space and the limited computing resources, it may be necessary to explore some other existing techniques to make the model tractable for state space analysis, such as reduction techniques [11] [17].

This work extends the application of CPNs to a new protocol, proposed for providing QoS guarantees over the Internet. Further work may include: modelling the RSVP service, performance analysis of RSVP using time facilities provided by Design/CPN, employing state space reduction techniques, and extending the RSVP model to a multicast network topology.

Acknowledgment

This work was carried out with financial support from the Commonwealth of Australia through the Cooperative Research Centres Program.

References

[1] Billington J. *Formal Specification of Protocols: Protocol Engineering.* Encyclopedia of Microcomputers, Marcel Dekker, New York, 1991, Vol. 7, pp 299-314.
[2] Blake S., et al. *An Architecture for Differentiated Services.* RFC 2475, IETF, December, 1998.
[3] Braden R., Clark D., and Shenker S. *Integrated Services in the Internet Architecture: an Overview.* RFC 1633, IETF, June, 1994.
[4] Braden R., et al. *Resource Reservation Protocol (RSVP) -- Version 1: Functional Specification.* RFC 2205, IETF, September, 1997.
[5] Braden R. and Zhang L. *Resource Reservation Protocol (RSVP) -- Version 1 Message Processing Rules*. RFC 2209, IETF, September, 1997.
[6] Comer D. *Internetworking with TCP/IP.* Vol 1, Prentice Hall, 1995.
[7] Durham D. and Yavatkar R. *Inside the Internet's Resource Reservation Protocol.* Wiley, USA, 1999.
[8] Feit S. *TCP/IP*. McGraw-Hill, 1998.
[9] ITU (CCITT). Recommendation Z.120: MSC. Technical report, International Telecommunication Union, 1992.
[10] Jensen K. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use.* Vol. 1, Springer-Verlag, 2nd edition, April, 1997.
[11] Jensen K. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use.* Vol. 2, Springer-Verlag, 2nd edition, April, 1997.
[12] Jensen K. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, Vol. 3, Springer-Verlag, April, 1997.
[13] Kristensen L.M., Christensen S., and Jensen K. *The practitioner's guide to coloured Petri nets.* International Journal on Software Tools for Technology Transfer, Springer, 1998, Vol. 2, Number 2, pp 98-132.
[14] Meta Software Corporation. *Design/CPN Reference Manual for X-Windows*, Version 2, Meta Software Corporation, Cambridge, 1993.
[15] Proceedings of the IFIP TC 6/WG6.1 International Workshop on Protocol Specification, Testing, and Verification, North Holland, Amsterdam, 1982-1999.
[16] Proceedings of the International Conference on Network Protocols, 1994 to 1996.
[17] Valmari A. *The State Explosion Problem.* Lectures on Petri Nets I: Basic Models, Vol. 1491, 1998, pp 429-528.
[18] Zhang L., Estrin D., and Zappala D. *RSVP: A New Resource Reservation Protocol*, IEEE Network Magazine, Sept./Oct., 1993, Vol. 7, pp 8-18.

Condensed Storage of Multi-Set Sequences

Marko Mäkelä[*]
Helsinki University of Technology,
Laboratory for Theoretical Computer Science,
P.O.Box 9700, 02015 HUT, Finland

June 27, 2000

Abstract

Tools for state space exploration, or reachability analysers, work by incrementally constructing a set of reachable states. The applicability of these tools is limited by the vast state space of real systems. One way to attack this problem are different reduction methods—another approach is to come up with techniques for representing the set of reachable states in a compact way.

The state—or marking—of a high-level Petri net can be viewed as a sequence of finite multi-sets. A method for encoding markings containing structured values is described, and a comparison to an earlier implementation is presented.

Keywords. Petri nets, reachability analysis, encoding multi-sets, ordered data types

1 Introduction

The limited amount of system memory is a major bottleneck in reachability analysis. Algorithms for reachability analysis and model checking need to keep track of the states that have been explored. In that way, they can detect cyclic behaviour and limit the investigation of successors to truly new states.

There are some techniques that only manage the set of reachable states and utilise similarities between the states. One of them, Binary Decision Diagrams [1, Chapter 5], has been successfully applied mainly in the verification of digital circuits. Techniques applied on the analysis of software systems include a state compaction method for product automata [5] and a method known as Graph Encoded Tuple Sets [6].

One problem with these so called symbolic techniques is that inserting a state may involve global changes, slowing down disk-based implementations. Another problem is that states have no identities: there is no way to retrieve a state from the structure by specifying an index number. Using such a structure for anything else than searching for states fulfilling a predicate or for determining whether a particular state has been explored is tricky.

[*]This research was financed by the National Technology Agency of Finland (TEKES), the Nokia Corporation, the Helsinki Telephone Corporation and the Finnish Rail Administration.

Explicit techniques, which store each state separately, make it possible to navigate in the generated reachability graph and to perform all sorts of queries on it afterwards. When the states are stored separately, they can be assigned index numbers, and it is easy to encode events, the edges of the reachability graph, as triples of two state numbers and a label identifying the action.

This work describes an explicit technique, a method of encoding sequences of multi-sets in a string of binary digits. Symbolic techniques appear promising, but we believe that explicit techniques have an advantage in some applications, such as in the analysis of general software systems, which cannot be characterised by simple laws and which make heavy use of structured data types. Our method, implemented in MARIA [11], has turned out to yield up to an order of magnitude smaller encodings than the method used in PROD [15], although MARIA allows the user to define data types just like in programming languages.

Since our techniques are not specific to any particular class of high-level Petri nets, we try to write in general terms. Even if all data types in MARIA have a finite domain, we shall see that our approach can also handle infinite-domain data types, such as lists.

2 The Reachability Graph

The reachable state space of a model can be represented as a reachability graph, a directed graph whose vertices correspond to reachable states and edges correspond to actions leading from one state to another.

In high-level Petri nets, the states are called *markings* and the actions are called *transition instances*. A transition instance consists of a high-level transition and an assignment for the variables that appear in the arcs and guards connected to the transition.

2.1 Managing the State Space in the File System

Applying explicit analysis techniques to models comprising tens or hundreds of millions of reachable states usually calls for the use of disk storage. Typical reachability analysis algorithms require random access to the set of states explored so far. A similar structure is not required for actions; for most purposes, they can be stored sequentially.

To optimise access to the stored states, one can calculate hash values of the states. When an analysis algorithm wants to determine whether a particular state has been explored, it computes a hash value of the state and searches for it in a memory-based data structure that maps hash values to state numbers. Only if a hash value match is found, the disk address of the encoded state is fetched from a directory file and the state is retrieved from a state file for comparison.

If the encoded states are very small, the memory-based map from hash values to state numbers may exceed the memory limit before the state file exceeds the size limit imposed by the file system. This problem can be addressed by maintaining the map in a disk-based B-tree [13, Ch. 18]. In that case, the system memory consumption remains bounded throughout the analysis, unless some data structures for on-the-fly model checking are kept in the main memory.

The edges of the reachability graph, consisting of source and target state numbers and of an encoded transition instance, are best stored in a separate file. Because the length of the encoded transition instance may vary, also the length is encoded in the file.

2.2 Encoding the Edges and Vertices

2.2.1 Mapping Items to Bit Strings

In order to represent the vertices and edges of the reachability graph as sequences of binary digits, we have to define how the entities they consist of are mapped to such sequences.

Places and Transitions If we denote the set of the places of a Petri net model with $\mathcal{P}$ and assume that there is a bijective mapping

$$o_{\mathcal{P}} : \mathcal{P} \to \{0, \ldots, |\mathcal{P}| - 1\}$$

we can uniquely represent each place with a string of $\lceil \log_2 |\mathcal{P}| \rceil$ binary digits. If $|\mathcal{P}| \leq 1$, no bits are required. The same applies for transitions, characterised by the set $\mathcal{T}$ and the order $o_{\mathcal{T}}$.

Data Items A value of a finite-domain data type $\mathcal{D}$ can be represented as a $\lceil \log_2 |\mathcal{D}| \rceil$-digit binary number, possibly spanning several machine words. This requires a *total order*

$$<_{\mathcal{D}} \subseteq \mathcal{D} \times \mathcal{D}$$

for each data type $\mathcal{D}$. For simple types, such as integers and enumerations, defining the order is straightforward. For structured types, such as tuples, tagged unions and fixed-length or variable-length vectors, the order can be defined lexicographically, e.g. so that variable-length vectors with less elements come first, and that the last component of a structure is the most significant one. This has been implemented in MARIA also for nested structured types.

Once there is a total order among data items, we can define a mapping from the data items to integers:

$$o_{\mathcal{D}} : \mathcal{D} \to \{0, \ldots, |\mathcal{D}| - 1\} : d \mapsto |\{k \in \mathcal{D} \| \, k <_{\mathcal{D}} d\}| \, .$$

It is easy to see that the mapping is bijective and that it preserves the order of the mapped items. Because $<_{\mathcal{D}}$ is a total order, $\mathcal{D}$ can be written as

$$\mathcal{D} = \{d_0, \ldots, d_{n-1}\}$$

such that $d_{i-1} <_{\mathcal{D}} d_i$ for all $0 < i < n$. Now $o_{\mathcal{D}}$ maps each d_i, $0 \leq i < n$, to a unique value:

$$\begin{aligned} o_{\mathcal{D}}(d_i) &= |\{k \in \mathcal{D} \| \, k <_{\mathcal{D}} d_i\}| \\ &= |\{d_0, \ldots, d_{i-1}\}| \\ &= i. \end{aligned}$$

Since $o_{\mathcal{D}}(d_i) = i$, it holds that $o_{\mathcal{D}}(d_i) < o_{\mathcal{D}}(d_j)$ if and only if $i < j$, or $d_i <_{\mathcal{D}} d_j$. Thus, $o_{\mathcal{D}}$ is an order-preserving mapping.

MARIA allows the domains of data types to be restricted with type constraints, internally represented as an ordered list of closed ranges. Our implementation of $o_{\mathcal{D}}(d)$ for constrained types compares the value d to the endpoints of each range in the constraint and performs subtractions and additions.

Mappings for unconstrained structured values are constructed through multiplication and addition from mapped component values. This is similar to the technique represented in [2], but we manage also deeply structured values and constraints consisting of several disjoint ranges.

Structured types can easily have a bigger number of distinct values than one machine word can represent. Our implementation does not convert values of such types to a single binary number, but it handles them component by component. For example, let there be a variable-length vector type

$$\begin{aligned}\mathcal{D} &:= \bigcup_{i=0}^{k} \mathcal{D}_e^i \\ \mathcal{D}_e^i &:= \underbrace{\mathcal{D}_e \times \cdots \times \mathcal{D}_e}_{i \text{ times}}\end{aligned}$$

with $|\mathcal{D}|$ so big that it does not fit in a machine word. To convert a vector value $\langle d_1, \ldots, d_i \rangle \in \mathcal{D}$ to a sequence of binary digits, our implementation encodes i as a $\lceil \log_2 k \rceil$-bit number and converts each element $d_1, \ldots, d_i$ separately to a bit string. If also the element type $\mathcal{D}_e$ is a large structured type, the elements are handled in a similar way; otherwise, the mapping $o_{\mathcal{D}_e}$ can be applied.

Tagged unions are handled in an analogous way: First, the active component is identified with a binary number. Then the encoded representation of the active component is appended to the bit string. Tuples and fixed-length arrays are simpler, since the number and type of components remain constant.

All data types that can be defined in MARIA have a finite domain. Also the variable-length buffer data type is assigned a capacity, the maximum number of elements a buffer value can contain. If there were any infinite-domain data types,[1] they could be handled in a similar way with large structured types. For instance, an unbounded string or linked list of an item type $\mathcal{D}$ can be represented by encoding each item separately and by using a special value for signalling the end of the sequence. If $|\mathcal{D}|$ can be represented in a machine word, it can be used as the special value. Otherwise, it is easiest to use one extra bit per data item as the end marker.

2.2.2 Encoding Edges

An edge of the reachability graph consists of two numbers identifying the source and target states and of a transition instance consisting of a transition identifier and an assignment for the variables required for firing the transition.

If there is no statistical information available on the transition enablings, the transitions can be assumed to occur with equal probabilities. In that case, our representation of the transitions $t \in \mathcal{T}$ with $\lceil \log_2 |\mathcal{T}| \rceil$-digit binary numbers $o_{\mathcal{T}}(t)$ is close to the optimum defined by the entropy of the system [14, Ch. 6–7].

When the variables of the transition instance are processed in a systematic order, it suffices to encode only the values of the variables and to append them to the bit string representing the label of the edge. Similarly, if the analyser generates all successors of a state in one step, it suffices to store the source state number only once for a bunch of edges originating from the state. Keeping track of the number of states generated so far allows the encoder to use less bits for representing the state numbers.

[1] We restricted ourselves to finite types to avoid difficulties with verification algorithms that operate on unfolded nets.

If the formalism allows some of the variables of an enabled transition to be undefined—that is, if all arc expressions and gates can be evaluated without dereferencing a variable—the encoder must use one bit for signalling whether the variable has been assigned a value.

All this data can be encoded into one sequence of binary digits. When the binary digit string is written to a file, it is good to align it at a byte or machine word boundary.

In some applications, it is not necessary to store the labels of the edges, since they can be reconstructed by analysing all enabled transition instances in the source state, and by finding the instances that lead to the specified target state. This is computationally expensive, but if it only has to be done when displaying to the user a counterexample path of at most a few hundred or thousand steps, the cost of saving tens of megabytes of disk space might be only a few seconds of wasted processor time.

2.2.3 Encoding Vertices

In the case of high-level Petri nets, the vertices of the reachability graph are markings. A marking is a family of multi-sets, indexed by places. A multi-set over a set is a mapping from the items of the set to the set of natural numbers, $\mu : A \to \mathbb{N}$. Unlike normal sets, a multi-set may contain more than one instance of an item. The number of times an item $a \in A$ is contained in a multi-set μ is called the *multiplicity* $\mu(a)$. The union operation of normal sets can be extended to multi-sets as an operation that adds multiplicities.

When the places $p \in \mathcal{P}$ are mapped to numbers $o_{\mathcal{P}}(p)$, the marking can be viewed as a sequence of multi-sets. The multi-set at the position $o_{\mathcal{P}}(p)$ of the sequence corresponds to the local marking of the place p.

A straightforward implementation encodes each multi-set in the sequence separately and appends it to a bit string representing the marking. The details are shown in the following section.

3 Storing Markings

Storing sequences of multi-sets in finite space involves a fundamental problem: the range of a multi-set μ is the infinite set of natural numbers. An implementation in a finite-memory computer must restrict the choice of the multiplicities $\mu(a)$ to a finite set, typically $0 \leq \mu(a) < 2^n$ with $n = 16$ or $n = 32$.

Since the multi-sets in the reachable markings of practical models usually map most items to zero multiplicity, it makes sense to represent each multi-set as a sequence of pairs $\langle \mu(a), a \rangle$ having $\mu(a) > 0$.

An implementation that enforces a limit $0 \leq \mu(a) < 2^n$ could encode the multiplicity of each $\langle \mu(a), a \rangle$ pair in n binary digits and mark the end of the sequence with a string of n zero bits. Such a simple encoding requires $(|\mathcal{P}| + d)\, n$ bits for storing the multiplicities of a marking of a $|\mathcal{P}|$-place net containing d distinct tokens.

3.1 Representing Multiplicities

A multi-set μ over a set A can be characterised by two quantities: the cardinality, or the total number of items

$$t = \sum_{a \in A} \mu(a)$$

and the number of distinct items

$$d = |\{a \in A \| \mu(a) > 0\}| .$$

The cardinality can theoretically be any natural number, but a finite-memory implementation limits it, typically $0 \leq t < 2^n$ for some n.

An user-defined *capacity constraint,* a Boolean condition on t, can reduce the number of bits required for representing t. If there are m different possibilities for the total number of tokens in a place, the actual number t can be represented using $\lceil \log_2 m \rceil$ bits, since a k-digit binary number can represent 2^k different things.

Encoding the cardinality t before the number of distinct items d has one advantage: it is straightforward to see that $1 \leq d \leq t$ when t is nonzero. Therefore, d can be represented using $\lceil \log_2 t \rceil$ bits.

For the greatest multiplicity $\mu_{\max}$ in the multi-set it holds that

$$\left\lceil \frac{t}{d} \right\rceil \leq \mu_{\max} \leq 1 + t - d.$$

If $\mu_{\max}$ is at its upper bound $1 + t - d$, the other $d - 1$ distinct items must have a multiplicity of 1 in order for the total number of items to be t. Similarly, if $\mu_{\max} = \left\lceil \frac{t}{d} \right\rceil$, the multiplicities of the remaining items must be equal to $\mu_{\max}$ or $\mu_{\max} - 1$.

So, the greatest multiplicity $\mu_{\max}$ can always be represented with

$$\left\lceil \log_2 \left(2 + t - d - \left\lceil \frac{t}{d} \right\rceil \right) \right\rceil$$

binary digits. After decoding $\mu_{\max}$, the decoder knows the remaining cardinality $t' = t - \mu_{\max}$ and the number of remaining distinct items $d' = d - 1$. If the multiplicities are encoded in descending order, the encoder always selects the greatest of the remaining multiplicities and represents it using less and less bits.

This encoding of multiplicities appears to be quite compact even when capacity constraints are not used. For representing $d = 5$ multiplicities, the simple encoding described in Section 2.2 would use $6n$ bits. The optimised encoding needs n bits for representing the cardinality. Assuming that it is 8, the number of distinct tokens is encoded in 3 bits. The greatest multiplicity lies between $\left\lceil \frac{8}{5} \right\rceil = 2$ and $8 - 5 + 1 = 4$; therefore it can be represented with 2 bits. Clearly, the improved encoding requires less than $n + 3 + 5 \cdot 2 = n + 13$ bits. The difference between $6n$ and $n + 13$ is tangible already when $n = 16$.

Our encoding scheme for multiplicities is a variable-length code. In the best case, when $d = 1$ or $d = t$, our code only requires $\lceil \log_2 t \rceil$ bits for representing d—no further bits are required for representing the multiplicities. Figure 1 compares the performance of our code against a fixed-length code that maps multiplicity distributions to a zero-based index numbers. For instance, there are 7 different multiplicity distributions for multi-sets of cardinality 5, if the

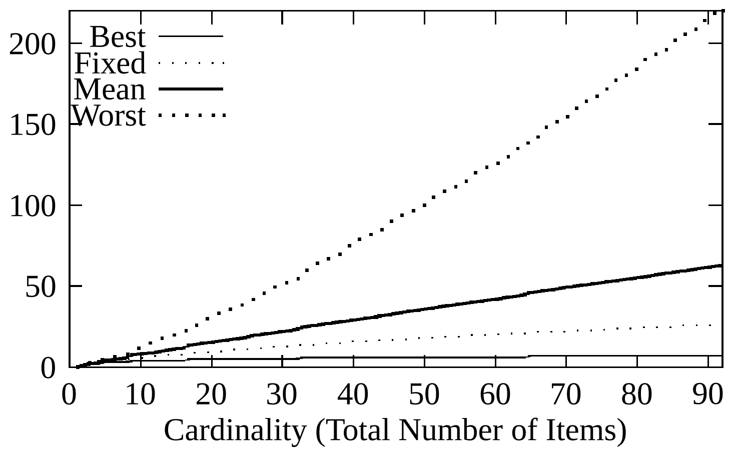


Figure 1: Number of Bits Required for Representing Multiplicities

multiplicities are sorted in descending order: 5, 41, 32, 311, 221, 2111, 11111. Each distribution can be represented by a $\lceil \log_2 7 \rceil$-bit number.

The average bit consumption of our code is slightly more than two times the size required by the fixed-length code. The result is not so bad, since the fixed-length code is computationally much more expensive than our method. Also, the reachable markings in typical Petri net models tend to consist of ordinary sets—an optimal case for our code.

3.2 Representing Empty Multi-Sets

In many practical models, there is a substantial number of empty places in most reachable markings. With our optimised multiplicity encoding, an empty place requires $\lceil \log_2 m \rceil$ bits of storage, if there are m different possibilities for the total number of tokens in the place.

As it is rather uncommon to define tight capacity constraints in models, representing the cardinalities typically requires one machine word per place. If the machine word length is n bits, we would still need $|\mathcal{P}|\,n$ bits for representing an empty marking. There ought to be a more compact encoding for empty places.

Our solution is to start the encoded marking with the number of empty places n_e, $0 \leq n_e \leq |\mathcal{P}|$. This requires $\lceil \log_2(|\mathcal{P}|+1) \rceil$ binary digits. There are

$$\binom{|\mathcal{P}|}{n_e} = \frac{|\mathcal{P}|!}{n_e!\,(|\mathcal{P}|-n_e)!}$$

ways to pick a subset of n_e empty places from the set of all $|\mathcal{P}|$ places. It is possible to enumerate these subsets and to represent each of them as a binary number with

$$\left\lceil \log_2 \binom{|\mathcal{P}|}{n_e} \right\rceil$$

digits. It is easy to see that this code occupies at most $|\mathcal{P}|$ bits, since the total amount of all subsets of the set $\mathcal{P}$

$$\sum_{n_e=0}^{|\mathcal{P}|} \binom{|\mathcal{P}|}{n_e}$$

evaluates to exactly $2^{|\mathcal{P}|}$. For $|\mathcal{P}| = 1$ we have $1+1 = 2^1$, and assuming that the claim holds for a set of magnitude $|\mathcal{P}|$, it follows that

$$\begin{aligned}
\sum_{n_e=0}^{|\mathcal{P}|+1} \binom{|\mathcal{P}|+1}{n_e} &= \binom{|\mathcal{P}|+1}{0} + \sum_{n_e=0}^{|\mathcal{P}|} \binom{|\mathcal{P}|+1}{n_e+1} \\
&= 1 + \sum_{n_e=0}^{|\mathcal{P}|} \binom{|\mathcal{P}|}{n_e} + \sum_{n_e=0}^{|\mathcal{P}|-1} \binom{|\mathcal{P}|}{n_e+1} \\
&= 1 + 2^{|\mathcal{P}|} + (2^{|\mathcal{P}|} - 1) \\
&= 2^{|\mathcal{P}|+1}.
\end{aligned}$$

Instead of constructing this kind of a fixed-length code, we developed and implemented in MARIA a simple variable-length encoding scheme, which we shall present below.

3.2.1 A Variable-Length Code

Clearly, if the number of empty places n_e happens to be 0 or $|\mathcal{P}|$, there is only one way to select the subset, and it can be identified by a zero-length code. In the following, we assume that $0 < n_e < |\mathcal{P}|$.

If $\frac{1}{2}|\mathcal{P}| < n_e < |\mathcal{P}|$—that is, there are more empty places than nonempty ones—then it makes sense to explicitly represent the identity of the nonempty places. The encoding we have defined so far identifies each place $p \in \mathcal{P}$ with an index number $0 \leq o_{\mathcal{P}}(p) < |\mathcal{P}|$, and it encodes the multi-sets associated with the places in ascending order of index numbers.

The smallest index number i_1 of a nonempty place must be in the range

$$0 = l_1 \leq i_1 \leq h_1 = n_e$$

since there are at most n_e empty places in the beginning of the sequence. So, i_1, the index number of the first nonempty place, can be stored using $\lceil \log_2(h_1 - l_1 + 1) \rceil$ binary digits. What about the following nonempty places i_{k+1}? It holds that

$$i_{k+1} \geq l_{k+1} = i_k + 1,$$

since the indices are processed in ascending order. It is easy to see that there are $i_k - (k-1)$ empty places before i_k, since i_k is the kth smallest index of a nonempty place. Thus, of the places following i_k, $n_e - (i_k - (k-1))$ are empty, and for the upper limit $h_{k+1} \geq i_{k+1}$ we have

$$\begin{aligned}
h_{k+1} &= l_{k+1} + n_e - (i_k - (k-1)) \\
&= i_k + 1 + n_e - i_k + k - 1 \\
&= n_e + k.
\end{aligned}$$

Since $h_1 = n_e$, it is easy to see that $h_{k+1} = h_k + 1$.

Similarly, if $0 < n_e \leq \frac{1}{2}|\mathcal{P}|$, we represent the indices of empty places. This is analogous to the previous case; we just start with $h_1 = |\mathcal{P}| - n_e$.

This technique is illustrated in Figure 2, which demonstrates a case with 13 places, 6 of which are to be identified. The one with the smallest index i_1 must fulfill the condition $0 \leq i_1 \leq$

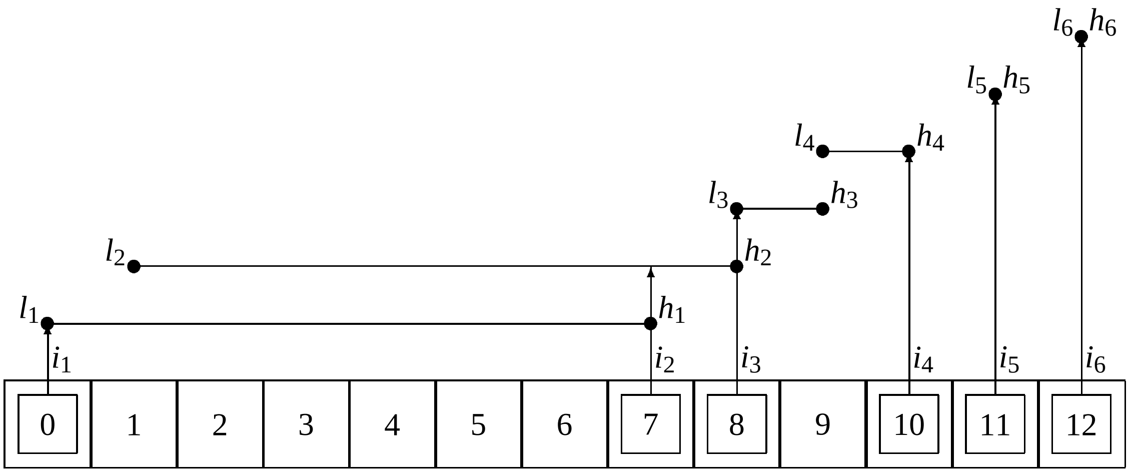


Figure 2: Representing a 6-Element Subset of a Set with 13 Elements

7. A fixed-length code representing i_1 takes 3 bits. Unfortunately for us, i_1 is at the smallest possible position, and the range for the next index is of the same size: $1 \leq i_2 \leq 8$. Since i_2 occurs almost at the end of its range, the uncertainty over the position of the remaining indices reduces. Our approach requires 1 bit for storing i_3 and i_4. After i_4 has been stored, no further bits are required. Our encoding uses a total of 8 bits for identifying the empty places. The fixed-length code would use $\left\lceil \log_2 \binom{13}{6} \right\rceil = 11$ bits for this case.

In the worst case, when all m places to be identified occur in the first m positions, our approach requires the same number of bits for representing each index, a total of $m \lceil \log_2(m+1) \rceil$ bits. In the best case where the first index occurs at the end of its range, the total requirement drops to $\lceil \log_2(m+1) \rceil$ bits.

3.2.2 Keep it Simple

Figure 3 compares the space consumption of our variable-length encoding scheme against the fixed-length code discussed in the beginning of this section. We have seen that the fixed-length code never uses more than $\lceil \log_2(|\mathcal{P}|+1) \rceil + |\mathcal{P}|$ binary digits. Its average bit consumption is

$$\log_2(|\mathcal{P}|+1) + \frac{1}{|\mathcal{P}|+1} \sum_{n_e=1}^{|\mathcal{P}|-1} \left\lceil \log_2 \binom{|\mathcal{P}|}{n_e} \right\rceil .$$

The average space consumption of our variable-length code appears to be more than one bit per place. Even if our implementation made use of fractional bits, the worst case for $|\mathcal{P}| = 20$ would require almost 39 bits, nearly two bits per place. This raises a thought: Why not use exactly one bit per place for marking empty places? The decoder would not even need to know the number of empty places in advance, which allows us to save further $\lceil \log_2(|\mathcal{P}|+1) \rceil$ bits.

This simple code can easily be optimised further for places having a capacity constraint. No signalling bit is required for places that are constrained to be nonempty. Also, if it is possible to represent the cardinality using no more than, say, 2 bits, the emptiness bit can be omitted.

A further optimisation can be made regarding places with no capacity constraints. In practice, places in Petri nets are likely to contain a small number of tokens. Using a shorter repre-

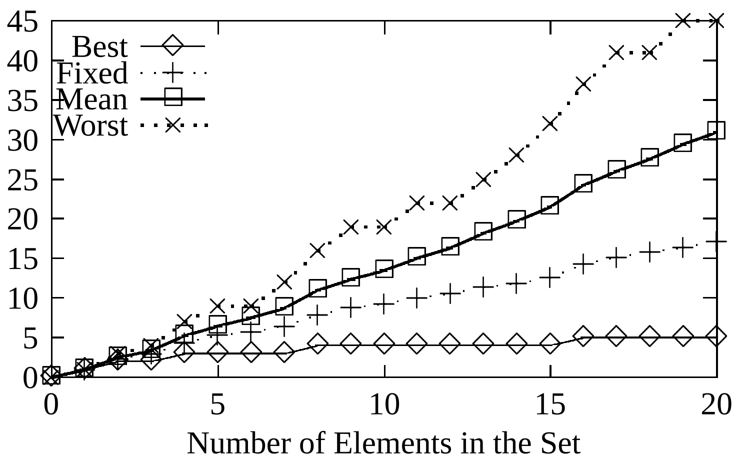


Figure 3: Number of Bits Required for Representing Subsets

sentation for small cardinalities seems to make sense. Above we have suggested a code of at most 1 bit for representing the cardinality $t = 0$. This code can only tell whether $t = 0$ or $t > 0$. In the latter case, more bits are required for encoding the exact value of t. Our implementation in MARIA uses 4 more bits for representing the values $1 \leq t \leq 8$, 10 for $9 \leq t \leq 264$, 19 for $265 \leq t \leq 65800$, and $4+n$ bits for representing the values $65801 \leq t < 2^n$.

3.3 Redundant Places

Certain commonly applied modelling practices introduce redundancy in the markings of Petri net models. Some of it can be removed by transforming the net to an equivalent one, but not everything. For instance, if there are no inhibitor arcs in the formalism, it is difficult to remove complement places.

All practical models are likely to contain redundant places. The state encoder would perform better if it could somehow omit all redundant places from the encoded marking. The only problem is that there must be a mechanism for computing the contents of redundant places when decoding the marking.

MARIA solves the problem by allowing the initialisation expressions of places to refer to the markings of other places. When a marking is about to be added to the reachability graph, the encoder ensures that there is no controversy in the initialisation expressions of redundant places, and issues an error message if there is. Thus, these user-supplied “invariants” can be viewed as an additional safety check supplied by the analyser, just like capacity constraints and checks in the expression evaluator.

4 An Example

Figure 4 illustrates a high-level Petri net model of a distributed data base management system, originally presented by Genrich, Lautenbach and Jensen [3, 9]. In the initial marking of the model, all places except **exclusion** and **inactive** are empty. The latter place is initialised with

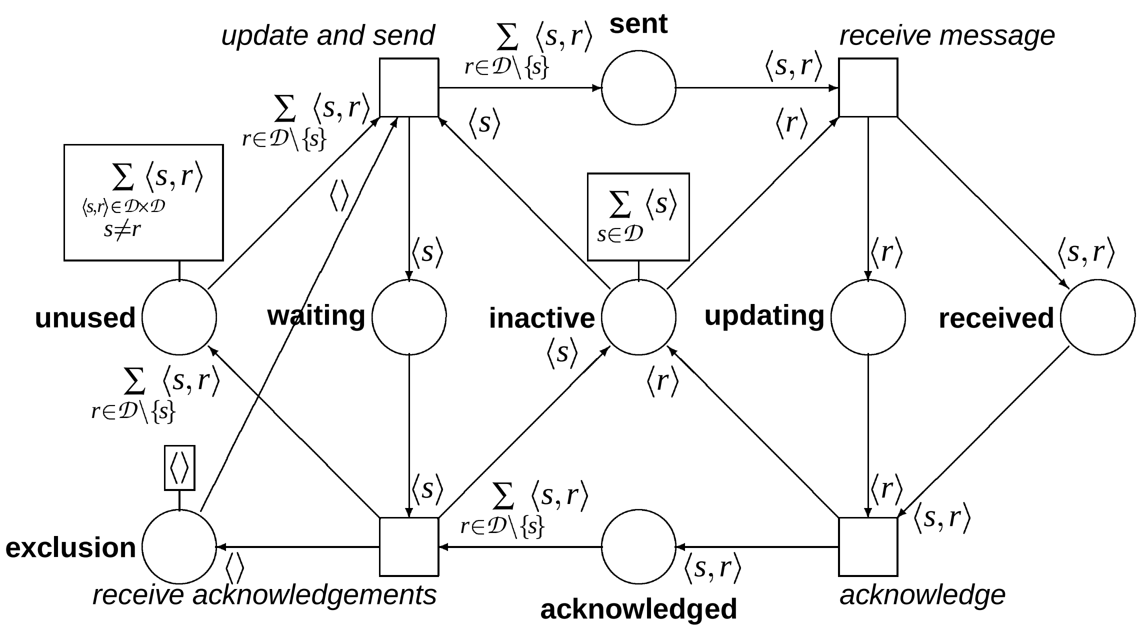


Figure 4: Model of a Distributed Data Base Management System

a multi-set sum of the items in the set $\mathcal{D}$ representing the data base servers. In other words, all data base servers are inactive in the initial state of the model.

It is fairly easy to see that when $\mathcal{D}$ is finite, the reachable state space of the model is finite. The model is also bounded: the places **waiting** and **exclusion** contain at most one token, the place **inactive** contains at most $|\mathcal{D}|$ tokens, and the other places may contain at most $|\mathcal{D}| - 1$ tokens.

4.1 Encoding the Initial Marking

The initial marking has 3 nonempty places: **unused**, **inactive** and **exclusion**. These places are actually redundant: The place **unused** and the arcs attached to it could be removed from the model without affecting its behaviour. The place **exclusion** is kind of complementary to the place **waiting**, and **inactive** contains all those items of $\mathcal{D}$ not contained in the places **waiting** and **updating**. Utilising this information, our scheme would encode the initial marking in 5 bits, one for each non-redundant place, signalling that the places are empty.

If our encoding scheme is told nothing about the redundancy, it uses a total of $|\mathcal{P}| = 8$ bits for identifying the three nonempty places.

The cardinality of the multi-set associated with the place **inactive** is $t = d = |\mathcal{D}|$. If there is a capacity constraint $0 \le t \le |\mathcal{D}|$, these two quantities can be encoded in $\lceil \log_2(|\mathcal{D}|+1)\rceil + \lceil \log_2 |\mathcal{D}|\rceil$ bits; otherwise, $n_c(|\mathcal{D}|) + \lceil \log_2 |\mathcal{D}|\rceil$ bits are required where n_c tells how many bits our variable-length code for cardinalities takes:

$$n_c(t) = \begin{cases} 4 & \text{if } 1 \le t \le 8 \\ 10 & \text{if } 9 \le t \le 264 \\ 19 & \text{if } 265 \le t \le 65800 \\ 4+n & \text{if } 65801 \le t < 2^n \end{cases}$$

All items in the multi-set for **inactive** have the multiplicity 1. Although the items in the multi-set represent the whole set $\mathcal{D}$, our encoder represents each value separately, using $|\mathcal{D}|\lceil\log_2|\mathcal{D}|\rceil$ bits.

For the place **exclusion**, we have $t = 1$. This leaves no choice for the number of distinct tokens, which is also 1. Encoding the item takes no bits, since the data type associated with the place has only one value—the empty tuple $\langle\rangle$. If there is a capacity constraint that dictates $0 \le t \le 1$, one bit is enough for encoding the multi-set. Otherwise t is represented with 4 binary digits, since it is in the range $1 \le t \le 8$.

The place **unused** is initially marked with a multi-set of the cardinality $t = |\mathcal{D}|^2 - |\mathcal{D}|$. The reachable markings of the model appear to fulfill the capacity constraint $t = 1 + |\mathcal{D}|^2 - 2|\mathcal{D}| \vee t = |\mathcal{D}|^2 - |\mathcal{D}|$ for this place. This capacity constraint allows t to be represented in one bit. The encoder does not know that the number of distinct items is $d = t$; it assumes $1 \le d \le t$ and therefore represents d as a $\lceil\log_2 t\rceil$-digit binary number. Each item in the multi-set requires $\lceil 2\log_2|\mathcal{D}|\rceil$ bits of storage, assuming that the model uses an unconstrained data type for storing the pairs.[2] Later we shall see that representing this redundant multi-set substantially increases the space requirements. In the following summary, we consider a model where this place has been removed.

To summarise, the initial marking—excluding the place **unused**—fits in $11 + n_c(|\mathcal{D}|) + (|\mathcal{D}| + 1)\lceil\log_2|\mathcal{D}|\rceil$ bits when no capacity constraints or redundancy information are exploited. With tight capacity constraints, only $8 + \lceil\log_2(|\mathcal{D}| + 1)\rceil + (|\mathcal{D}| + 1)\lceil\log_2|\mathcal{D}|\rceil$ bits are required. The difference $3 + n_c(|\mathcal{D}|) - \lceil\log_2(|\mathcal{D}| + 1)\rceil$ is always positive in our implementation, since $n_c(t) > \lceil\log_2 t\rceil$. For $|\mathcal{D}| = 10$, utilising the capacity constraints saves 9 bits. When the redundancy information is utilised, the initial marking (where all non-redundant places are empty) can be encoded in only 5 bits, independent of $|\mathcal{D}|$ and n.

4.2 Encoding All Reachable Markings

Our scheme for encoding markings has been implemented in MARIA [11, 12], a reachability analyser for Algebraic System Nets [10] with user-definable finite-domain structured data types. We compare the performance of MARIA with PROD [15], a reachability analyser for a kind of Predicate/Transition Nets [4].

Tables 1 and 2 illustrate the performance of our state encoding scheme. We analysed the model `dbm.net` distributed with PROD without and with unfolding, and three variants of a corresponding model with MARIA: without and with capacity constraints, and with redundant places indicated.

The figures in Table 2 are for models where the redundant place **unused** has been removed. The space consumption drops to less than a fifth when this place is omitted. A natural explanation is that this place has a complementary character: it contains a large number of tokens in all reachable markings. If PROD or MARIA used the initial marking as a reference when encoding other markings, the differences between the two tables would be considerably smaller.

The figures do not include the space required for the graph directory. In PROD, it consists of a fixed header and of a record of 8 machine words—typically 32 bytes—per state. In MARIA, the directory is a table of hash values and file offsets. On a 32-bit system with 32-bit file offsets,

[2] This number would drop to $\lceil\log_2(|\mathcal{D}|^2 - |\mathcal{D}|)\rceil$ if we defined the domain of the place to be a multi-set over $(\mathcal{D}\times\mathcal{D})\setminus\bigcup_{s\in\mathcal{D}}\{\langle s,s\rangle\}$ instead of $\mathcal{D}\times\mathcal{D}$.

Table 1: Reachability Graph Sizes for the Distributed Data Base Model

Model Size		Encoded State Space in Bytes				
$\lvert\mathcal{D}\rvert$	States	PROD	(unfolded)	MARIA	(cap.)	(red.)
1	2	19	3	4	2	2
2	7	99	29	28	15	9
3	28	645	844	223	168	86
4	109	3,925	1,745	1,403	1,090	414
5	406	21,519	10,151	8,439	7,487	2,396
6	1,459	107,967	54,307	46,109	42,292	10,807
7	5,104	505,297	280,229	208,807	198,599	43,569
8	17,497	2,239,617	1,296,227	908,810	873,816	168,089
9	59,050	9,507,051	5,605,363	4,423,852	4,308,020	738,397
10	196,831	38,972,539	23,134,187	18,006,540	17,685,747	2,683,381

Table 2: Reachability Graph Sizes for the Model Excluding the Place **unused**

Model Size		Encoded State Space in Bytes				
$\lvert\mathcal{D}\rvert$	States	PROD	(unfolded)	MARIA	(cap.)	(red.)
1	2	17	3	4	2	2
2	7	76	21	21	14	9
3	28	389	139	150	111	86
4	109	1,848	761	724	543	414
5	406	8,113	3,651	3,565	3,159	2,396
6	1,459	33,548	16,045	15,725	14,266	10,807
7	5,104	132,693	76,067	60,786	55,683	43,569
8	17,497	507,400	357,219	233,702	217,213	168,089
9	59,050	1,889,585	1,511,227	998,945	952,558	738,397
10	196,831	6,889,068	6,009,887	3,559,389	3,526,147	2,683,381

the bookkeeping overhead is 8 bytes per encoded state.

When there is no capacity constraint, our implementation represents the cardinality of a multi-set using a variable-length code, which occupies $1+4$ bits more than one machine word in the worst case. The other extreme is a capacity constraint that allows only one value for the total number of tokens. Defining a capacity constraint can thus save more than one machine word for each non-empty place in the marking.

We also translated a variant of the ISDN-DSS1 protocol model [8] from PROD to MARIA format. The encoded representation of the 20,084 states takes 37.2 bytes per state in PROD (38.3 for an unfolded model) and 9 in MARIA, or 13.7 if no capacity constraints are defined.

The run-time overhead of our encoding method is negligible, as our implementation makes heavy use of automatically generated, dynamically linked C code. When analysing the above mentioned ISDN-DSS1 model, the analyzer spends less than 4 percent of its total time in the encoder. This can partially be explained by the relatively large number of places and transitions in the model, which shifts the bottleneck to transition instance analysis. When analysing different variations of the data base model, we experienced that encoding states takes 9–22 percent of the total time. The worst figure was obtained for a model that included the redundant place **unused** and supplied a marking-dependent initialisation expression for it.

5 Conclusion and Future Work

We have presented a scheme for condensed explicit storage of markings of high-level Petri nets, representing the multiplicities of multi-set items in a compact way. Even though our scheme does not utilise any similarities between or inside markings in any way, an implementation of it performs up to an order of magnitude better than a previously implemented scheme even for simple models.

Our idea of using $\lceil \log_2 |\mathcal{D}| \rceil$ binary digits for representing multi-set items belonging to a finite set $\mathcal{D}$ assumes that each item occurs with equal probability. This is often not the case with practical models, and it would be worth investigating how well Holzmann's ideas on recursive indexing and compression training runs [7] could be combined with our approach.

References

[1] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[2] Jaco Geldenhuys and Pieter de Villiers. Runtime efficient state compaction in SPIN. In Dennis Dams, Rob Gerth, Stefan Leue and Mieke Massink, editors, *Theoretical Aspects of Model Checking, 5th and 6th International SPIN Workshops*, pages 12–21, Trento, Italy, July 1999. Springer-Verlag, Berlin, Germany, 1999.

[3] Hartmann J. Genrich and Kurt Lautenbach. The analysis of distributed systems by means of Predicate/Transition-Nets. In Gilles Kahn, editor, *Semantics of Concurrent Computation*, volume 70 of *Lecture Notes in Computer Science*, pages 123–146, Evian, France, July 1979. Springer-Verlag, Berlin, Germany, 1979.

[4] Hartmann J. Genrich. Predicate/Transition Nets. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Central Models and their Properties—Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247, Bad Honnef, Germany, September 1986. Springer-Verlag, Berlin, Germany, 1987.

[5] Patrice Godefroid and Gerard J. Holzmann. On the verification of temporal properties. In *13th IFIP Symposium on Protocol Specification, Testing and Verification*, pages 109–124, Liège, Belgium, May 1993.

[6] Jean-Charles Grégoire. State space compression in SPIN with GETSs. In *2nd International SPIN Verification Workshop*, New Brunswick, NJ, USA, August 1996.

[7] Gerard J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *3rd International SPIN Verification Workshop*, Enschede, The Netherlands, April 1997.

[8] Nisse Husberg, Teemu Tynjälä and Kimmo Varpaaniemi. Modelling and analysing the SDL description of the ISDN-DSS1 protocol. To appear in *Application and Theory of Petri Nets 2000: 21st International Conference, ICATPN'00*, Århus, Denmark, June 2000.

[9] Kurt Jensen. Coloured Petri Nets and the invariant method. *Theoretical Computer Science*, 14(3):317–336, June 1981.

[10] Ekkart Kindler and Hagen Völzer. Flexibility in algebraic nets. In Jörg Desel and Manuel Silva, editors, *Application and Theory of Petri Nets 1998: 19th International Conference, ICATPN'98*, volume 1420 of *Lecture Notes in Computer Science*, pages 345–364, Lisbon, Portugal, June 1998. Springer-Verlag, Berlin, Germany.

[11] Marko Mäkelä. *Maria: Modular Reachability Analyzer for Algebraic System Nets*. Online documentation, `http://www.tcs.hut.fi/maria/`.

[12] Marko Mäkelä. *A Reachability Analyser for Algebraic System Nets*. Licentiate's thesis, Helsinki University of Technology, Department of Computer Science and Engineering, Espoo, Finland, March 2000.

[13] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, USA, 1984.

[14] Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, Urbana, IL, USA, 1949.

[15] Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen and Tino Pyssysalo. PROD reference manual. Technical Report B13, Helsinki University of Technology, Department of Computer Science and Engineering, Digital Systems Laboratory, Espoo, Finland, August 1995.

Compositionality in the GreatSPN tool and its application to the modelling of industrial applications[*]

S. Bernardi, S. Donatelli, and A. Horváth

Dipartimento di Informatica
Università di Torino, Torino, Italy
{bernardi,susi, horvath}@di.unito.it

Abstract

An implementation of compositionality for Generalized Stochastic Petri Nets (GSPN) and for Stochastic Well-formed Nets (SWN) has been recently included in the GreatSPN tool. Given two GSPNs (or SWNs), and a labelling function for places and transitions, it is possible to produce a third one as superposition of places and transitions of equal label, for SWN colour domains and arc functions have to be treated appropriately.

The main motivation for this extension was the need to evaluate a library of fault tolerant "mechanisms" that have been recently defined, and are now under implementation, in a European project called TIRAN.

The goal of the TIRAN project is to devise a portable software solution to the problem of fault tolerance in embedded systems, while the goal of the evaluation is to provide evidence of the efficacy of the proposed solution. Modularity being a natural "must" for the project, we have tried to reflect it in our modelling effort.

In this paper we discuss the implementation of compositionality in the GreatSPN tool, and we show its use for the modelling of one of the TIRAN mechanisms, the so-called *Local Voter*.

1 Introduction and motivations

TIRAN (Tailorable fault tolerance framework for embedded applications) is an European ESPRIT project, involving six European partners from industries and universities, that is defining and implementing a new approach to fault tolerance in embedded systems. The TIRAN solution is built around a software solution which provides fault

[*]We acknowledge contribution of the EEC project 28620 TIRAN.

tolerance capabilities to automation systems. TIRAN basically consists of a library of functions that add fault tolerant behaviour to software, a support for their execution, and a language to specify how to react to errors: this is what is called "the TIRAN framework." The framework is meant to allow application programmers to equip their programs with a variety of *software-based* solutions for fault masking, error detection, isolation and recovery. The framework is currently under development on different hardware platforms, while two pilot applications are being developed to test the framework[5].

Modelling in TIRAN is meant for validation, evaluation, and library documentation. Models are built out of a software specification document. Validation is done with respect to a number of "environment scenarios", that include application model, fault model and recovery action specification.

The role of the modelling team in the project is to provide models of the single mechanism and of the system software, in such a way as to be easily composed with models of the different target applications developed by the partners as test cases for the library. Moreover each mechanism is usually a set of tasks that interact through the communication primitive offered by the run time support. Due to the high complexity of the problem, coloured nets [18, 19] have been used for most mechanisms. Important required feature of coloured nets include efficient solution mechanism [7, 8], as well as modularity, reuse and easy modifications of models.

Figure 1 shows the compositional approach to TIRAN modelling. Starting from the specification of the components (whether they are TIRAN mechanisms or hardware specification, or user behaviour) a model per each component is built. The different models are then integrated into a single GSPN/SWN model for evaluation of the whole TIRAN framework. There are a number of boxes all tagged as "model construction," but they may actually require different construction techniques since the input specification are of different types: the specification of the mechanisms is done with UML diagrams (typically state diagrams and message charts) plus some textual comments, fault specification is instead taken from the requirements specification document, that describes the type of faults and the type of the affected component in a semi-formal language, hardware specification has not been included yet, while the user specification is again based on the requirement specification document.

Most of the mechanisms are built as a collection of tasks, and usually there is a state diagram per task, plus a specification of the interactions among them: the corresponding models are built using GSPN/SWN and the compositional facility of GreatSPN described in this paper.

The construction of the integrated model is, in general, a more complicated task, since this is where model reuse really comes into play. To adequately support this composition the PSWN class [1] has been recently defined, that allows the definition of parametric colour classes, and for which a compositional operator has been defined that allows to import and export values and types. Unfortunately no implementation is yet available for PSWN.

SWN were a natural choice for their efficient analysis techniques both for state

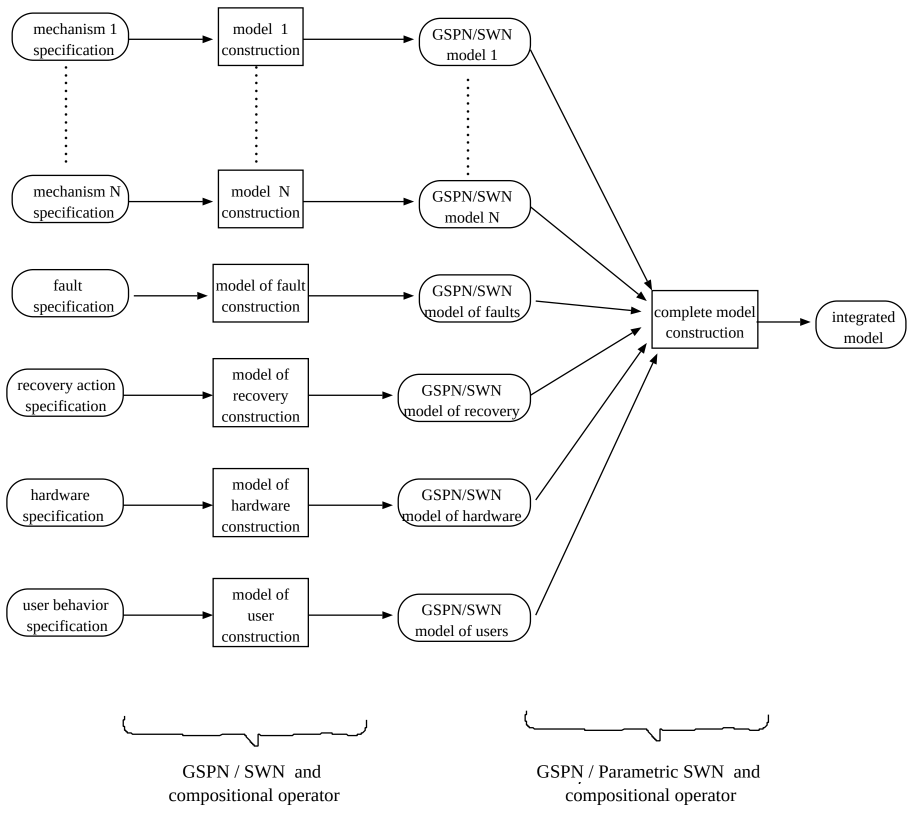

mechanism 1 specification
model 1 construction
GSPN/SWN model 1
mechanism N specification
model N construction
GSPN/SWN model N
fault specification
model of fault construction
GSPN/SWN model of faults
recovery action specification
model of recovery construction
GSPN/SWN model of recovery
hardware specification
model of hardware construction
GSPN/SWN model of hardware
user behavior specification
model of user construction
GSPN/SWN model of users
complete model construction
integrated model
GSPN / SWN and compositional operator
GSPN / Parametric SWN and compositional operator

Figure 1: Compositional modelling in TIRAN

space generation and for performance evaluation based on aggregated Markov chains, but the tool GreatSPN [9], that nicely supports state space generation exploiting symmetries, steady state computation exploiting lumpability, and discrete event simulation with confidence interval computation, has also a number of weak points, since there is: **a**) no support for modularity, **b**) very few tools for debugging the model (no invariant computation or check, no reachability analysis is possible apart from checking the properties that are more relevant for performance evaluation, typically the presence of an home state, or to do an inspection of the reachability graph written in ASCII form), and **c**) no concept of a parameterized "library of models".

To overcome this weak points a number of activities are planned and/or are under implementation by the GreatSPN group, in particular: 1) to implement composition over places and transitions for GSPN and SWN 2) to implement PSWN, recently defined in [1], to allow reuse of coloured models in different contexts 3) to export GreatSPN model to PROD nets [23], so as to apply all PROD tools for reachability analysis 4) to export GreatSPN model to AMI-nets [12], so as to apply the P-invariant computation for coloured nets [11] that is available in CPN-AMI [12]. 5) to extend the definition and to implement the $\mathcal{PSR}$ methodology [13] for the modelling of layered hardware and software architecture.

This paper discusses the implementation of the first point, and shows its use in a non-trivial modelling case taken from the TIRAN library: a mechanism to implement application replication and voting.

Of the remaining points, the export to PROD nets is at the latest stages of implementation, while the export to AMI-nets has not started yet. For what concerns PSWN, their implementation is a non trivial extension of the composition operator that has been implemented. Indeed the compositional rule implemented is not very sophisticated from the point of view of the treatment of the colour classes: there is no concept of parametric colour classes, nor of import and export values and types, as it is the case in the PSWN class defined in [1].

There are a number of techniques proposed in the literature for the composition or compositional analysis of high level models: [15, 10, 2, 6, 4, 20], and a very thorough survey of these methods can be found in [20], but in this paper we concentrate only on tool support for compositional construction, and not for compositional analysis.

To the best of our knowledge there is no other tool offering the possibility of composing stochastic coloured nets based on labelling, although composition based on labelling is a well established technique for Petri nets [3], and there is an implementation available for a class of high level nets called M-nets [4], that do not include a notion of time, and that have been used to provide a semantic to the programming language $B(PN)^2$ [4].

However there are tools to assist modular modelling also in a stochastic context. CPN-AMI gives the possibility to paste modules next to each other in one model and use the modular services of the graphical interface for the fusion of places or transitions [12]. Design/CPN makes use of hierarchy to assist the user to build complex models, a transition that represent a complex activity may be replaced by a subnet [14], and a similar approach is taken in HiQPN [17], that offers also a larger number of performance evaluation features. UltraSAN models may be combined using the operations *replicate* and *join*, these operations provide common places that can be used for communication between the submodels [22].

The choice of GreatSPN was taken based on practical consideration (like the availability of the source code and of their knowledge), but mainly because of the need to have aggregated symbolic state space generation like the one provided by GreatSPN for SWN (the same analysis is also provided by CPN-AMI, but through an export to GreatSPN). An example of the importance of aggregated, symbolic state space generation will be given in the example section.

Section 2 introduces the compositional rule for SWN and its implementation in GreatSPN. Section 3 describes the Local Voter mechanism, as given in the TIRAN specification document, while Section 4 shows and discusses the SWN models of the local voter. Section 5 concludes the paper.

2 Composition for SWN in the GreatSPN tool

2.1 Composition of two labelled SWN

The composition rule proposed is centered around the classical idea of "matching labels": transitions and places are labelled and pairs of transitions (places) with matching labels are superposed. Let L_T (L_P) be the set of labels for the set of transitions T (of places P), then we can define the labelling function λ, under the hypothesis that L_T and L_P are disjoint, as:

$$\lambda(x) = \begin{cases} T \longrightarrow \mathcal{P}(L_T) & \text{if } x \in T \\ P \longrightarrow \mathcal{P}(L_P) & \text{if } x \in P \end{cases} \tag{1}$$

that implies that more than one label can be associated with a single place or transitions (we call this aspect "multilabelling", and it should not be confused with multilabelling in Petri Boxes [3], that refers to bag of labels). The formal definition of superposition is not given here for space restriction reasons, since it is a simplification of the one presented in [1] (that does not consider parametric colours), while we recall here, through examples, the main ideas.

First, we concentrate on the case in which only transitions are superposed. Let $\mathcal{N} = \langle P, T, \mathbf{Pre}, \mathbf{Post}, \mathbf{Inh}, \mathbf{pri}, \mathcal{C}, cd, \mathbf{w}, \lambda\rangle$ be a labelled SWN obtained by the composition of two labelled SWN $\mathcal{N}_1$ and $\mathcal{N}_2$; then the elements of $\mathcal{N}$ are obtained as follows.

The set of places P is the union of the sets of places, i.e. $P = P_1 \bigcup P_2$ (renaming of place names may be necessary in order to avoid matching names). The colour domain function cd gives $cd_1(\mathsf{p})$ if $\mathsf{p} \in P_1$, $cd_2(\mathsf{p})$ otherwise.

The unlabelled transitions are considered non-observable with respect to the composition, and those whose labels do not appear in the other operand, are not involved in superposition. These transitions are simply copied into T (as for places, renaming may be necessary). To show how the operation proceeds to superpose transitions let us assume that $\mathcal{N}_1$ is multilabelled, while $\mathcal{N}_2$ is not, and the labelling is non-injective. Let $T_2(l)$ denote the set of transitions t' of T_2 with $l \in \lambda(\mathsf{t}')$, where $\lambda(\mathsf{t})$ gives the set of labels of t. In $\mathcal{N}$ there will be a replica of $\mathsf{t} \in T_1$ for each element in $\bigotimes_{l \in \lambda(\mathsf{t}), T_2(l) \neq \{\}} T_2(l)$, where $\otimes$ is the Cartesian product. An example is shown in Fig. 2, for transition **t1**: $\lambda(\mathbf{t1}) = \{\mathit{l1}, \mathit{l2}, \mathit{l3}\}$ and the above defined Cartesian product has the elements $\{\mathsf{t2}, \mathsf{t4}\}$ and $\{\mathsf{t3}, \mathsf{t4}\}$. In the composed net **t11** (**t12**) is obtained by superposing **t1**,**t2** and **t4** (**t1**,**t3** and **t4**).

If two arcs connected to different transitions that are involved in the same superposition have identical variable names in their arc expression, then these variables are renamed in the arc expression of all the arcs connected to one of the two transitions. If these variables appear in the guard of the transition whose arcs' expressions are changed, the renaming is performed in the guard as well. As an example, in Fig. 2, during the superposition of **t1**, **t2** and **t4** the variable x of the arcs and guard function connected to **t2** is renamed to $x1$. (As it will be mentioned in Section 2.2 the implemented version of the algorithm allows the user to override the above described

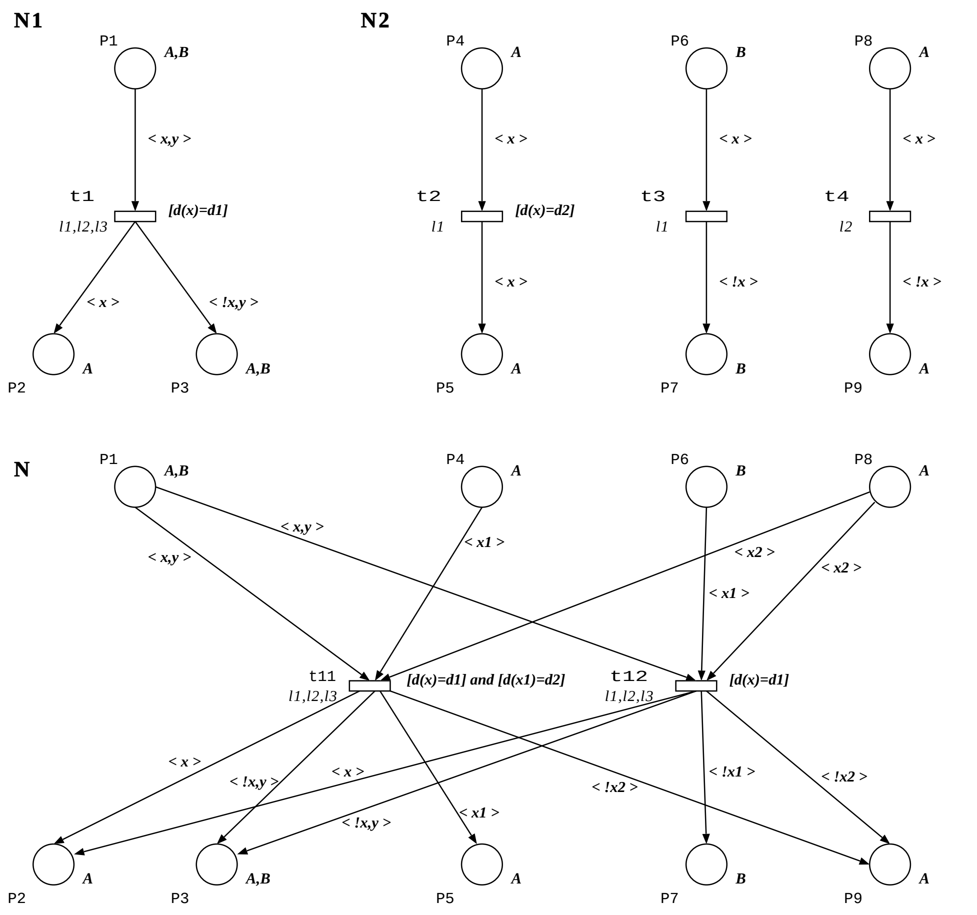


Figure 2: A multilabelled, non-injective example

renaming rule to "unify" values of the nets.) When two superposed transitions have both a guard function these guard functions are joined with logical *and* relation.

The matrices $\mathbf{Pre}, \mathbf{Post}, \mathbf{Inh}$ describing the arc structure of $\mathcal{N}$are built in the following way. The arcs of $\mathcal{N}_1$ and $\mathcal{N}_2$ connected to transitions that are not involved in superposition are simply copied into $\mathcal{N}$. An arc connected to a transition involved in superpositions will have as many instances as the times the transition is superposed. In our example the arc P1-t1 has two instances in the composed net: P1-t11 and P1-t12.

The priority function **pri** gives the same value as before for the transitions that are not involved in superposition. A transition resulting from superposition inherits the priority value from the involved transition of $\mathcal{N}_1$. The weight function **w** is handled similarly to the priority one. We assume that there are not marking dependent weights and rates, and we basically leave the user the task of redefining **pri** and **w** for the final net, since compositional ways to handle **pri** and **w** are still an open question, although some attempts to address this problem may be found in [16, 21].

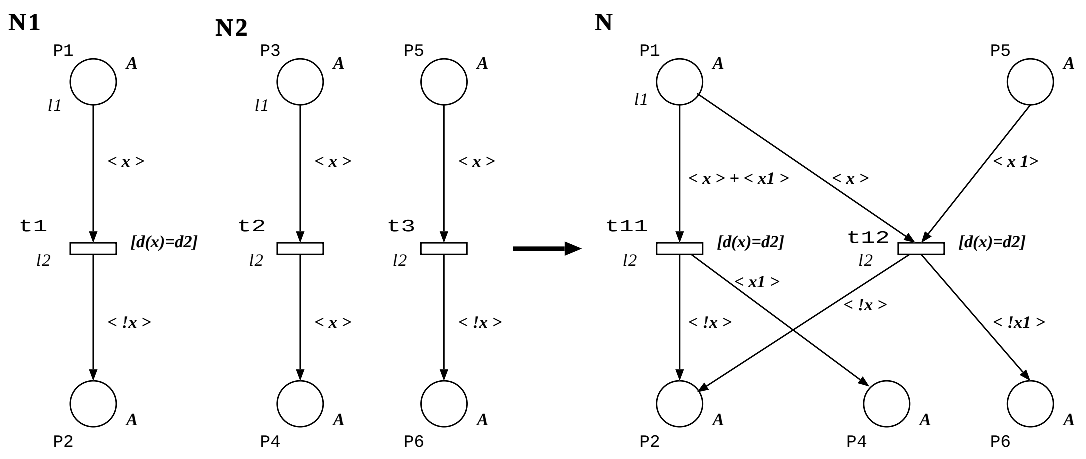


Figure 3: Superposition of places and transitions

The set of basic colour classes $\mathcal{C}$ and their definitions are assumed to be common for $\mathcal{N}_1$ and $\mathcal{N}_2$.

The operation to superpose places is the direct counterpart of the operation described above, with the additional, obvious, constraint, that places of equal label should have the same colour domain. However the superposition of places is less complicated as it does not require renaming of arc or guard expressions.

The simultaneous application of superposing places and transitions has two features that were not shown in the above description. First, having an arc whose place (transition) is involved in n_p (n_t) superpositions, there will be $n_p \cdot n_t$ instances of the arc in the composed net connecting all the instances of its place with all the instances of its transition. Second, having two arcs whose places and transitions are superposed, the arc expressions of these two arcs are added. An example for the latter is shown in Fig. 3 where the arc expressions of the arcs $\mathsf{P1} - \mathsf{t1}$ and $\mathsf{P3} - \mathsf{t2}$ are summed.

2.2 Implementation

The compositional operators described in the previous subsection are implemented by a program called **algebra**, that uses and produces SWN nets in GreatSPN format. The modeller may build the component nets using the graphical interface of GreatSPN. Since the present interface does not allow to define labels, they are encoded in the name of the transitions and places. Both transition and place names have the structure tag|label1|label2..., where tag is the name of the transition or place followed by its labels separated by bars.

The user may define the set of labels which will be taken into account during the composition. This feature may be useful when composing more than two nets and the modeller wish to avoid that all labels are considered at all stages of the composition.

Right now **algebra** is able to deal with non-injective labelling in both operands,

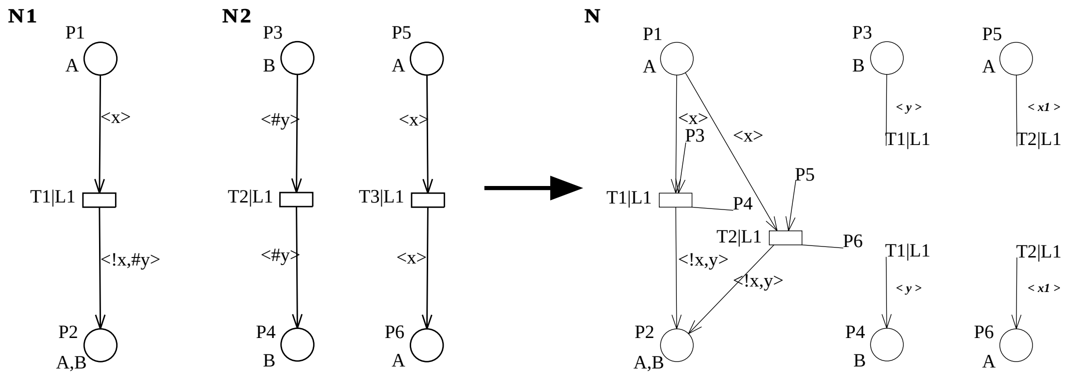


Figure 4: Superposition using GSPN

while only one of the two may be multilabelled. This was judged an adequate compromise between the complexity of the implementation and the foreseen use of the operator (this choice is adequate, for example, to support the implementation of the $\mathcal{PSR}$ methodology cited before as point 5 of the "wish-list" of GreatSPN).

When composing **algebra** attempts to create a well-readable net. The "shape" of the original components are maintained. The user may define where the individual components are placed in the composed model. If a transition (or place) has multiple instances in the resulting net, the additional instances are placed around the original position of the transition. When, as a result of the composition, an arc's place and transition are in different subnets the arc is drawn as "broken arc" in the resulting net. A small example for the output of the program is given in Fig. 4. GreatSPN does not draw arc expressions on broken arcs, those have been written on the figure "by hand".

Fig. 4 demonstrates another feature of the program: if a variable name starts with the character #, it is not renamed during the superposition. This allows the modeller to use the same variables in different components, so as to "unify" values.

algebra may be called from the command line by

```
>> algebra net1 net2 op labels net [placement shiftx shifty]
```

The two operands are `net1` and `net2`, the resulting SWN is `net`. The operator is defined by `op` and may be `t` to superpose transitions, `p` to superpose places or `b` to superpose both places and transitions. The set of labels over which the superposition will be performed may be given in the file `labels`, this file has the following format:

```
transition={tl1|tl2}
place={pl1|pl2|pl3}
```

The labels that are not given in this file are not considered during the operation. If the file does not exist all labels are considered. The last three arguments may be used to define the placement of the components: if the parameter `placement` is 1 (2) the two nets are placed next to each other horizontally (vertically), if it is 3 the second net is shifted by (`shiftx,shifty`) compared to the first net.

3 The local voter specification

The Local Voter mechanism (LV) aims at masking the occurrence of faults during the execution of a piece of code of an application process. Fault masking is achieved by the adoption of a spatial redundancy of the execution of the piece of code and by the voting on the results coming from the replicas.

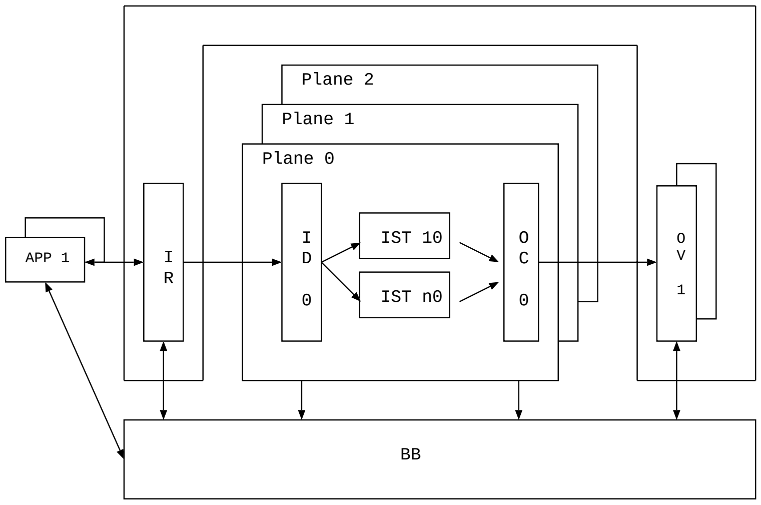


Figure 5: A description of the local voter

Depending on the voting technique adopted in the LV and on the spatial redundancy, a limited number of faults may be masked; for instance, by using a majority voting algorithm and by running concurrently N replicas, up to $[\frac{N-1}{2}]$ faults can be made transparent for an application process.

Figure 5 shows a graphical representation of LV taken from the specification document of TIRAN; the LV can be used concurrently by several application processes and three replicas are considered per application.

The replicas are executed on separate "planes", that naturally correspond to separate processing nodes. The application process APP_i that uses the LV mechanism is split in two parts, a part that does not require a replicated execution, and a part that instead requires it. If there are n applications that can use LV, then each application has its distinct piece of code to be executed.

Since each replica, called IST_{ij} in the figure, should receive the same input data there is a task IR (input replicator) that performs a replica over the three planes of the input data: the data do not go directly to the application replica, but to the input dispatcher of the corresponding plane ID_j,that takes care of passing it on to the right application.

A similar approach is used to collect the results: when a replica IST_{ij} ends its computation it sends its output data to the output collector of the corresponding

Acr.	description	no. of them
APP	application	n
IR	input replicator	1
ID	input dispatcher	3
IST	replicated software to vote upon	$3 * n$
OC	output collector	3
OV	output voter	n
BB	backbone	–

Table 1: Acronyms

plane OC_j, that takes care of forwarding it to the appropriate voting task OV_i; there is one voting task per application.

The components of the local voter interact with the backbone BB, that is a sort of run-time support for the TIRAN library of mechanism, that handles all exceptions as well as the recovery actions. All interactions among tasks are based on mailbox communications.

Table 1 lists the acronyms used in the paper for the different tasks, and for each task lists how many copies of that task there are in a LV that serves n applications.

The OV behaviour is described by a state diagram in the specification document, that basically amounts to a 2-out-of-3 voting. An additional textual specification also states that, as soon as OV_i for application APP_i receives the first output from one of the replicas, it sets a time-out for receiving the other replicas. Each OV sends a message back to the corresponding application only if all three replicas are received before the time-out expires, and there is a match 3-out-of-3 of the results. In all other cases BB comes into play.

4 The SWN model of the local voter

The following assumptions were made to model LV: tasks communicate in an asynchronous manner via mailboxes, and there is one mailbox for each ordered pair of tasks, time required to prepare a message is in general negligible, while the time to actually transmit it from the task output buffer to the recipient mailbox is not. For what concerns the graphical representation, we have used grid places to emphasize mailboxes and shadowed boxes to delimit portions of the nets that corresponds to "recovery actions", and that will be explained in the next subsection.

Three colour classes have been defined:

AP is the colour class of applications that can request a replicated execution of a piece of their code, and it is defined as $AP = \mathrm{u}App$, that is to say a single static subclass, unordered, defined as $App = \{ap1, .., apn\}$;

P is the colour of the planes, there are always three planes in LV, therefore $P = \mathrm{u}Pl$, and $Pl = \{pl1, pl2, pl3\}$;

Exc is the colour used to distinguish the positive or negative outcome of a LV activity, and it is built out of two static subclasses $Exc = \mathrm{u}Ecx1, Ecx2$, where $Exc1 = \{e1\}$ means that there has been a time-out expiration, while $Exc2 = \{e2\}$ means that there was no time-out expiration.

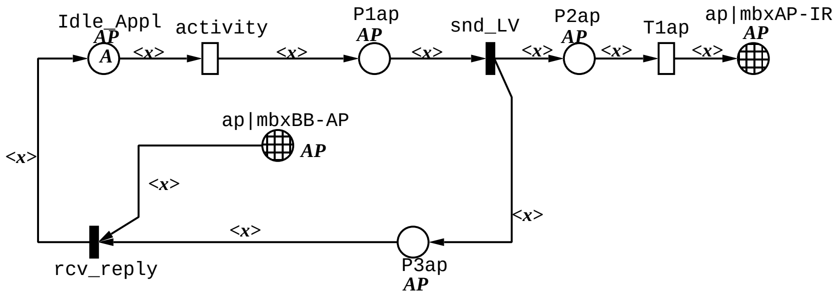


Figure 6: The application model

Figure 6 shows the SWN model of the application, that cyclically executes its own activity, sends a message to the task IR of the local voter, and waits for a message coming from the backbone BB.

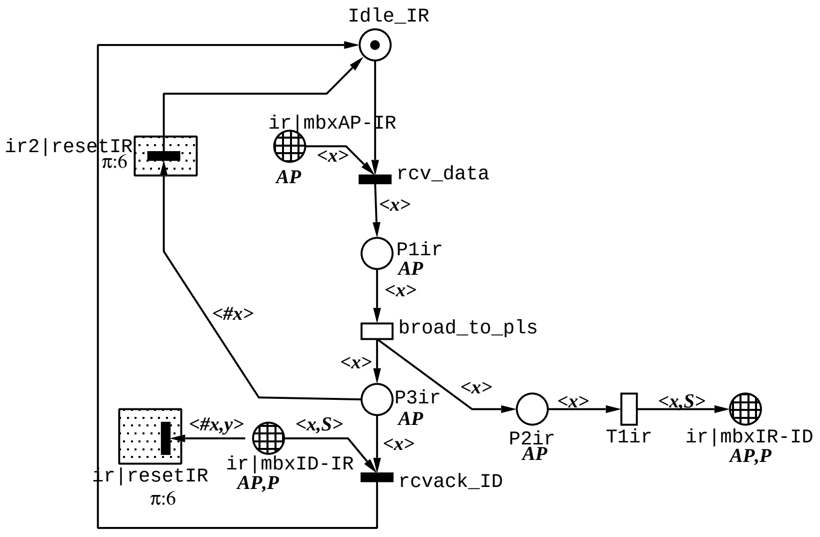


Figure 7: The input replicator model

Figure 7 shows the SWN model of the input replicator IR: it waits for messages coming from the applications. As soon as a request of replicated execution for application App_i is received, it broadcasts it to the input dispatcher task of each plane, and waits for an acknowledge from ID. Since there is only one IR task, then no colours are needed.

Figure 8 shows the SWN model of the input dispatcher ID. There is one task ID for each of the three planes: it receives from IR the identity of the application to be executed, it acknowledges reception to IR, and it activates a task corresponding to the requested replica (called instance) for that plane.

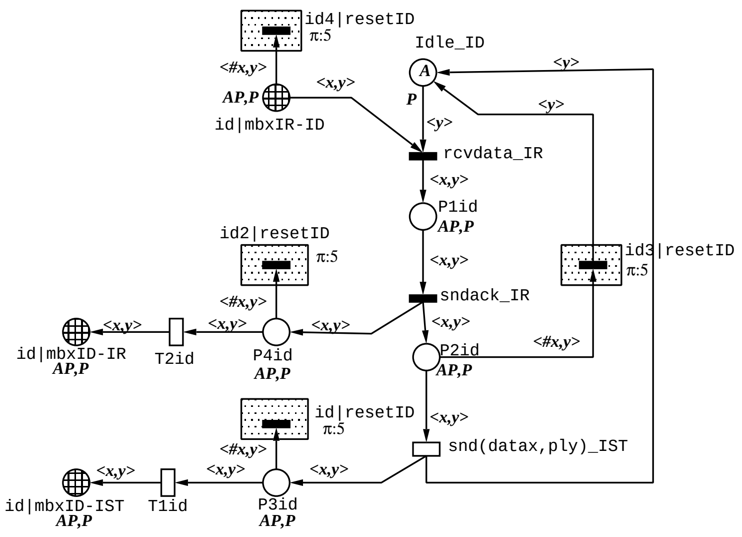


Figure 8: The input dispatcher model

Figure 9 shows the SWN model of the replica of the code to be executed on the different planes: since the TIRAN framework allows only static tasks, then it is correct to assume that all replica are activated at the beginning and then suspend themselves waiting for a message from the ID tasks. There are $|AP| \times |P|$ instances, i.e. one for each application and for each plane. Each instance (x, y) waits for a message (x, y) from ID y. When a message (x, y) is received the instance of application x on plane y starts its activity, modelled by timed transition comp, and then sends the result of the computation to OC.

Figure 10 shows the SWN model of the output collector (OC): there is one such process for each plane, and each OC waits for a message coming from the replicas running on its plane, and it forwards it to the output voter. According to the textual portion of the specification, the OC should wait for a "ready message" from OV, but since the conditions under which OV should send this ready message are not specified, we assume that OV is always willing to accept messages in its mailbox.

Figure 11 shows the SWN model of the output voter task OV: there is an OV for each application that can use LV. Each OV executes the voting algorithm (majority voting 2 out of 3) on replicas of the same application, independently from the others. OV waits for the replicas outcome from the three different planes. As soon as the first outcome is received, a timeout for reception of the other two replicas outcome is set. Then three situations may occur:

C1 all the three outcomes are received before the time-out expiration, i.e. transition recv3noTO fires and voting on the three outcomes takes place;

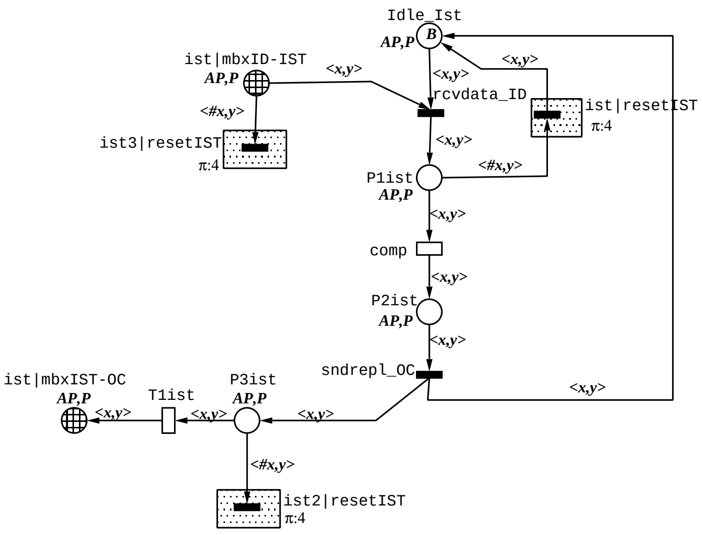


Figure 9: The model of the replicated code

C2 the time-out has expired and two of the three outcomes have been received (firing of transition recv2&TO), and a vote on the two replicas takes place;

C3 the time-out has expired and only one of the three outcomes has been received by OV, i.e. firing of transition recv1&TO.

Under condition C1 a message of exception of type $e2$ (no time-out has occurred) is sent to the the backbone BB; in cases C2 and C3 a message of exception of type $e1$ (a time-out has occurred) is sent to BB. Observe that we are not passing on to the backbone the information on whether the vote was successful or not, although this will be a trivial extension, since the success or failure of the 2-out-of-3 algorithm, according to the state diagram of the specification document, is modelled in detail in the SWN of Figure 11.

When the message is sent to BB, OV waits for an acknowledge from BB to return back into its idle state. Observe that we are assuming that no direct answer goes back directly from OV to APP, not even in the case of a "normal" 3-out-of-3 voting, since we impose that all restarted are caused by BB.

Figure 12 shows the SWN model of the Backbone task, or, more precisely, of that part of BB devoted to interactions with LV. BB is in an idle state until it receives an exception message coming from OV. If the exception is of type $e2$, i.e. no time-out has occurred, then BB sends an acknowledge to OV and to the application. If instead the exception is of type $e1$, then a time-out has occurred, and therefore a reset operation is needed, before sending back the messages to OV and to APP.

4.1 Local voter without recovery actions: an open model

A first analysis was performed for the case of a "single run" for each application. In order to obtain the complete model the single nets have to be composed using the

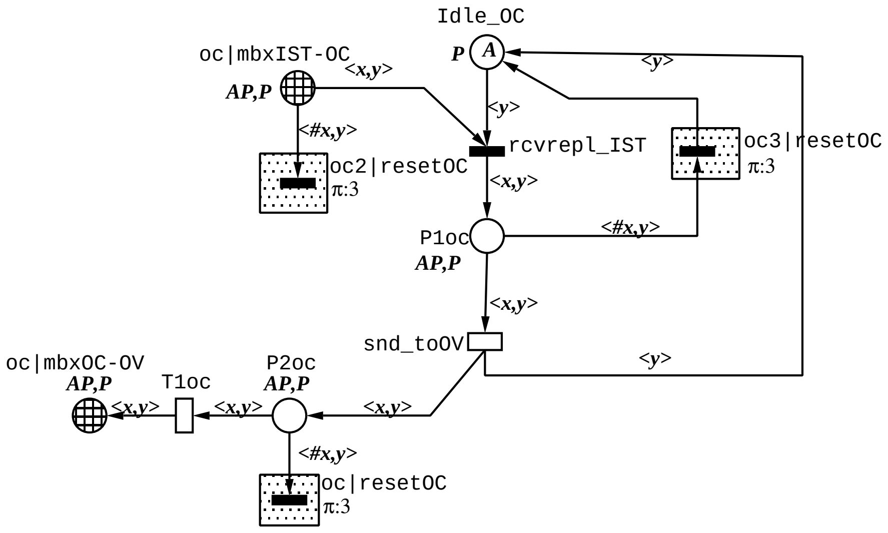


Figure 10: The model of the output collector

program **algebra** explained in Section 2. The nets used are the one without shadowed portion, so that no message is passed from OV to BB, so that each application is executed only once. The resulting SWN net has been solved, for the single application case, using the symbolic reachability graph construction of GreatSPN, that produces 589 tangible states corresponding to 4261 ordinary ones.

There are 3 symbolic dead markings, corresponding to 7 ordinary dead markings. Each marking is described in terms of dynamic colour subclasses associated to places, and, for each dynamic subclass, its cardinality is given. For each symbolic marking the number of corresponding ordinary markings is given; for instance, the following dead marking:

```
MARKING D856 # ordinary marking: 3 (dead)
Idle_BB(1) oc|mbxOC-OV(1<App0,Pl1>) Idle_OC(1<Pl0>1<Pl1>) Idle_IR(1)
Idle_ID(1<Pl0>1<Pl1>) Idle_Ist(1<App0,Pl0>1<App0,Pl1>) P3ap(1<App0>)
Exception(1<App0,Exc10>)
|Exc10|=1 |Exc21|=1 |App0|=1 |Pl0|=1 |Pl1|=2
```

corresponds to a case of time-out expiration: only one replica has been received by OV and the time-out has expired while waiting for the remaining replicas. All components, except OV and APP, are in their initial states (idle state), APP and OV are both waiting for a msg from BB, that will, of course, never arrives. Observe that symbolic marking provides a more abstract information with respect to ordinary one: in this case the abstraction is on the identity of the replica (plane) that has finished first. Other deadlocks represent the case of reception of all the three replicas before the time-out expiration, corresponding to an ordinary deadlock marking, and the case of time-out expiration after two replicas have been received, corresponding to three ordinary markings.

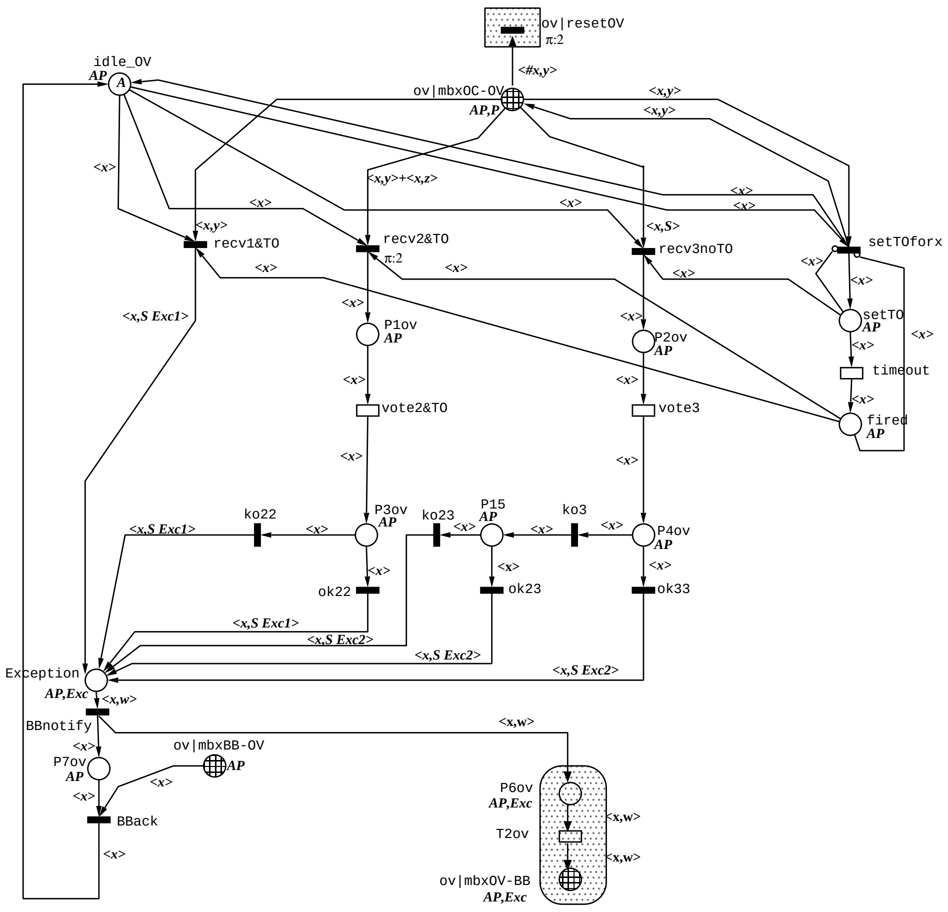


Figure 11: The model of the output voter

4.2 Local voter and recovery actions: an ergodic model

All the deadlocks found describe a "good" (expected) behaviour, so that it makes sense to proceed to add also the reconfiguration activities needed to restart an application. The model obtained composing all nets, including also with the shadowed portions, is ergodic (there is a single strongly connected component)

The basic idea is that the recovery action taken by BB is:

- to remove messages from mailboxes that refer to the application that has signaled the exception;
- to take the corresponding tasks back to the their initial states.

To accomplish this BB enables a number of immediate transitions, one per model component, and they are labelled in such a way as to superpose with the cleaning transitions in the model components. Observe that these transitions are assigned a different priority, mainly to avoid the generation of useless interleavings, that could significantly slow down the state space generation.

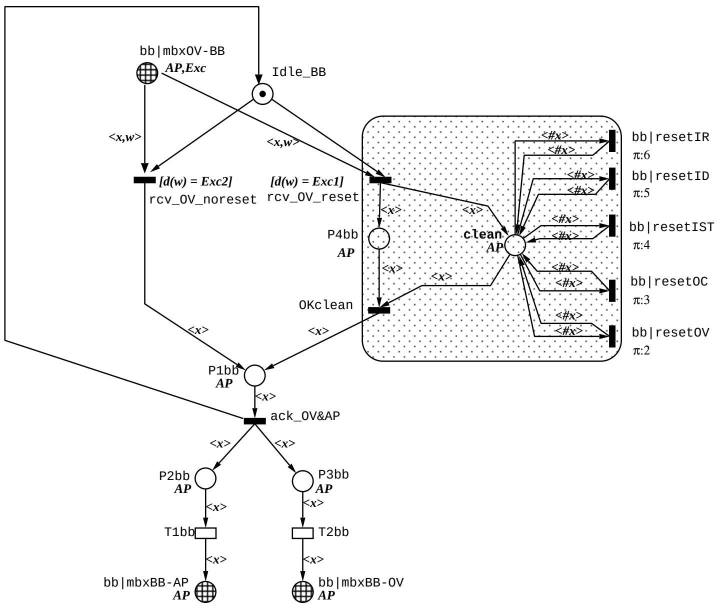


Figure 12: The model of the backbone

The symbolic reachability graph for the single application case has 1452 tangible states, while the ordinary one has 7074. Initial marking is a home state. The generation took a few minutes on a rather standard 64Mbyte Pentium 2 machine.

Since the running time of the reachability graph generation grows very significantly into an almost trashing situation even when increasing only by 1 the number of applications, then we have tried the discrete event simulation available in GreatSPN, to compute performance indices for our LV model while varying the number N of applications. Table 2 shows the list of timing parameters to be assigned to transitions to perform a simulation. For a timing assignment in which values differ for up to one order of magnitude the simulator was able to provide estimates for throughput and mean value of tokens in places in a few minutes, for an accuracy level set to 10 percent and a confidence level of 0.95. Table 3 shows the results obtained varying the number of application processes that use the LV mechanism in the range $[1, 10]$.

4.3 Local voter without recovery actions and explicit faults

In the models considered up to now no faults are explicitly included in the model, so that a time-out can expire only due to a delay in the completion of one of the replicas. To this goal the model of IST has been modified to include a timing transition that models the fault and that takes IST into an error state place. The resulting model, for the single application case, has 1185 symbolic tangible markings (corresponding to 6008 ordinary ones) and there are 7 symbolic dead markings, corresponding to 20 ordinary dead markings. Among them a very interesting one is the marking that represents the state of the model where all the instances are in an error state, and

Param.	Description	Value(ms.)
act	time of normal operations of an application	1
tr_toIR	transmission time of a msg to IR	0.1
broad	time for IR to broadcast to IDs	0.1
snd_toist	time for ID to send a msg to a same-plane istance	0.1
tr_acktoIR	transmission time of an ack to IR	0.1
tr_toID	transmission time of a msg to IDs	0.1
tr_toIst	transmission time of a msg to Ists	0.1
comp	time spent by the istances to perform operations	2
tr_toOC	transmission time of a msg to OCs	0.1
snd_toov	time spent by OC to send the replica to OV	0.1
tr_toOV	transmission time of a msg to OVs	0.1
TO	time-out	5
vote3	time spent by OVs to vote on 3 replicas	0.1
vote2&TO	time spent by OVs to vote on 2 replicas	0.1
tr_toBB	transmission time of msg to BB	0.1
tr_BB-OV	transmission time of an ack from BB to OV	0.1
tr_BB-AP	transmission time of a reply to application.	0.1

Table 2: Timing parameters

No.Appl.	$X_{activity}$	Acc.(%)	X_{rcv_reply}	Acc.(%)	N_{P2ir}	Acc.(%)
1	0.211637	5.092404	0.211637	5.092404	0.020518	9.141612
2	0.419355	0.483286	0.419433	0.488460	0.041899	1.311778
3	0.625945	0.663775	0.626182	0.666935	0.062012	1.594000
4	0.827432	1.076573	0.827014	1.103949	0.081316	1.966425
5	1.025215	1.037140	1.027469	1.023215	0.101616	2.34418
6	1.227929	1.227031	1.228012	1.141775	0.120755	4.246115
7	1.413914	0.904866	1.414766	0.911834	0.145302	3.139288
8	1.595689	2.375840	1.596397	2.388058	0.163623	6.134849
9	1.779154	1.517227	1.777283	1.534383	0.177373	4.467230
10	1.952531	3.056724	1.949649	2.366157	0.206051	7.254697

Table 3: Simulation results

this correspond to a case in which no replica will ever reach OV, so no time-out will be set. This case is, up to now, not considered in the specification document.

Observe that, to produce an ergodic model it will not be enough to consider the reset activity of BB, but an explicit testing for the three replicas all being in an error state should be added to the model.

5 Conclusions

In this paper we have described the compositional operator that is now implemented in GreatSPN for the superposition over labelled places and transitions for GSPN and SWN nets, and its application for the study of a mechanism for fault tolerance called local voter. Tool support was fundamental in the model developing phase, since specification were still changing and the definition of the abstraction level for modelling was not set.

To provide more flexibility to these models we should include a modular definition of the communication (for example to investigate rendez-vous instead of mailboxes), and different models of fault.

Another open point is how to take into account the hardware on which the applications and the framework will run: indeed for this point we are planning to extend the definition of $\mathcal{PSR}$ to SWN models. It seems envisionable that the size of the models will only allow simulation since the models that we have produced are quite detailed as they are meant for validation, while an interesting open problem is how to derive from these models some more compact ones to be used for performance evaluation purposes.

We have already mentioned in the introduction that there is very little support for reachability analysis right now in GreatSPN, so that, if a net has an error that make some place unbounded, the state space generation simply does not stop. Indeed before running the state space generation we have always run simulation, that allows, by checking the accumulated performance indices, to check whether certain place markings grow in a suspicious manner.

Acknowledgements. We would like to thank the anonymous reviewers for their very careful reading of the paper, and for pointing out various ways to improve it.

References

[1] P. Ballarini, S. Donatelli, and G. Franceschinis. Parametric Stochastic Well-formed Nets and compositional modelling. In *Proc. of the* 21st *International Conference in Application and Theory of Petri Nets, ICATPN 2000*, Aarhus, Denmark, June 2000. Springer Verlag. LNCS, to be published.

[2] E. Battiston, O. Botti, E. Crivelli, and F. De Cindio. An incremental specification of a hydroelectric power plant control system using a class of modular algebraic

nets. In *Proc. of the* 16^{th} *International Conference on Application and Theory of Petri nets 1995*, Torino, Italy, 1995. Springer Verlag. Volume LNCS935.

[3] E. Best, R. Devillers, and J. Hall. The Petri box calculus: a new causal algebra with multilabel communication. In G. Rozenberg, editor, *Advances in Petri Nets*, volume 609 of *LNCS*, pages 21–69. Springer Verlag, 1992.

[4] E. Best, H. Flrishhacl ANF W. Fraczak, R. Hopkins, H. Klaudel, and E. Pelz. A class of composable high level Petri nets with an application to the semantics of $B(PN)^2$. In *Proc. of the* 16^{th} *International Conference on Application and Theory of Petri nets 1995*, Torino, Italy, 1995. Springer Verlag. Volume LNCS935.

[5] O. Botti, V. De Florio, G. Deconinck, R. Lauwreins, F. Cassinari, A. Bobbio, S. Donatelli, A. Lein, H. Kufner, E. Thurner, and E. Verhulst. The TIRAN approach to reusing software implemented fault-tolerance. In *Proc.* 8^{th} *Euromicro Workshop on Parallel and Distributed Processing (PDP2000)* , Rhodos, Greece, Jan. 2000.

[6] P. Buchholz. A hierarchical view of GCSPN and its impact on qualitative and quantitative analysis. *Journal of Parallel and Distr. Computing*, (15), July 1992.

[7] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. On Well-Formed coloured nets and their symbolic reachability graph. In *Proc.* 11^{th} *Intern. Conference on Application and Theory of Petri Nets*, Paris, France, June 1990. Reprinted in *High-Level Petri Nets. Theory and Application*, K. Jensen and G. Rozenberg (editors), Springer Verlag, 1991.

[8] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. A Symbolic Reachability Graph for Coloured Petri Nets. *Theoretical Computer Science B (Logic, semantics and theory of programming)*, 176(1&2):39–65, April 1997.

[9] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation, special issue on Performance Modeling Tools*, 24(1&2):47–68, November 1995.

[10] S. Christensen and L. Petrucci. Modular state space analysis of coloured Petri nets. In *Proc. of the* 16^{th} *international conference on Application and Theory of Petri nets 1995*, Torino, Italy, 1995. Springer Verlag. Volume LNCS935.

[11] J.M. Couvreur and J. Martinez. Linear invariants in commutative high-level nets. In *Proc.* 10^{th} *Intern. Conference on Application and Theory of Petri Nets*, Bonn, Germany, June 1989.

[12] *CPN-AMI.* `http://www-src.lip6.fr/logiciels/framekit/cpn-ami.html`.

[13] S. Donatelli and G. Franceschinis. The PSR methodology: integrating hardware and software models. In *Proc. of the 17th International Conference in Application and Theory of Petri Nets, ICATPN '96*, Osaka, Japan, june 1996. Springer Verlag. LNCS, Vol 1091.

[14] *Design/CPN.* `http://www.daimi.au.dk/designCPN`.

[15] S. Haddad and P. Moreaux. Evaluation of high level Petri nets by means of aggregation and decomposition. In *Proc. of the 6th International Workshop on Petri Nets and Performance Models*, Durham, North Carolina, U.S.A, October 1995.

[16] Jane Hillston. The nature of synchronization. In U. Herzog and M. Rettelbach, editors, *Proc. 2nd Workshop on Process Algebra and Performance Modelling*, Erlangen, 1994.

[17] *The HiQPN Software.* `http://ls4-www.informatik.uni-dortmund.de/QPN`.

[18] K. Jensen. *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use. Volume 1.* Springer Verlag, 1992.

[19] K. Jensen. *Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use. Volume 2.* Springer Verlag, 1995.

[20] Isabel C. Rojas M. *Compositional construction and Analysis of Petri net Systems.* PhD thesis, University of Edinburgh, 1997.

[21] E. Teruel, G. Franceschinis, and M. De Pierro. Clarifying the priority specification of gspn: Detached priorities. In *Proc. 8th Intern. Workshop on Petri Nets and Performance Models*, Zaragoza, Spain, September 1999. IEEE-CS Press.

[22] *The UltraSAN Software.* `http://www.crhc.uiuc.edu/UltraSAN`.

[23] K. Varpaaniemi, J. Halme, K. Hiekkanen, and T. Pyssysalo. PROD reference manual. Technical Report Series B, number 13, Helsinki University of Technology, August 1995.