Modelling and Analysis of Distributed Software Using GSPNs

Susanna Donatelli and Giuliana Franceschinis

Dipartimento di Informatica
Università di Torino, Torino, Italy
Phone: +39-11-7429111, Telefax: +39-11-751603
E-mail: {susi,giuliana}@di.unito.it

Abstract. This chapter discusses the role that Generalized Stochastic Petri Nets (GSPN) can play in the static analysis of distributed software. The material is organized along two main lines: the need and the advantages of studying both qualitative and quantitative aspects of a program, and the need for doing it in an automatic manner. The role of performance evaluation in the analysis of distributed software is illustrated through a small example, classical in the qualitative approach (the dining philosophers). Although small this example allows to point out the need and the requirements of automatic translation and to discuss the main hypothesis behind program performance evaluation through GSPN models. A procedure for the automatic generation of GSPN models starting from a distributed program written in a CSP-like language, and for the definition of program performance indices in terms of GSPN ones is then given and illustrated by means of a realistic example.

Keywords: Computer Aided Distributed Software Engineering, Verification, Performance Analysis, Stochastic Petri Nets, Automatic Generation of GSPN models.

1 Introduction

Previous chapters have shown the role that Petri nets can play in the qualitative analysis of systems, and in particular for distributed systems and programs. A previous chapter on timed Petri nets and on Generalized Stochastic Petri Nets (GSPN) [2] has already introduced the concept of stochastic duration of activities associated with transitions, and how it allows to study the performances of the modelled system.

The aim of this chapter is to show the use of GSPN for the integrated qualitative and quantitative analysis of distributed programs, to help the programmer to decide whether its program is correct (does it meet the given qualitative specification?), and it is fast enough (does it meet a given quantitative specification?). In particular we shall discuss the role that GSPN can play in the analysis of distributed programs, their advantages and disadvantages, and the possibility of adopting them as the basic language of a tool that can be used by a programmer with little or no knowledge of Petri nets.

There are two classes of techniques normally used for the analysis of distributed programs: **dynamic** and **static**. Dynamic analysis consists of choosing a set of representative input data to test the programs: program properties are inferred from these sample executions. Static analysis, on the other hand, is a method that draws conclusions on the run time behaviour of the program by "simply" looking at its code without requiring any execution. The two techniques are essentially complementary, and despite the fact that we concentrate our attention here only on static analysis, it is in general a good idea to apply them both, whenever possible.

Dynamic analysis can be used to gain confidence in the existence of qualitative properties, like, for example, the absence of deadlock, by performing a number of runs on different test cases. It has its weak point in assessing the representativeness of the test data and thus in the generality of the conclusions drawn from a set of test cases, added to the difficulty of performing reproducible testing of non-deterministic programs. Moreover, to be run, the program must already exist in an executable form, so that the dynamic approach is not very well suited for an analysis in the early stages of the design.

The classical approach of static analysis is instead to derive from the program a formal model that is subsequently studied to infer the properties of the program. Since during static analysis nothing is known about the run-time behaviour of the program (for example its input data), no assumption is made about which of the possible execution paths is actually followed by the program. Static analysis thus tends to account also for many program behaviours that would never occur in real executions, however all possible run time behaviours are surely included. In particular it is assumed that a static analyser is correct if any deadlock that may appear at run time is detected by static analysis.

Three types of anomalies may be detected by static analysis [20]: unconditional faults, conditional faults, and nonfaults. Unconditional faults correspond to errors that will definitely occur during program executions. Conditional faults represent instead errors whose occurrence depends either on nondeterministic choices or on specific sets of input data. Nonfaults, finally, are errors which are reported by the static analyzer although they will never occur during program executions: nonfaults may appear, for example, when the correct behaviour of programs is ensured by control variables that are ignored during static analysis. Static analysis thus provides a pessimistic picture of program behaviour and, indeed, a measure of efficacy of a static analyser is its ability to reduce the number of nonfaults.

The role that Petri nets can play here is to serve as formal models, since the basic mechanism of concurrency, synchronization and conflict are native in this language. Petri nets are an executable formalism, and indeed we shall make a static analysis of the program by "executing" its Petri net model, that is to say by building the set of reachable states. Petri nets also allow a structural analysis, that does not require to build the state space, and we shall use it whenever possible (the reader can find in [16] a throughout discussion on the use of structural analysis to study deadlock). The usefulness of Petri net models

for the static analysis of concurrent programs was also pointed out in [20] for programs written in languages such as Ada [19], Occam [6] and others, while relations between Petri nets and other concurrent languages and paradigms such as CCS [15] and CSP [14] have been widely studied in the literature [12, 11].

Since our aim is to mix, in a synergetic manner, the qualitative and quantitative analysis, we use as Petri net language the GSPN "dialect", for its peculiar feature of allowing the study of functional and performance properties *on the same model.*

The same distinction existing for qualitative analysis into dynamic and static can actually be envisioned for quantitative analysis as well. As explained in the chapter on Timed Nets, there are two classical approaches to performance evaluation, namely *measurement* and *modelling.* Measurement consists in observing the system to be analyzed under a number of test cases, and to measure in each case the value of a set of performance indices: if the system to be analyzed is a program, an example of index can be the execution time. The measured values are then used to assert the performance of the program. Measurement based analysis has indeed the same advantages and disadvantages as dynamic analysis.

Modelling consists instead in building a model of the system to be analyzed, and to compute on the model a set of performance indices that are then interpreted as performance attributes of the system. The indices are usually computed by building and solving in steady state the Continous Time Markov Chain (CTMC) isomorphic to the state space of the GSPN, or by performing a Montecarlo simulation of the model. The approach here is analogous to the static analysis presented above, in particular the approach based on the steady state solution of the CTMC is similar to state space based static analysis, as the one proposed by Taylor [20] since they are both centered around the construction of the set of reachable states.

It is often the case, however, that state space techniques are not feasible, due to the excessive dimension of the state space. In these cases, GSPNs still play a relevant role, since simulation can be used to estimate performance indices of the model, and therefore of the program, while the observation of the execution of the model, independently of its timed behaviour, allows the programmer to reason about program behaviours using a formal specification, and can also aid the user in visualizing the many possible execution sequences of the program. Indeed the GSPN model of a program, built along the rules of static analysis, summarizes all possible run time behaviours of the program.

In the remaining of this chapter we shall deal with the problem of representing a distributed program with a GSPN model to support the static analysis of qualitative properties, and the performance evaluation of program performance indices, in an *automatic manner.* To reach this goal we shall first discuss, in Section 2, a well known small problem, the dining philosophers with unidentified forks: through this example we intend to show the usefulness of the approach, but also its limits, as well as the need for tools that can support the analysis. Section 3 takes a critical view on the example, to identify its peculiarity, and to show the limits, and some of the advantages, of a GSPN based analysis. At-

tention will be paied to the problem of choosing an adequate abstraction level for our model representation of the reality, in particular for what concerns the modelling of program variables and input values, to the issue of timing (how are mean delays and distributions assigned to transitions), and to the definition, interpretation and significance of performance results. The classical approach to static analysis does not model program variables at all: this may imply a large number of non faults, sometimes so large as to make any type of analysis non feasible. We shall discuss the modelling of variables in Section 3.3. Section 4 discusses in depth the problem of the automatic translation of distributed programs into GSPN models, and shows a non trivial application example.

To be able to show examples, and to discuss the issue of the automatic construction of GSPN models of distributed programs, we refer in this chapter to a specific class of languages that allows an application to be organized as sets of cooperating tasks using a message passing paradigm of the rendez-vous type (realized though a form of synchronous communication over a *channel*). Major examples of languages of this type are Occam, CSP and Ada, and we shall use a CSP-like syntax.

2 An example

We present a very simple and intuitive example to show the goal of the integrated qualitative and quantitative analysis of distributed programs by considering a message passing solution of a variation on the theme of the well known dining philosophers problem: there are *numphil* philosophers and a pool of *numphil* forks, each philosopher needs any two forks to eat. A solution, written in CSP style, is presented in Fig. 1. The syntax *chan_name ? var_name* means to execute an input from channel *chan_name*, and the data value read from the channel is put in variable *var_name*; the symbol ! represents instead an output. The semantics of input and output statements follows the rule of communication in CSP, based on the rendez-vous paradigm: a process that executes an input (output) from *chan_name* is blocked until there is another process that is ready to execute an output (input) on that same channel: input and output have perfectly symmetric behaviour. The par $P_1, P_2, \ldots P_n$ statement activates the n named processes. The alt command represents instead a non deterministic (unspecified) choice among different communication statements. Each communication statement can be *guarded* by a boolean guard: only communication statements whose guards are true can be considered for the choice.

In the example there are three different processes: Forks_Monitor to manage the forks, Philos that represents a generic philosopher, and process root that activates one instance of Forks_Monitor and *numphil* instances of Philos, passing to these processes two channels, grant_fork for the monitor to provide to a philosopher the right to use a fork, and acc_rel for a philosopher to give a fork back to the monitor.

Each philosopher executes in an endless loop two output commands from channel req_chan to acquire the two forks, operations that correspond to a rendez-

```
Philos( chan req_chan, chan rel_chan)
while(true)
    THINK
    req_chan!dummy
    req_chan!dummy
    EAT
    rel_chan!dummy
    rel_chan!dummy

Forks_Monitor( chan grant_fork, chan acc_rel)
int Avail=numphil
while(true)
alt
    (Avail>0): grant_fork?dummy
       Avail--;
    acc_rel?dummy
       Avail++;

root
chan grant_fork,acc_rel
par
    Forks_Monitor(grant_fork,acc_rel)
    for i=1 to numphil Philos(grant_fork,acc_rel)
```

Fig. 1. The code of the philosophers acquiring one fork at a time

vous with the monitor when it executes an input from channel grant_fork . It can then proceed to the "eating" activity, summarized by the macro EAT. The philosopher releases the two forks by doing two output statements on the formal parameter channel rel_chan, operations that correspond to a rendez-vous with the monitor when it executes an input from channel acc_rel.

The monitor process keeps a count of the forks available in variable *Avail*, that is incremented and decremented according to whether a fork has been released or acquired. At each step of the while loop the monitor can receive either a request or a release of a fork: observe that a request of a fork is taken into consideration only when *Avail* is greater than 0.

A GSPN model of the system for *numphil* = 2 is shown in Fig. 2. The left and right portions of the net model the two philosopher processes, while the central part is the monitor. The variable *Avail*, local to the monitor, has been modelled by a place with an initial marking of two. The two alternatives of the alt statement of the monitor (communication on grant_fork or on acc_rel) give rise to four possible rendez-vous each, modelled by the eight immediate transitions of the model; notice that the four left immediate transitions, that represent the acquisition of forks, test and modify the place that models variable *Avail*,

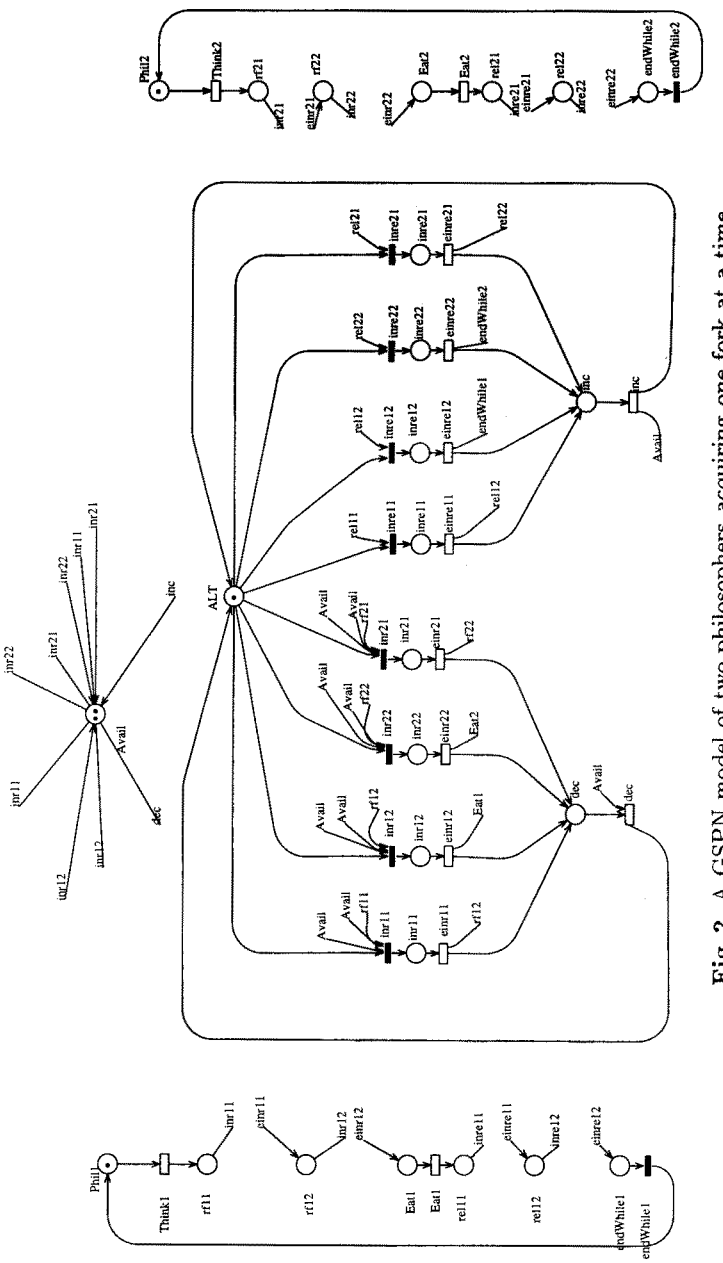


Fig. 2. A GSPN model of two philosophers acquiring one fork at a time

the place is instead only modified in the case of the rendez-vous on the acc_rel channel.

The GSPN model of Fig. 2 can now be studied to infer qualitative and quantitative characteristics of the program.

A structural analysis of the model individuates 4 P-semiflows, and 2 T-semiflows. All places are covered by at least a P-semiflow, and therefore each place of the model is bounded, and, consequently, the net is bounded as well. Table 1 shows the P-invariants computed from the P-semiflows. The first three invariants have the same interpretation: P_1 represents all possible states of the first philosopher process (left in the picture), P_2 represents all possible states of the second philosopher process (right in the picture), P_3 represents all possible states of the monitor process (middle in the picture), and the last one, P_4, is a mixing of the value of the *Avail* variable and of the state of the monitor.

Each transition of the model is included in at least a T-semiflow, and this is a necessary, but not sufficient, condition for the net to be live. Table 2 shows the T-semiflows. T_1 (T_2) represents the first (second) philosopher possible actions, and, whenever it is possible to fire a corresponding firing sequence, the system comes back in the initial marking.

P_1	inre11 + inre12 + inr11 + inr12 + rf11 + rf12 + endWhile1 + rel12 + rel11 + Eat1 + Phil1 = 1
P_2	inre21 + inre22 + inr21 + inr22 + rf21 + rf22 + endWhile2 + rel21 + rel22 + Eat2 + Phil2 = 1
P_3	inc + dec + inre11 + inre12 + inre21 + inre22 + inr11 + inr12 + inr21 + inr22 + ALT = 1
P_4	2*inc + 3*inre11 + 2*inre12 + 2*inr22 + inr11 + 3*inre21 + inr21 + 2*inre22 + rf12 + rel12 + 2*rel11 + 2*Eat1 + 2*inr12 + rf22 + 2*Eat2 + rel22 + 2*rel21 + Avail + ALT = 3

Table 1. P-invariants from the 4 P-semiflows of the GSPN model of Fig. 2

T_1	2*inc 2*dec einre21 einr22 einr21 einre22 Eat2 Think2 inre21 inre22 inr21 inr22 endWhile2
T_2	2*inc 2*dec einre11 einre12 einr11 einr12 Eat1 Think1 inre11 inre12 inr11 inr12 endWhile1

Table 2. T-semiflows of the GSPN model of Fig. 2.

Indeed the system is not live, since of the 55 reachable markings, one is a deadlock: the state in which there is a token in places *rf22*, *rf12*, and *ALT*.

The models obtained by removing a philosopher (initial marking of place *Phil1* = 0 or *Phil2* = 0) have no deadlocks, which suggests that, as expected, the deadlock comes from an interleaving of actions of the two philosopher processes that want to access the pool of common resources (the forks). By checking the two philosophers alone, we ensure that the model of each process has no deadlock (a deadlock in a single process model is generally caused by some missing arc in the net).

A visual interactive simulation of the net immediately reveals that a firing sequence that takes the net into a deadlock state is, for example, *Think1*, *inr11*, *einr11*, *Think2*, *dec*, *inr21*, *einr21*, and *dec*. The problem is that the two forks can be granted to two different philosophers, that are not going to release them unless they get a second one, which obviously causes a deadlock.

One way to eliminate the deadlock is to oblige the philosophers to get both forks at the same time. The modified code is shown in Fig. 3, and the corresponding net in Fig. 4. A structural analysis of the model individuates again 4 P-semiflows, and 2 T-semiflows, that have the same interpretation as before. The reachability analysis reveals no deadlocks, and produces 25 states. We can check on the set of reachable states that it does not exist a state with both places *Eat1* and *Eat2* marked: this ensures that the two eat activities are in mutual exclusion. We can also observe that there exists no reachable state in which one of the eat place is marked together with place *Avail*. In the general case of $N > 2$ the property that we need to check is that at most $numphil/2$ philosophers can be eating at the same time.

The qualitative analysis also reveals that the reachability graph is strongly connected, a sufficient condition for the steady state probabilities to be well-defined: the classical qualitative analysis can be therefore integrated by a quantitative analysis that allows to compute performance indices, but also to derive some assertion on the state space properties in a probabilistic form. Quantitative analysis is built around two basics performance indices: *throughput* of transitions (the mean number of firing of a transition t per unit time) and *distribution* of tokens over the places (probability of having a number n of tokens in place p).

Symmetric behaviour We begin our analysis by assuming a timing behaviour fully symmetric: each timed transition in the system is assigned a weight [1] of 1 which implies that all timed activities (computation and communication) have the same distribution, which implies, obviously, the same mean delay value. The solution in steady state with the above timing assignment reveals a throughput of 0.110938 for *all* transitions representing activities of the philosophers, and twice as much for transitions *inc* and *dec*. A symmetric assignment of weights

[1] According to GSPN definition, the weight of a transition t is the rate of the exponential distribution of the random variable "delay of transition t", and the mean value of the delay associated with the transition is therefore the inverse of the weight (the mean value of an exponential distribution is the inverse of its rate).

```
Philos( chan req_chan, chan rel_chan)
while(true)
    THINK
    req_chan!dummy
    EAT
    rel_chan!dummy

Forks_Monitor( chan grant_fork, chan acc_rel)
int Avail=numphil

while(true)
alt
    (Avail>1): grant_fork?dummy
       Avail=Avail-2;
    acc_rel?dummy
       Avail=Avail+2;

root
chan grant_fork,acc_rel
par Forks_Monitor(grant_fork,acc_rel)
    for i=1 to numphil Philos(grant_fork,acc_rel)
```

Fig. 3. The code of the philosophers acquiring two forks at a time

implies indeed a symmetric behaviour of the philosophers which, in the mean, take 9.01404 (computed as 1/0.110938) units time to complete a sequence of think and eat. If we want to know how much each philosopher is delayed by the presence of the other, we can solve the same system with an initial marking of zero token in place *Phil2*, which yields a throughput of zero for all transitions related to the second philosopher, and a throughput of 0.2 for the transitions related to the first one: in absence of contention a philosopher completes his cycle in a mean time of 5 time units (which may look a little surprising considering that each cycle requires 6 timed activities to complete, namely a think, an eat, two exchanges of message, a decrement and an increment of the shared variable, but we should consider that the decrement can go in parallel with the eating, and that the increment goes in parallel with the thinking) Therefore the delay experienced by a philosopher due to the presence of the other one is of about 4 time units.

Another interesting question is about the behaviour of forks: how long philosophers have to wait for a fork, and does the behaviour (qualitative and/or quantitative) of the system change by increasing the number of forks?

If we indicate with $P\{x\}$ the steady state probability of condition x, and with $\mathbf{m}(p)$ the marking of place p, then the probability that philosopher 1 has to wait for acquiring the forks can be expressed as the probability of the philosopher

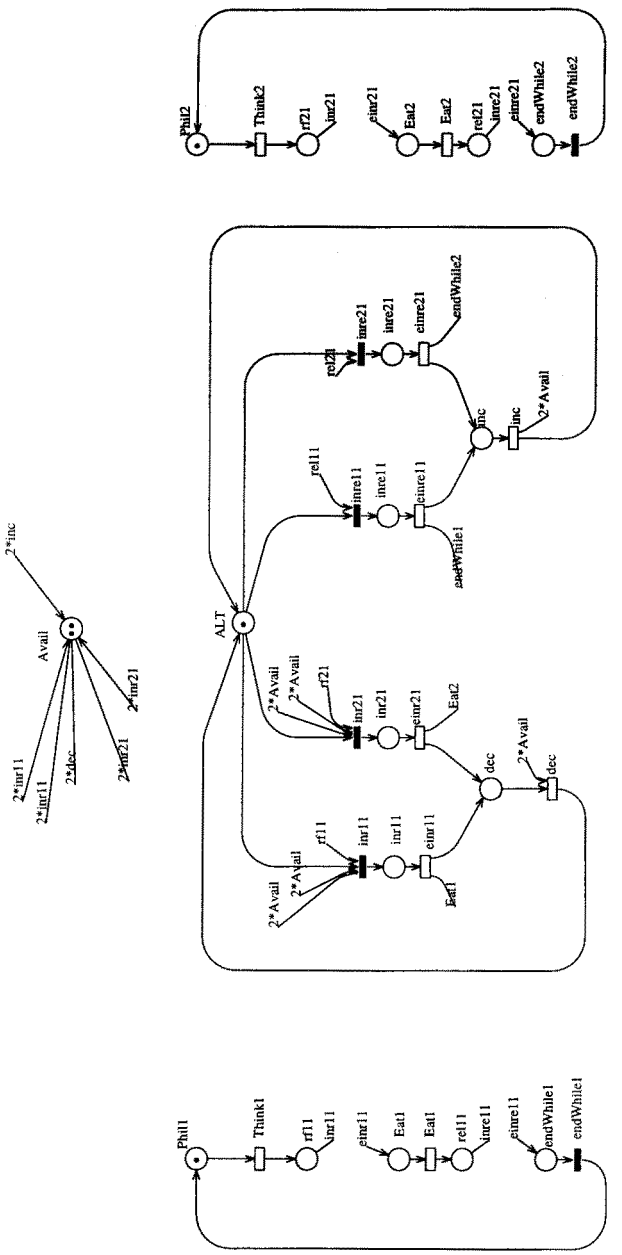


Fig. 4. A GSPN model of two philosophers acquiring two forks at a time

being in place *rf11*, that is to say $P\{\mathbf{m}(rf11) = 1\}$. The value computed for it is 0.50078, that is to say, more than half of the time of the philosopher is spent waiting for the communication over the grant_fork channel. It is interesting to observe that even if there are only two forks, there is a non null probability that philosopher 1 has to wait before being able to give the forks back on the acc_rel channel ($P\{\mathbf{m}(rel11) = 1\} = 0.05547$), and this is because the forks monitor is executing a decrease variable activity right after granting the forks, and may not be ready to receive the forks back when the philosopher has finished his eating.

We know that with two forks the eating activities are mutually exclusive since $P\{\mathbf{m}(Eat1) = 1 \text{ and } \mathbf{m}(Eat2) = 1\} = 0.0$. Moreover the probability of having 0, 1, or 2 forks available is 0.55469, 0.0, and 0.44531, respectively; the value of 0.0 for the probability of having a single fork available is not surprising given that the forks are always acquired and released by pairs. When we increase the number of forks by one ($\mathbf{m}(Avail) = 3$), the throughput of the system is maintained, and the probability of having $0, 1, 2$, or 3 forks available is $0.0, 0.55469, 0.0$, and 0.44531 respectively (the only change is that there is a fork always sitting in place *Avail*). When we increase the number to four, we should expect the two philosopher to be pretty independent, since the shared resource is non blocking any longer. The throughput obtained is now 0.122222, for a cycle time of 8.1818331, which is far from the 5 unit time of the philosopher running alone: by increasing the number of forks we have simply moved the contention from the forks onto the shared resource "Forks_Monitor process".

As a final example that makes our assumptions a little bit more realistic, let us assume that the increment and decrement activity are quite quick, and that the communication takes less than the eat or think activity: we can, for example, assign a weight of 100 to decrement and increment of the variable, and of 50 to the communication, in this case for two forks we get a cycle time of 2.5687923 and of 2.0415913 for four forks, while we have a value of 2.0401959 for the single philosopher case. Indeed if communication and variable modification activities are much faster, the delay due to the presence of the other philosopher is reduced, and the presence of four forks makes the two processes almost independent.

Asymmetric case We consider two asymmetric loads. In the first case we assume that the first philosopher thinks 10 times slower than philosopher2, which implies to decrease its rate from 1.0 to 0.1. The consequence on the throughput and on the cycle time are quite sensible: the throughput of philosopher 1 has dropped from 0.110933 to 0.05677 (with a cycle time increase from 9.0144502 to 17.614937, that can also be compared with the cycle time of philosopher 1 when there is no other philosopher, which is 11.040087), while that of philosopher 2 has increased from 0.110933 to 0.155849 (cycle time decrease from 9.0144502 to 6.4164672). The increase in the cycle time is clearly due to the increase in think time, while the decrease in the cycle time of philosopher 2 is due to the lower contention for the forks. The probability of a philosopher waiting for the forks is viceversa similar for the two philosophers: it is 0.23360 for philosopher 1 and 0.29868 for philosopher 2: both philosophers spent some 20 to 30 % of their time waiting for forks.

If instead philosopher 1 keeps the forks for a time that is 10 times longer than philosopher 2 (all transitions rates equal to 1.0, but for transition *Eat1*, that is set equal to 0.1), then the two cycle times are similar, they both increase to a value around 17.5 (17.651004 for the first one and 17.451398 for the second one), but the waiting times are rather different (0.25834 for the first philosopher and 0.74214 for the second).

The analysis reported here has the aim of studying how the processes influence each other, and how the execution times changes according to different hypothesis on the temporal behaviour of the processes: it is indeed very specific to the program that we are considering. In general the choice of the performance indices to be computed is strictly related to the goal of the evaluation of the system, but there are also general indications that we may want to extract from a quantitative analysis. One need frequently arising in the literature is that of providing a quantitative characterization of the program, that can be used by load balancing or mapping algorithms. A data structure that is often used by these algorithms is the communication graph, a graph with one node per process: there is an arc between node p_i and p_j if process p_i exchanges messages with p_j, or viceversa. The weight of the arc should provide an estimate of the amount of communication exchanged between the two processes. A possible choice for the weight is to use the sum of the throughput of all transitions that represent a communication between p_i and p_j. Fig. 5 shows the communication graph for the program of Fig. 3, computed using the throughput of the transitions of the GSPN model of Fig. 4 that represent a communication statement. Communication graphs can be enriched by labelling each node with an indication of the execution time of the corresponding process. Again these values have been computed using the GSPN model in Fig. 4, as the sum of the throughput of all transitions that represent activities that we have classified as "computation:" this transitions are *Think1* and *Eat1* for the first philosopher, *Think2* and *Eat2* for the first philosopher, and *dec* and *inc* for the monitor. An algorithm for allocating the program on two connected processors, according to the indications provided by the communication graph will presumably choose to allocate on the same node Philos_2 and Fork_monitor, since they are the two lightest processes and, moreover, there is more exchange of communication between Philos_2 and Fork_monitor than between Philos_1 and Fork_monitor (and in general communication within the same node is less expensive than communication between different nodes).

An additional data that can be computed from the GSPN model and that may be used in the allocation procedure, is the level of interference between processes: the probability that two processes are executing at the same time. Indeed two processes whose computing activities are mutually exclusive, are likely to be candidate to share the same processing node. For example we can compute the interference between the two philosophers, as the sum of the probabilities of all those marking in which both philosophers are performing a computing (eat or think) activity. This probability is 0.21162 for the two philosophers, 0.27982 for Philo_1 and Forks_Monitor, and 0.16641 for Philo_2 and Forks_Monitor. Indeed

also the interference shows that the choice of placing on the same node Philo_2 and Forks_Monitor is the more advantageous.

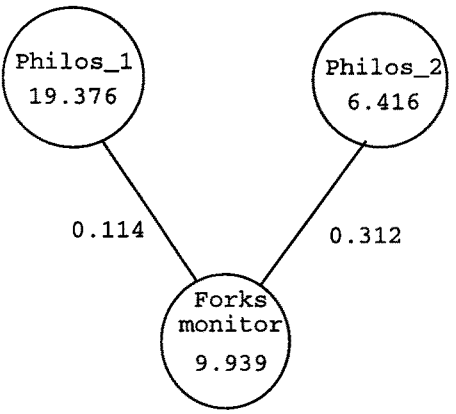


Fig. 5. The communication graph for the program of Fig. 3.

3 GSPN modelling of distributed programs

The example of the previous section is indeed a successful case: it was rather straightforward to build the model and solve it, and it was easy to translate back the results of the analysis in terms of program behaviour (for example it was very easy to interpret the net deadlock state as a program state). But what are the characteristics of the philosopher program that makes it suitable to analysis, and what are the more or less implicit hypothesis that we have based our analysis upon?

We shall start by first considering the peculiarity of the program, and their impact on the analysis, together with suggestions on how to deal with more general cases.

3.1 Learning from the example

The philosopher program was given in a very abstract form: the code only contains statements that describe either process activations or communications, and other activities in the philosophers are summarized by the macro activities THINK and EAT. There is in the example a one to one relationship between statements of the program and elements of the net, where all places, with the exception of *Avail*, basically represents the program counter of each process.

When modelling real programs, we should expect instead a larger proportion of "normal" statements, and a large number of variables that can be used to store results of computation, how to decide what to model explicitly, and what

to neglect? The choice of the abstraction level should be driven by the goal of the analysis. Main goal of the qualitative analysis is to check the presence of deadlock, and therefore all "concurrency" statements (par, alt, ?, !) should be part of the model. Of course also the path used to reach a certain concurrency statement should be represented. The quantitative analysis "only" demand is instead to model activities for which we are able to provide a reasonable estimate of their duration. As we shall later see this is not a trivial task.

The state space for the example is rather small: this is not always the case, although the abstraction level chosen binds the complexity of the model to the concurrent structure of the program, and not to the sequential part, the resulting model may still be too big to be solved. The only possible qualitative analysis is the one based on P- and T-semiflows, and on net structures such as deadlocks and traps. Quantitative analysis instead can be performed using discrete event simulation to compute stochastic estimates of the envisioned performance indices (this technique does not require to store the set of reachable states of the model, and, although expensive, can be used for much larger nets than the one solvable through construction and solution of the associated CTMCs).

There is a single variable in the philosopher program, that can take a finite, and low, number of values. Indeed variable *Avail* can assume only two values, as reported by the quantitative analysis, and this is not always the case. Explicit representation of variables is undoubtly one of the major sources of complexity, especially when there is a low degree of dependency among the variables values, so that the state space of the possible joint values of all variables becomes close to the Cartesian product of the possible values for each variable. The selection of the variables that should be modelled explicitly is therefore a critical one.

A single deadlock state was found for the example models, and it was straightforward to show that it corresponds to a deadlock state of the program. Given a deadlock state of the net, we need to determine whether it is a non-fault or whether the real problem can actually block at run time. It is therefore very important to be able to have an easy way to associate program states to net states: this is straightforward, since statements of communication, that are the ones that can cause deadlock, are explicitly represented, and from the places that represent processes' program counters it is easy to determine the program state. Once the net deadlock has been translated into a program state it may not be obvious, and it is left to the modeller, to decide whether that program state is a reachable one. For example in the rather simple spooler program presented in [1], chap.10, there is no deadlock, but the corresponding PN model has 5 deadlock states: all of them are non faults. The model of a rather simple 2D-FFT program, shown in [9], has 1021 states and 123 deadlocks, all of which are non-faults.

A classical situation in which a non-fault arises is when two processes communicate between them from inside loops, and each loop is executed the same number of times by the two processes: if the variables that control the loop are not explicitly represented, then the choice between re-executing and exiting the loop is taken in a probabilistic manner, and if a process decides to exit the

loop, thus not performing any additional communication, while instead its peer process takes the opposite decision, then the model reaches a deadlock state.

Non faults are typical of static analysis, since this type of analysis does not consider variables. Since static analysis using Petri nets actually amounts to executing the static model, the run time behaviour of variables whose values do not depend from run-time supplied input values can be taken into account by the analysis.

The example program has no dependencies from run-time supplied input values. How to model a program whose structure of synchronization can depend on variables whose values are only known at run time? The obvious answer is to add to the program model an abstract representation of the environment, providing the input values at run time. A particular class of problems for which this is rather straightforward is that of process control systems in which the relevant information is not the value of the input signals, but their frequency. Control signals are easily represented by timed transitions, and the interested reader can find in [4] an application of this idea to a monitoring system.

The example program has a cyclic behaviour: the three processes created by root execute a while(true) statement, but this is certainly not always the case. Indeed some deadlock states of the program actually represent program termination states, so that they are perfectly acceptable. The only problem is that no steady state solution can be computed for models with deadlock states, nevertheless it could make sense to speak of classical steady state performance results as "relative frequency of communication over channel c". The solution in this case is to "restart" the system from each terminal state, by adding to the net a subnet that changes the state from the deadlock state to the initial one. Obviously this is possible only when there is a limited, and well defined, number of termination states: we shall see in the next section one such case.

3.2 Hypothesis behind the analysis

By taking GSPN as a modelling formalism, we are implicitly assuming that duration of (blocks of) statements can be modelled by exponentially distributed variables, and that we are able to characterize the distribution parameters, moreover we have assumed that it is reasonable to compute performance of distributed programs without taking into consideration the contention for physical resources. We shall now discuss each of these issues separately.

No contention on physical resources: the model of the philosopher program does not represents explicitly the contention for processors and communication media: indeed, whatever is the physical environment in which the program is going to be executed, we are always going to get the same values of the performance indices, since no hardware description is included in the model. The analysis performed is therefore valid under the (unrealistic) assumption of infinite resources. This choice, which may appear as a disadvantage, has the positive aspect of providing results that are valid in whatever context, in particular it provides a lower bound on the execution time: it therefore allows a study of the efficiency of the program

per se, independent from the number of processors, the mapping, and the load balancing algorithms.

This analysis approach is intended to be performed from the early steps of the implementation, when the full program is not yet available and the target architecture may not have been dimensioned. Being machine independent, the analysis provides very general results that may be checked at later stages of the software life cycle by building machine dependent models [7] or by monitoring the final implementation on the target architecture.

Exponential distribution of delays In GSPN models the delays associated to transitions are exponentially distributed; this choice produces two major advantages: the resulting stochastic model is a Markov chain that can be solved numerically, and the behaviour of the timed model with respect to qualitative properties is the same as the untimed one, so that all the large set of results for untimed Petri nets can still be applied. But is the exponential assumption a realistic one? We can think in general that a statement requires a fixed amount of time for its execution, and the same holds for communications. Nevertheless the randomness introduced by the exponential assumption can account for some variable behaviour of the system: indeed each transition in a GSPN model of a program may represent a variable number of statements, including loop statements; moreover, since the model does not represent the hardware, a random delay associated to transitions may account for the variable execution time of a piece of code or of a communication due to contention for the cpu's or the channels.

Computation of the parameter of the delay distributions To fully specify an exponential distribution it is necessary to give a parameter (the rate) which is the inverse of the mean value: we therefore need to estimate the mean delay of the activity represented by each transition. This may be a formidable task, especially when transitions represent macro activities, or when the duration of the activity depends from input data.

Indeed, when the model is used in the early stages of the design, it may make more sense to perform a study in relative terms than in absolute ones: in the philosopher example we have considered each delay in terms of the communication delay, set to one as a reference value, and goal of the analysis was more to provide indication on the sensitivity of the performance indices with respect to variations in the delays of basic activities, than not to provide absolute values for the performance indices.

3.3 Modelling variables

The example can be used to reason on the importance of a correct choice of the variables to be modelled: if *Avail* is not modelled the GSPN does not have any deadlock state, although the program does have a deadlock state. This problem is obviously caused by the fact that, if *Avail* is not included in the model, we are abstracting with respect to the number of available forks, which implies that

no philosopher is ever blocked. It should be remarked that a live model of a deadlocked program violates the main requirement of static analysis (a deadlock in the program implies at least one deadlock in the model): we shall see in the next section a more complicated translation of the alt that maintains this requirement.

Viceversa a variable that controls branching may or may not be modelled, and there could be no consequences or disastrous ones.

As a final remark, observe that if in the program there is a variable that counts how many times the philosopher eats and we include it in the program, the model becomes unbounded, or at least difficult to solve if we assume a mod(N) increment operation.

4 Automatic modelling and analysis of distributed programs

The simplicity of the example of the previous section may have given the (wrong) impression that building models of distributed programs is a simple matter. If a program is complex, in particular if its communication structure is complex, the construction of the model is an error prone activity, moreover there is a high risk to model what we think the program does, more than what it actually does. The answer to these problems lays, of course, in automatic translation.

In this section we present the automatic code translation algorithm, an example of application of the automatic translation to a time warp distributed simulation program, and examples of relevant performance indices whose definition can be automatically generated when the program model is constructed. The translation has been implemented in a tool called EPOCA [8], the tool used for the analysis of the GSPN models is GreatSPN[5].

4.1 Automatic translation of CSP-like concurrent programs

In Section 2 it has been shown on a simple example how a program can be modelled using the GSPN formalism. In this section we shall describe an algorithm for automatic translation of programs written in a CSP-like language. The reference language that we want to automatically translate comprises a set of statements that we classify as *sequential* (meaning that these statements are not specific of a programming language for writing concurrent programs) and a set of *concurrent* statements. The sequential statements that we shall consider for translation are: assignment, if-then-else, and while, that have the usual semantics, and seq, that allows to compose n statements sequentially. The concurrent statements that we shall use are: par (activation of n concurrent processes), alt (non deterministic choice among n statements conditioned on the truth value of some logical condition and/or on the availability of a rendez-vous on a given channel), and the synchronous input/output communication statements whose syntax is chan-in?message and chan-out!message respectively.

This language allows only *static process creation*, i.e. the set of processes that compose the concurrent program is known at compile time. All processes are activated through some par statement and there is a unique *root* process which originates all other processes composing the concurrent program. Processes communicate through declared common *channels*.

A *process declaration* consisting of a name, an interface and a body, defines a process type: the process name is used in the par statements to define the type of processes being activated (of course many instances of the same process type can be activated in the same par as well as in different par statements). The interface of a given process type is defined as a set of parameters representing unidirectional channels: the connections through actual channels among the activated process instances are defined in the par statement that activates the processes. Finally the body of a process type declaration defines the behaviour of all processes of that type.

All communications are of the rendez-vous style, i.e. both partners of a communication need to be ready to exchange the data (one ready for an input and one for an output over a common channel) before the communication can actually take place.

The alt statement allows a process to wait for a message arriving from other processes on a given subset of channels, upon reception of a message an action is performed that depends on the reception channel. It is also possible to add a logical condition to consider only a subset of input channels at each particular execution of the alt statement, moreover special mechanisms are also available to set a timeout so that it is possible to avoid indefinite blocking on the alt statement. If messages are ready to be received on more than one channel at the same time, one of them is chosen non deterministically.

The abstraction level that we have to use when constructing a model of a program depends on several factors, namely on the type of properties of the program one wants to study, on the available information on the possible ranges for input data, on the (computational) cost one is willing to pay to obtain the results and to interpret them (this in turn depends also on the features of the available analysis tools).

On one hand, in order to get very precise results out of the model one may want to include as much detail as possible, so that the model behaviour exactly reflects the actual program behaviour, but this choice leads to huge and intractable models. On the other hand keeping the abstraction level as high as possible, according to the desired results, has the advantage of decreasing the computational cost needed to get the results, of allowing to study the logic behaviour of the program at a higher abstraction level than that of the code (hiding lengthy sequences of statements that might be not very significant from the point of view of the processes interaction, for example) hence making it easier to give an high-level interpretation of the results. The drawback of less detailed models is of course that of providing less accurate results, and in particular of opening the possibility of detecting behaviours on the model that are not possible in the real program (see for example the discussion of non-faults in Section 1).

The abstraction level chosen for the automatic translation algorithm is oriented to the verification of a correct and efficient interaction among the processes composing the program. We have thus chosen to explicitly represent all the process activation and interaction statements (i.e., all **par**, **alt**, **?**, and **!**) and all those control statements (**while**, **if**, **seq**) that contain a process activation/interaction statement at any level in their body. All the portions of sequential statements between two process activation/interaction statements are instead abstracted out as single activities (macro-statement), whose precise behaviour is not modeled.

Concerning data representation, we first consider the choice of not including any data representation in the model, then we show that it is possible to identify a subset of integer or boolean variables in a program that are used for control purposes (e.g. counters in loops) that can be easily (and automatically!) modeled, without causing an unacceptable growth in the model state space and that often can eliminate the problem of non-faults.

Let us describe the steps needed to translate a concurrent program:

1. Produce a *process schema* from each process declaration in the program by coalescing into single macro-statements all those sequences of statements that do not include any process activation/interaction statement.
2. Translate each process schema into a GSPN model using the translation rules depicted in Fig. 6, and representing the activation of a process in a **par** by a single transition (i.e., do not substitute a process activation with the translation of the corresponding process schema), the communication statements by immediate transitions (thus disregarding the explicit representation of the communication partner, as well as the synchronization aspect of the rendez-vous), and the macro-statements by single timed transitions (whose associated delay is an estimate of their execution time).
3. Starting from the **root** process schema model, substitute each transition representing the activation of a process $\mathbf{proc}_i$ with a copy of the GSPN model of the corresponding process schema. Since each $\mathbf{proc}_i$ can in turn activate other processes, the substitution continues in depth-first mode until all the transitions that represent the activation of a process have been replaced.
4. Pairs of communication transitions that belong to different processes and that represent their (mutual) communication are *fused* to concretely represent the synchronization deriving from the rendez-vous protocol, and are then *expanded* into an "immediate transition-place-timed transition" sequence to express the time needed to complete the transfer of a message from the sender process to the receiver.

The above four steps allow to produce a PN model (with priorities over transitions). In order to get a GSPN model, a further step is needed to define the weights/rates of transitions. To assign the rate of timed transitions we can define *rate parameters* corresponding to the basic activities (like the time required to execute a single statement, e.g. an assignment, or to perform a communication of a single byte), and then use the rate parameters (or an expression of them) to define the rate of each timed transition. The problem becomes thus to automatically associate to each transition an expression representing the complexity of the

code modelled by it, as a function of some basic rate parameters. Following this approach, all transition rates are expressed in terms of a small set of parameters, so that it is rather easy to study the sensitivity of the quantitative behaviour of an application to changes in the duration of the basic program activities. The weights of immediate transitions allow to probabilistically characterize the branching points in the program control flow (if, while, alt statements): unless the programmer can give precise information on the probability of following each branch, the same probability is assigned to all branches.

Let us discuss each translation step in details. The first step is conceptually very simple and it requires the implementation of a simple preprocessor to produce the process schemas.

The second step requires the implementation of a translator based on the rules of Fig. 6 where τ denotes the translation function, A, B and A_i stand for any statement, while proc_i stands for a process name. Each statement is translated into a GSPN in which it can be identified one *entry place* and one *exit place*; the entry/exit places have an empty input/output set respectively. As already said above, any *macro statement* is modelled by a very simple model consisting of a single timed transition with one input place (the *entry place*) and one output place (the *exit place*), and any communication statement is represented by a similar net with an immediate transition instead of a timed one. In this step it is necessary to store the information on which transition represents a process activation and which represents an input/output statement on a given channel: this information will be used in later translation steps.

Let us discuss each translation rules of Fig. 6: the if construct (Fig. 6.a) is translated as a free choice (out of the entry place) between two immediate transitions, one associated with the *then* branch, and one associated with the *else* branch. The output place of transition t_{then} is the entry place of the GSPN model resulting from the translation of statement A. The exit place of model $\tau(A)$ is then connected to an immediate transition (representing the end of execution of statement A) whose only output place is the exit place of the if model. The translation of the *else* branch follows the same rule. In case the else branch does not contain any statement, the else branch subnet is substituted with a single immediate transition t_{else} whose input place is the if entry place and whose output place is the if exit place.

The model of the while statement (Fig. 6.b) has an initial free choice between the immediate transitions t_{exit} and t_{loop}. The output place of t_{exit} is the exit place of the while model, hence transition t_{exit} represents the exit from the while loop when the condition *cond* is false. The output place of t_{loop} is the entry place of the GSPN model resulting from the translation of statement A, the exit place of model $\tau(A)$ is then connected to immediate transition t_{cycle} representing a jump to the condition test at the beginning of the while loop (in fact its only output place is the entry place of the while model). Observe that if the condition is simply *true*, then transition t_{exit} should be eliminated from the while model (hence leaving an isolated exit place).

The representation of the seq (Fig. 6.c) is simply obtained by superposing

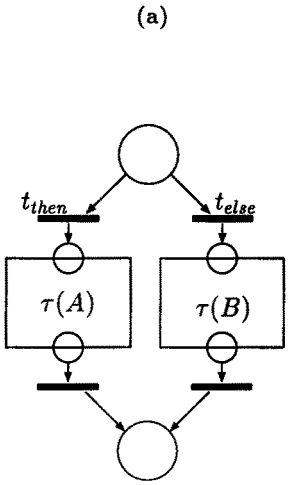
the exit place of the GSPN submodel $\tau(A_i)$ and the entry place of the GSPN submodel $\tau(A_{i+1})$, for all $i = 1, \ldots, n-1$, hence modeling the sequential control flow. The entry and exit places of the **seq** model coincide with the entry place of the $\tau(A_1)$ submodel and the exit place of the statement $\tau(A_n)$ submodel respectively.

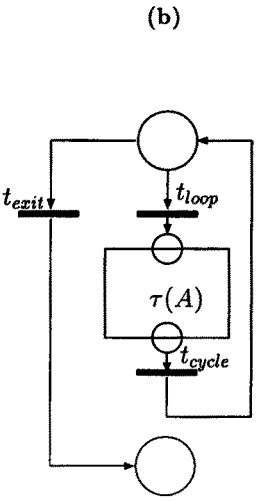
The model of a par statement (Fig. 6.d) consists of a transition t_{par} with the model entry place as unique input, and with as many output places as the number of processes to be activated, each output place b_i of transition t_{par} represents the activation of process of type $proc_i$, and has a single output transition t_{proc_i} which is a very (!) abstract view of the behaviour of process $proc_i$ (that will be refined in the third translation step). Each transition t_{proc_i} has a single output place e_i representing the termination of the process. Transition t_{sync} has a single output place, which is the exit place of the par model, and n input places, $e_1, \ldots, e_n$: this transition models the fact that the par statement terminates only when all the activated processes have terminated.

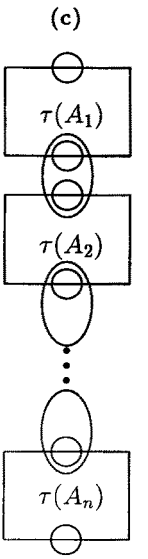
The model of the alt statement (Fig. 6.e) can be seen as the dual of the par model, in fact only one branch of the alt model is "executed" while all branches of the par are "executed". The upper part of the model represents a free (probabilistic) choice among the n guards of the alt represented by the n immediate transitions $G_1, \ldots, G_n$. The output place of transition G_i is the entry place of submodel $\tau(A_i)$. The exit place of submodel $\tau(A_i)$ is then connected to an immediate transition whose output set contains only the alt model exit place[2].

The proposed model of the alt statement is correct only if the guards are pure communication guards, and have no boolean condition associated. Should guard G_i comprise a boolean condition $cond_{G_i}$, then we should consider the two possibile truth values for this condition, choose arbitrarily one of the two, and then perform the free choice in the alt model only considering those guards for which $cond_{G_i}$ is assumed to be true. Only this way we can consider all possible paths in the program execution, as we have already observed in the philosophers example of Section 2. In Fig. 7 a translation of an alt statement is proposed, assuming that the subset $G_1, \ldots, G_k$ of guards are pure communications, while the subset $G_{k+1}, \ldots, G_n$ of guards include a boolean condition (the new immediate transitions $true\_cond_i$, $false\_cond_i$ in the upper part of the model have higher priority than transitions G_i). Intuitively, the first set of free choices between pairs of immediate transitions $true\_cond_j$, $false\_cond_j$ allows to randomly decide if the boolean condition associated with guard G_j is true or false. Then one of the enabled guards is chosen randomly, and places representing the truth

[2] Observe that the immediate transitions in the alt exit place input set are redundant: they could be eliminated by simply merging all the exit places of the $\tau(A_i)$ submodels, and considering the resulting place as the alt exit place. This can be done because by construction the $\tau(A_i)$ exit places have no output transitions. Similarly, the last two immediate transitions of the if model and the t_{cycle} transition of the while model could be eliminated and their input place merged with the output place: this optimization of the produced model can be easily implemented, however this is a trade-off between model size and readability.







Conditional statement

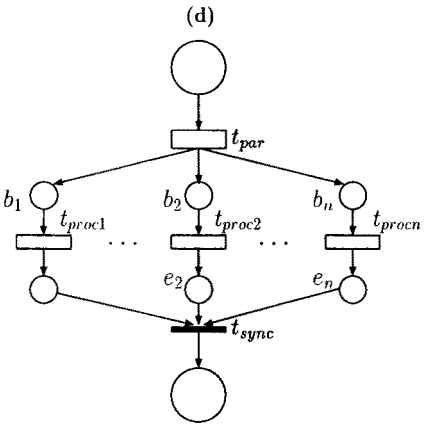
if (*cond*)
    A
else
    B

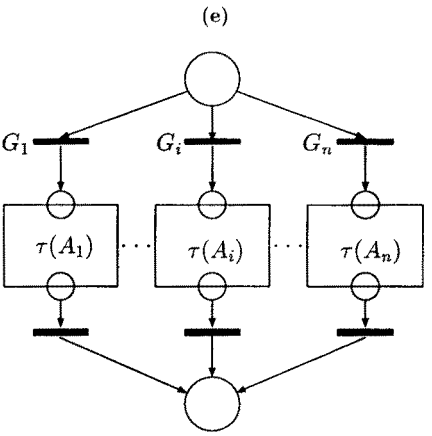
While statement

while (*cond*)
    A

Sequential composition

seq
    A_1
    .
    .
    A_n





PAR statement

par
    $proc_1(channels)$
    ...
    $proc_n(channels)$

ALT statement

alt
    G_1 : A_1
    ...
    G_n : A_n

Fig. 6. Translation schemes for the sequential (a,b,c) and concurrent (d,e) language statements

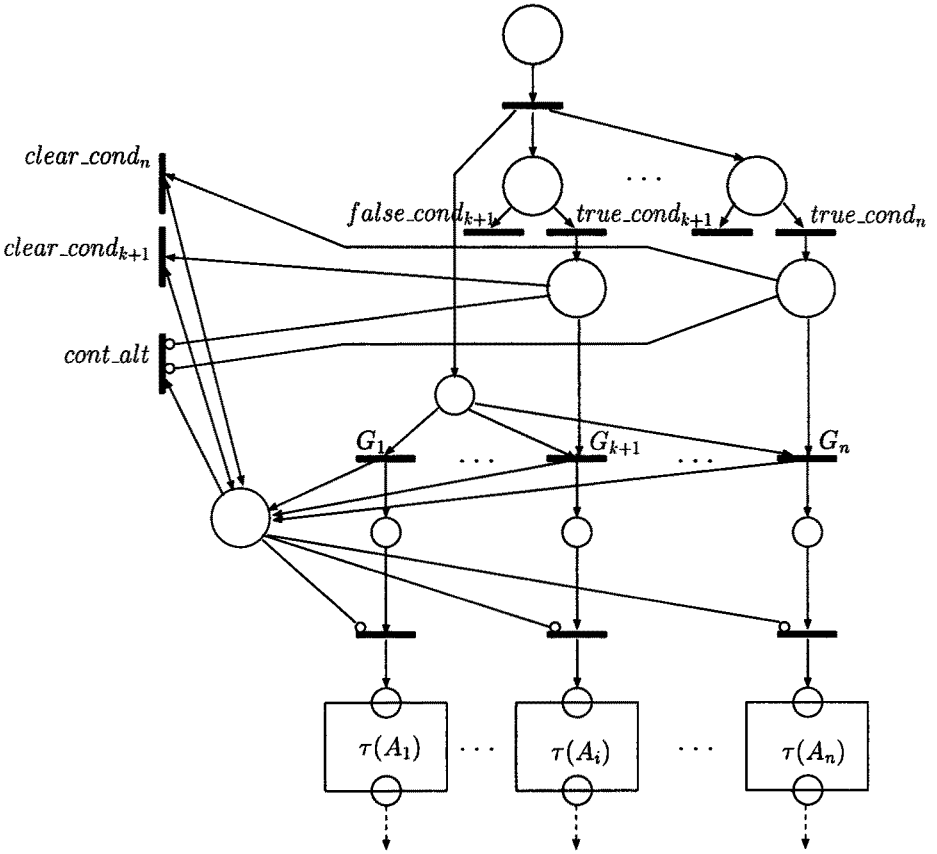


Fig. 7. Translation scheme for the alt statement including boolean guards

value for the conditions are cleared (transitions $clear\_cond_j$) before proceeding to the execution of the statement associated with the chosen guard.

We shall discuss again the problem of modeling guarded commands including boolean conditions in the guards in a later section on the modelling of the program control variables.

Once each process schema has been translated in isolation, we can proceed to the third step: starting from the root process model, substitute in all par statement submodels the timed transitions representing the activation of a process instance with the corresponding process schema model. The substitution proceeds recursively since the inserted process schema models may in turn contain other par statement submodels. For example the translation of the distributed simulator program of Section 5 makes six substitutions inside root (Fig. 11), two for Sim, two for Rec and two for Tran, and each of the two activations of Rec gives rise to two substitutions, one due to the activation of Buffer and another one due to the activation of Sender (Fig. 14).

In this step, care should be taken to keep track of the association of actual parameters (channels) with formal parameters upon process activation, with the aim of generating a table containing for each actual channel $Chan_i$ the list of immediate transitions in the whole model representing input/output statements on $Chan_i$: this information is essential for the implementation of the fourth step.

The model obtained in the third step contains all the information on the process activation structure of the program, however it still doesn't model the synchronization among processes due to communication. The fourth step transforms the model to include this information. The set of immediate transitions representing communication statements in the model built by the third step, can be partitioned into as many pairs of subsets $InChan_i$, $OutChan_i$ as the number of distinct actual channels $Chan_i$ in the whole program. $InChan_i$ ($OutChan_i$) is the set of all immediate transitions representing an input (output) statement on channel $Chan_i$. Since potentially any input transition may synchronize with any output transition on the same channel, we need to add into the model a synchronization transition t_{io} for each pair $(t_{in}, t_{out}) \in InChan_i \times OutChan_i$: the input (output) set of the new transition will be given by the union of the input (output) sets of transitions t_{in} and t_{out}. After adding the synchronization transitions, all the transitions in $InChan_i$ and $OutChan_i$ must be removed from the model. To express the fact that the communication activity takes time, the newly created communication transitions must undergo a last transformation, namely each transition t_{io} is expanded into a sequence t_{sync}, p_{comm}, t_{comm} where t_{sync} is an immediate transition whose input set is the input set of t_{io} and whose only output place is p_{comm}, while t_{comm} is a timed transition (whose associated delay represents the duration of the communication activity) whose only input place is p_{comm} and whose output set is the output set of t_{io}. An example of application of the fourth step is shown in Fig. 8: in the left part of this figure, a portion of code is shown, comprising two processes ($Proc_j$ and $Proc_i$) interacting through channel $chan1$. In the middle part of the figure, is shown the corresponding model portion resulting from the third translation step. In this model $InChan_1 = \{chan1?x, chan1?y\}$ and $OutChan_1 = \{chan1!msg\}$. The right portion of Fig. 8 shows the result of application of step 4: transitions $Sync\_chan1x$ and $Comm\_Chan1x$ are the result of composition and expansion of the pair of transitions $chan1?x$ and $chan1!msg$, similarly transitions $Sync\_chan1y$ and $Comm\_Chan1y$ are the result of composition and expansion of the pair of transitions $chan1?y$ and $chan1!msg$.

4.2 Translation of control variables

We have discussed in the previous sections some of the problems that can arise if variables are ignored when constructing the GSPN model of a concurrent program. In this section we show how boolean control variables can be included in the program model to reduce the possibility of nonfaults detection; the variable translation rule that we propose is automatizable and can be easily extended to work also with unsigned integer variables. Although the introduction of this refinement in the model may potentially lead to much larger state spaces, it is

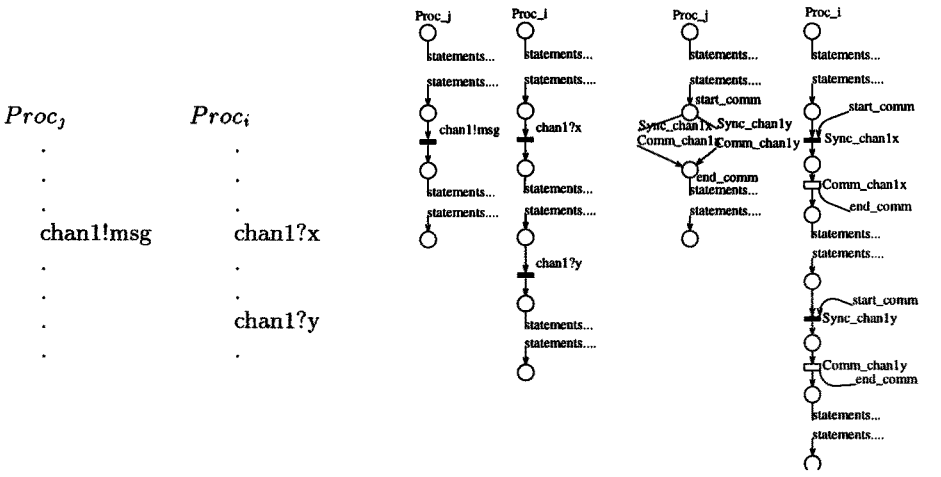


Fig. 8. An example of superposition and expansion of communication transitions.

often the case that the increase in state space size is not that dramatic, since the introduction of variables can reduce the set of possible behaviors modelled by the net, and, moreover, certain variables may depend completely on the state, so that no additional states are generated. In the 2D-FFT example appeared in [9], the number of states of the model without variables is 1021 (including 123 deadlock states which are non-faults), while the number of states drop to 169 for the model that explicitly represents the variables that control cycles. Two limit examples of variables can be considered: if we add to each philosopher a variable indicating whether the philosopher is eating or not, then its explicit modelling leaves the state space unchanged (the corresponding state will be implicit). Viceversa, the addition of an "eating times counter" as explained in Section 3.1, makes the model unbounded.

If the state space growth after introduction of variables is such that state space analysis becomes unfeasible, the other opportunity is to model in a more abstract way the variable(s), however in general this approach is not automatizable and hence we shall not discuss this possibility further in this chapter.

To translate a variable we first have to decide how to encode its value in the model state, then we have to implement in the model the events that cause the variable to change value, finally we need to connect properly the place(s) representing the variable value and the transitions representing a choice based on the variable value.

Let us restrict ourselves to boolean or unsigned integer variables: in this case a single place is enough to encode the variable value. For an unsigned integer variable, its value could be simply represented by the number of tokens in the place, while for the boolean variable, the most natural encoding consists of associating the *false* value with the empty place and the *true* value with one token in the place.

Now let us consider an assignment statement [3] on a boolean variable: Fig. 9 shows the translation rule for the two assignment statements $x := true$ and $x := false$. Observe that assigning a value to a variable consists of *throwing away*

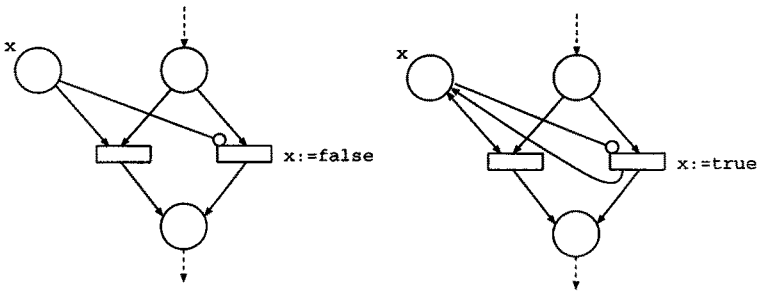


Fig. 9. Translation rule for boolean variable assignment.

its old value and then *storing* its new value, hence if we want to assign the value *true* to boolean variable x we have to add a token (representing the new value) in the corresponding place and possibly also remove a token from it, if its previous value was already *true*. In the case of an unsigned integer variable, getting rid of the previous value may involve a sequence of immediate transition firings to clear the place, followed by the firing of a transition putting in the variable's place as many tokens as the new value to be assigned to the variable. The translation becomes more complex if the value to be assigned is not a constant but the result of an arithmetic operation on other modelled variables (some of these more complex translation rules can be found in [18]). It is important that all the immediate transitions used to model a variable modification must have higher priority than any immediate transition representing a test over the variable, to avoid visibility of intermediate inconsistent values of the variable.

Observe that a communication of a value over a channel is a special case of assignment, and as such it must be modelled in a similar way.

Let us now consider control flow statements that are conditioned on the value of a modelled variable. For the sake of space we have chosen one control flow statement for all,i.e. the if statement, the others being easily derivable from it. When the if statement condition is simply $(x = true)$ then the translation rule for the if statement can be simply modified as shown in Fig. 10(a). The case of more complex boolean expressions involving two or more variables is slightly more complex: the model must comprise a part that generates in an auxiliary place the truth value of the boolean expression, then the marking of

[3] Observe that once it has been chosen that a given control variable has to be modelled, the first translation step must be modified to keep the statements modifying the modelled variable separated from the surrounding sequential code, moreover the second step must be modified to include the variable translation rules described in this section.

the auxiliary place is used to influence the choice of the proper branch of the if statement submodel. In Fig. 10(b) an example is shown where the expression is an *and* between two boolean variables. Observe that the model is such that the marking of the auxiliary place is generated when the control flow statement becomes enabled, and it is cleared after the decision of which branch to follow has been taken.

The extension of the translation rules to unsigned integer variables is not conceptually difficult, more details can be found in [18].

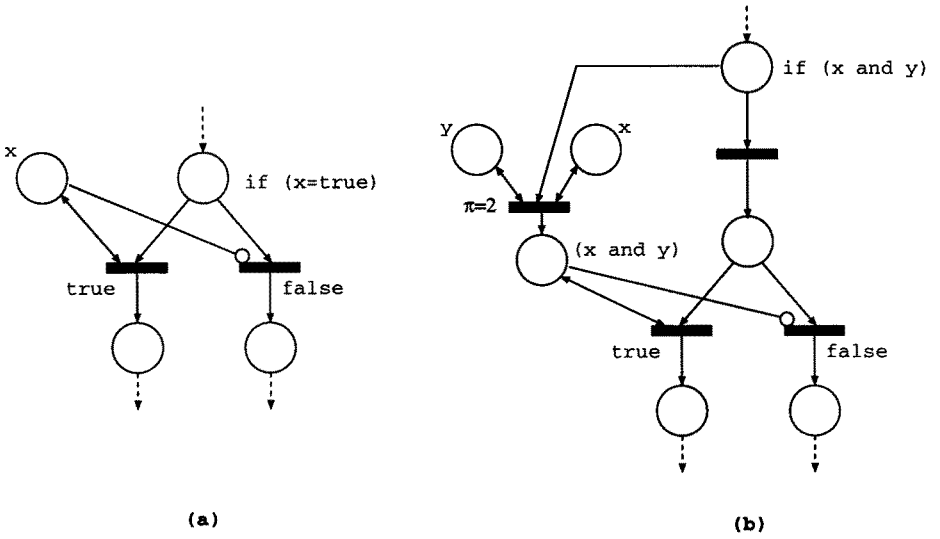


Fig. 10. Translation rule for boolean variable test.

One last remark is important, concerning how to decide which variable should be explicitly modelled. Although it is difficult to give a completely general (and automatically applicable) rule, the basic idea is to identify and explicitly model the variables that control the decision points in the control flow and that can thus have an influence on the possible process interaction patterns.

4.3 Definition of program performance indices at the net level

The example of the previous section has shown a case of program performance analysis: some of the performance indices computed were specific to the philosopher example, but some other were more general (like the weight for the communication graph, or the interference among processes). Notice that also the translation of program performance properties into performance indices of the GSPN model is an error-prone activity, and it is therefore important to define automatic translation also for the indices.

The first step towards an automatic translation is to identify a set of program performance indices that do not depend on a specific application: in [4] relevant indices have been identified related to *phases*, communication, and processing interference.

A phase is a subset of processes that can be active at the same time. Relevant indices with respect to phases are: the set of phases, the probability of each phase, the maximum degree of parallelism, and the minimum degree of parallelism. If P is the program, and N its GSPN model, then the set of phases $F(P)$ can be defined as:

$$F(P) = \cup_{M \in RS(N)} \{act\_proc(M)\}$$

where $act\_proc(M)$ is the set of processes active in marking M and $RS(N)$ is the reachability set of N. The probability $Pr(f)$ of a phase f is computed as the sum of the probabilities of states that have a set of active processes equal to f. From the set of phases it is also possible to compute the maximum degree of parallelism ($max_{f \in F(P)} \mid f \mid$), that provides information on the maximum number of processes that can be performing computation, or communication, at the same time. The mean degree of parallelism provides instead an indication of the mean number of processes that can be working at the same time, and therefore of the mean resource requirement of the program. Its definition is

$$\sum_{f \in F(P)} \mid f \mid \cdot Pr(f)$$

Relevant indices about communication behaviour of a program are: the rate of communication between two processes P_i and P_j all along the program execution ($communication(P_i, P_j)$), and in a specific phase f ($communication_f(P_i, P_j)$). Their definitions in terms of the throughput $X(t)$ (mean number of firing of transition t per unit time) is

$$communication(P_i, P_j) = \sum_{t \in com(P_i, P_j)} X(t)$$

where $com(P_i, P_j)$ is the set of transitions that represent communications between P_i and P_j. The corresponding index for phase f is instead: $communication_f(P_i, P_j) = \sum_{t \in com(P_i, P_j)} X_f(t)$ where $X_f(t)$ is the throughput of transition t while the system is in phase f.

Interference between processes P_i and P_j is defined as the probability of the two processes using the cpu at the same time. For a definition at the net level, we need to identify the set of transitions $Cmp(P_i)$ ($Cmp(P_j)$) that represent computation activities of P_i (P_j), to get

$$Interference(P_i, P_j) = \sum_{t,t': t \in Cmp(P_i), t' \in Cmp(P_j)} Pr(enab(t) \cap enab(t'))$$

where $Pr(enab(t) \cap enab(t'))$ is the probability of markings that enable both t and t'.

5 Translation of a time warp distributed simulator

The example program used to illustrate the automatic translation algorithm is a time warp distributed simulator (a similar example has been presented in [3]). It can be classified as a SPMD (Single Program Multiple Data) program since when it is executed, each processor runs the same "procedure" that implements an event driven Montecarlo simulator of a queueing network (QN) model. Given a QN model to be simulated, it is partitioned into P disjoint submodels (where P is the number of available processors), which are distributed among the P simulator instances. For space reasons, we shall consider the case of P=2.

The simulator instance mapped on a given processor runs asynchronously with respect to all the other simulator instances, however when it handles an event affecting a queue that does not belong to its submodel, it sends a message to the simulator instance responsible for that queue, describing the event and its occurrence time. Each simulator instance has a *local virtual time* (LVT) which is updated locally whenever a new event is processed. When a simulator instance receives a message, there are two possibilities: either the message time is greater than the LVT, in this case the corresponding event lies in the *future* of the simulator instance who received the message and hence it is properly inserted into the local event list, or the message time is less than the LVT, in this case the corresponding event is located in the past and it was not considered by the simulator (that was not aware of its existence), as a consequence a *rollback* is performed that resets the current status to a consistent past status at a time t less than the message time.

This type of distributed simulation is called *optimistic* because every simulator instance runs ahead as much as possible hoping that only few *old messages* will arrive. When a rollback occurs, the simulator must *undo* the events in between the message time and the LVT: this is implemented by keeping an history of the state evolution, i.e., by saving the status at given *checkpoints* in time and keeping track of all processed events from the last checkpoint. Besides bringing back the local clock and status, it is also necessary to undo the effect of the messages that have been sent to other simulator instances in the period of time just canceled: this is implemented by sending an *antimessage* for each message sent in the period of time that must be undone.

Periodically all the simulators execute a protocol that allows them to agree on a *global virtual time* (GVT): after the execution of this protocol each simulator instance can get rid of the information corresponding to checkpoints with time stamp less than the GVT. The GVT protocol usually is initiated by a simulator instance that runs out of space and wants to throw away part of its history. The model that we shall derive does not include the part of the simulator implementing the GVT protocol.

The simulation must terminate when the simulation time reaches a given bound *TEOS*. Since the local clocks advance independently, it may happen that the LVT of one simulator instance reaches TEOS while the other instances have still their LVT less than TEOS. In this situation, the instance that has reached the TEOS cannot just terminate because it could receive a message that might

cause a rollback. Instead, when LVT becomes greater than TEOS the simulator instance switches into a *COMA* state. The *COMA* state hence represents a local end of simulation: the global end of simulation is reached when all the simulator instances are in the *COMA* state. A simulator instance can switch from the *COMA* state to the *active* state when it receives a message that causes a rollback bringing the LVT back to a value less than the *TEOS*

In our example each simulator instance is implemented as a concurrent program comprising three processes running in parallel: a *receiver*, a *transmitter* and a *simulator*. The receiver and transmitter take care of the communication with the other simulator instances, while the simulator performs the actual simulation. The processes are activated by a root processor whose code and corresponding model are shown in Fig. 11 (assuming a simple case of a two simulator instances).

```
root
PAR
    Sim(chanS1T1,chanR1S1)
    Rec(chanR1S1,chanT2R1)
    Tran(chanT1R2,chanS1T1)
    Sim(chanS2T2,chanR2S2)
    Rec(chanR2S2,chanT1R2)
    Tran(chanT2R1,chanS2T2)
```

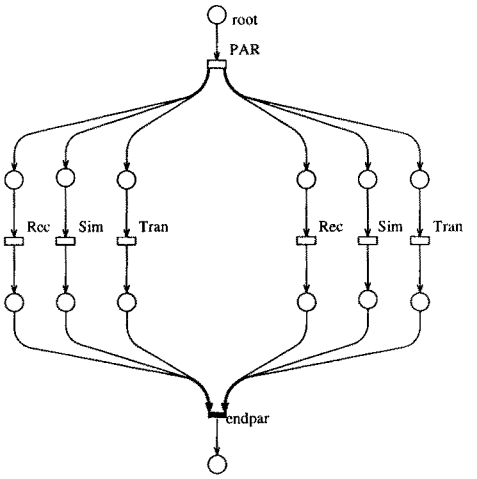


Fig. 11. Root process code and model

In Fig. 12 (left) the simulator procedure is shown. The interface of this process consists of an output channel $chanS_iT_i$ for communications with the transmitter process, and an input channel $chanR_iS_i$ for communications with the receiver process. The simulator performs an endless loop and behaves in a different way depending on its current state: active (LVT $<$ TEOS) or COMA (LVT $\geq$ TEOS). A cycle of the simulator in active state consists of extracting the first event in the event list (if it is not empty) and process it: the status and the event list are updated as well as the LVT, and if needed a message for the outside world is generated and forwarded to the transmitter. Then the simulator checks whether there are new arriving messages from the outside world, if

```
Sim(chanS_iT_i,chanR_iS_i)
seq
LVT = 0
while true
if (LVT < TEOS)
     seq
     if (EventList is not Empty)
     ( get event notice (Event) from event list)
     LVT = Event.time
     if (Event.type = Arrival)
     ( update the event list and server queue)
     else {Event.type = Departure}
     seq
     ( update the event list and server queue)
          if (external destination queue)
               {Prepare Arrival msg)
               chanS_iT_i ! msg
          endif
     endseq
     endif
     endif
     alt
          chanR_iS_i ? msg :
               seq
               if (msg.type = AntiMessage)
               (Delete event from event list)
               else {msg.type = Arrival}
               (Insert arrival in event list)
               endif
               if LVT > msg.time
                    (perform rollback )
                    chanS_iT_i ! RBmsg
               endif
               endseq
          clock ? after timenow plus timeout
     endseq
else { LVT ≥ TEOS: Coma management }
     seq
     chanR_iS_i ? msg
               if (msg.type = AntiMessage)
               (Delete event from event list)
               else {msg.type = Arrival}
               (Insert arrival in event list)
               endif
     if LVT > msg.time
          (perform rollback )
          chanS_iT_i ! RBmsg
     endif
     endseq
end if
end while
endseq
```

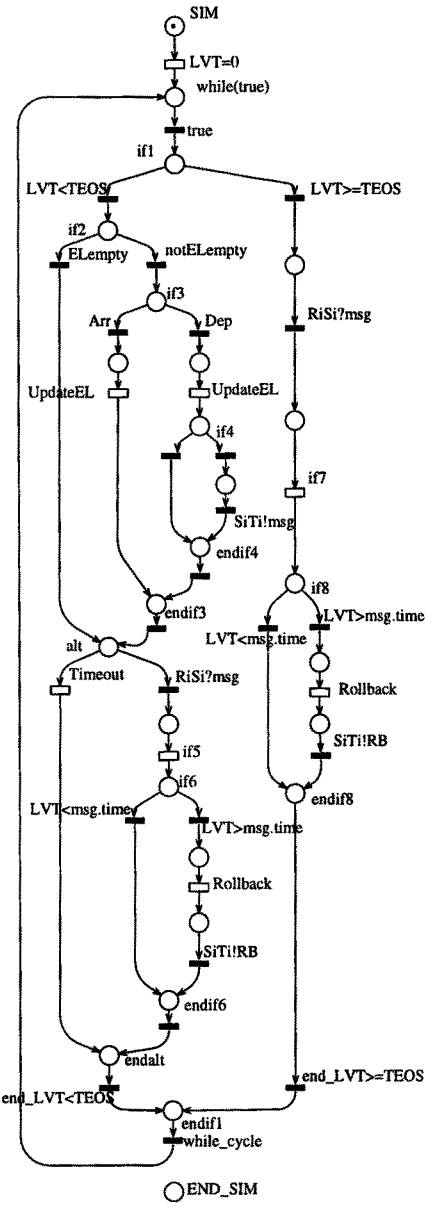


Fig. 12. Simulation process code and model

any is available: this is implemented through an **alt** statement that realizes an input from channel $chanR_iS_i$ with a timeout (input from *special* channel *clock*) to escape if there isn't any new message. If a new message has arrived it first checks its time and if needed performs a rollback and sends a rollback message to the transmitter. If a new message has arrived and its time is greater than the LVT, the corresponding event is simply inserted in the event list. The simulator reaches a *COMA* state when its LVT becomes greater than a predefined maximum simulation time called *TEOS*. As previously mentioned, The *COMA* state represents a local end of simulation. A simulator instance can switch from the *COMA* state to the active state only when it receives a message that causes a rollback bringing the LVT back to a value less than the *TEOS*: for this reason the input statement in the else branch of the if ($LVT < TEOS$) statement doesn't need a timeout escape mechanism. When a simulator performs a change from the active state to the *COMA* state, it initiates a protocol to check if all other simulator instances are in this state and in affirmative case it terminates. In the example program that we are going to present, the termination protocol is not included, i.e., the termination corresponds to a deadlock state of the program.

```
Tran(chanT_iR_k,chanS_iT_i)
while true
chanS_iT_i ? msg
if msg.type = rollback
      while (there are anti-msgs to send)
      ⟨Remove message from history⟩
      ⟨Prepare anti-msg ⟩
      chanT_iR_k ! anti-msg
      endwhile
else
      ⟨Store message in history⟩
      chanT_iR_k ! msg
endif
end while
```

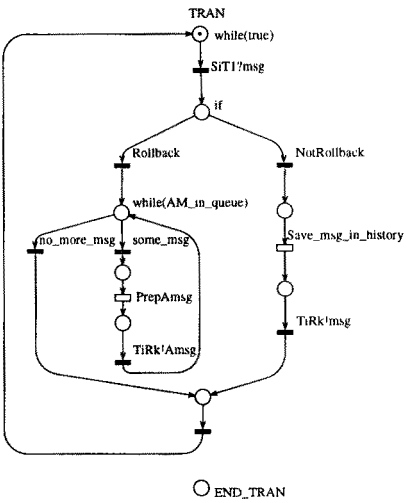


Fig. 13. Transmitter process code and model

In Fig. 13 (left) the transmitter procedure is shown. The transmitter performs an endless loop: it receives a message from the simulator, stores it in a history structure, then forwards it to the destination simulator instance. When a rollback occurs, the transmitter receives a special message from the simulator; in this case it sends one antimessage for each stored message sent in the past with time stamp greater than or equal to the value of the LVT after the rollback.

In Fig. 14 (left) the receiver procedure is shown. It is obtained as parallel

activation of two (sub)processes: a Buffer and a Sender. The reason for structuring the receiver in this way rather than a simple cycle of message reception followed by a message forward, is that the simpler version would lead to a deadlock, as explained in [3]. The Sender subprocess performs an endless loop simply waiting a message from the Buffer (on channel $chanBuf_iSnd_i$), forwarding it to the Simulator (on channel $chanR_iS_i$), and then sending a signal to the Buffer (on channel $chanSnd_iBuf_i$) to notify that it is ready to receive and forward the next message. The Buffer subprocess performs an endless loop awaiting either a message from a transmitter (on channel $chanT_jR_i$) or a signal from the Sender subprocess (on channel $chanSnd_iBuf_i$); this process uses the boolean variable *flag_tx* to keep track of the status of the Sender: *flag_tx* is *true* if the Sender is ready to receive a message from the Buffer, otherwise it is false. When the Buffer receives a message from the transmitter, it inserts the message into a local buffer, then if the buffer is not empty, there are messages to be sent whose timestamp is less than the TEOS, and the Sender subprocess is ready to receive a message (*flag_tx* is *true*), it extracts the first message from the buffer and sends it to the Sender, setting *flag_tx* to false. When the Buffer receives a message from the Sender, if the buffer is not empty, there are messages to be sent whose timestamp is less than the TEOS it extracts the first message from the buffer and sends it to the Sender keeping *flag_tx* to false, otherwise it sets *flag_tx* to *true*.

Now we shall discuss how the GSPN models depicted next to the processes code segments can be automatically obtained by applying the translation rules presented in Subsection 4.1. Let us consider the simulator process: the translation starts from the outer construct and proceeds recursively towards the inner ones: hence the first construct to be translated is a seq of an assignment and a while, place *SIM* is the input place of the assignment and of the whole seq, place *while(true)* is obtained by superposing the assignment exit place and the while entry place, place *END_SIM* is the while (and also the seq) exit place, finally transition *LVT=true* represents the execution of the assignment statement. The reason for place *END_SIM* being isolated is due to the particular condition of the while statement that is always true, which causes the elimination of the t_{exit} transition connecting the entry and the exit place of the while statement. The next step consists of translating the body of the while statement, i.e., an if statement. It is easy to recognize the if entry and exit places and the two transitions labelled *LVT<TEOS* and *LVT>TEOS* representing the choice of either branch of the if, as well as the transitions representing the end of each branch and the exit place (*end_if1*). Let us consider the then branch of the if statement (left portion of the net): it contains a seq comprising an if, followed by an alt. Place *if2* is the entry place of the if and of the seq, place *alt* is the result of superposing the if exit place and the alt entry place, finally place *end_alt* is the seq exit place (that coincides with the alt exit place). Observe that this if statement has an empty else branch, hence we merged the start and end transitions of the corresponding subnet. The subnet corresponding to the translation of the alt statement deserves a little explanation since it starts with a choice between a timed transition and an immediate transition (the output set of place *alt*). This subnet cannot be

obtained by applying the translation rules discussed in Section 4.1; the reason for introducing this new type of subnet is due to the special guard *clock ?* after *timenow* plus *timeout* which implements a timeout: this branch of the alt is taken only if no other guards become enabled within a given time interval *timeout*. It may seem a nonsense to have a conflict between a timed and an immediate transition due to the priority of immediate over timed transitions: in the simulator process model, the transition labelled *Timeout* will never fire. This is due to the fact that at this step of the translation procedure, the model doesn't include the synchronization between communicating processes yet, indeed this aspect is introduced in the model only at the fourth step when the immediate transitions representing input/output statements on the same channel are merged. After the application of the fourth translation step, the enabling of immediate transition $R_iS_i?msg$ will become conditioned on the availability of a token representing the fact that another process (a receiver) is ready to send a message on channel $chanR_iS_i$: if such a token will be missing for a sufficiently long time, then transition *Timeout* will have the opportunity to fire.

The branch of the alt submodel corresponding to the (pure communication) guard $chanR_iS_i$ *? msg*, represents the translation of the seq statement which is to be executed when the corresponding guard is chosen. The seq includes two if statements (the corresponding entry places are $if5$ and $if6$): it is interesting to observe that the two statements are translated in a different way because the first one doesn't contain any concurrent statement in either branch, and as a consequence it is treated as a generic sequential macro-statement, while the second one is translated following the rule of Fig. 6.a because its then branch contains a communication transition. Observe that the model could be reduced by eliminating some of the immediate transitions that are redundant: for example transitions *endif6*, *endLVT<TEOS*, *endLVT>TEOS*, and *while_cycle* could all be eliminated and the corresponding input places merged together and superposed with place *while(true)*, however this operation would worsen the model readability. On the other hand it is not possible to eliminate also transition *LVT<msg.time* in the same way because it is in structural conflict relation with another transition (*LVT>msg.time*).

Let us consider the translation of the receiver process: the three GSPN submodels depicted in Fig. 14 (right) represent the processes Receiver, Sender and Buffer. It is interesting to briefly discuss the buffer process submodel because it comprises the representation of the boolean control variable $flag\_tx$: if we didn't model this variable the receiver model could easily cause a deadlock (buffer trying to send a message to a sender that is not ready to receive it). The variable is represented by a the place labelled *flagtx*: the presence of a token in this place means that the current value of the variable is *true*, the absence of tokens in this place means that the variable is set to *false*; initially this place is empty. There are two type of transitions connected to this place: transitions representing choices that test the place without modifying it and transitions that modify the marking of the place. It can be easily recognized the translation schema for the assignment presented in Fig.9 (transitions *setflag_tx* and *resetflag_tx*) and the

translation schema of the if taking into account the value of a modelled variable (transitions *flagtx_false* and *flagtx_true*).

Finally, let us discuss the effect of applying steps 3 and 4 of the translation procedure: in order to deal with models of manageable size, we will first consider in details only the receiver subnet portion, then we will show how the complete program model looks like, without further explanation. In Fig. 15 is shown the result of applying the translation step 3 to the receiver process: the two transitions labelled *Sender* end *Buffer* in the **Receiver** submodel are substituted with the corresponding subnets. The initial transition, labelled **par** represents the activation of the two subprocesses, the subnet on the left represents the **Buffer** submodel while the subnet on the right represents the **Sender** submodel.

```
Rec(chanR_i S_i,chanT_j R_i)
par
   Buffer(chanBuf_i Snd_i,chanT_j R_i)
   Sender(chanR_i S_i,chanBuf_i Snd_i)

Buffer(chanBuf_i Snd_i,chanT_j R_i)
seq
flag_tx = true
COMA = false
while true
alt
   chanT_{i-1} R_i?msg :
   seq
   ⟨insert msg in buffer⟩
   if flag_tx
   if not(COMA ∧ (∀ msg in buf, msg.time>TEOS)) then
       ⟨ extract msg from buffer ⟩
       ⟨ update COMA ⟩
       chanBuf_i Snd_i!msg
       flag_tx=false
   end if
   end if
   end seq
   chanSnd_i Buf_i?end :
   if not(COMA ∧ (∀ msg in buf, msg.time>TEOS)) then
       ⟨ extract msg from buffer ⟩
       ⟨ update COMA ⟩
       chanBuf_i Snd_i!msg
       flag_tx=false
   else
       flag_tx=true
   end if
end while
end seq

Sender(chanR_i S_i,chanBuf_i Snd_i)
while true
seq
   chanBuf_i Snd_i?msg
   chanR_i S_i!msg
   chanSnd_i Buf_i!end
end seq
end while
```

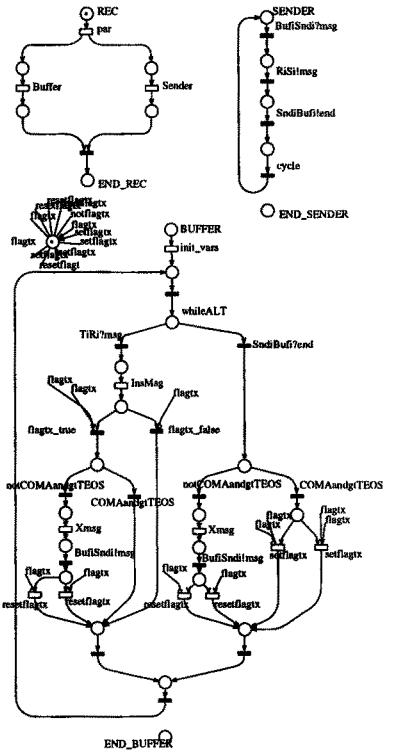


Fig. 14. Receiver code and model

In Fig. 16 is shown the result of applying translation step 4 to the receiver submodel portion.

We avoid showing the complete model of the whole program in a figure because it would hardly fit a page. The use of an High-Level PN formalism (e.g. Stochastic Wel-Formed Nets, presented in Chapter[10]) would allow us to represent in a more compact way the model, however it would be still quite

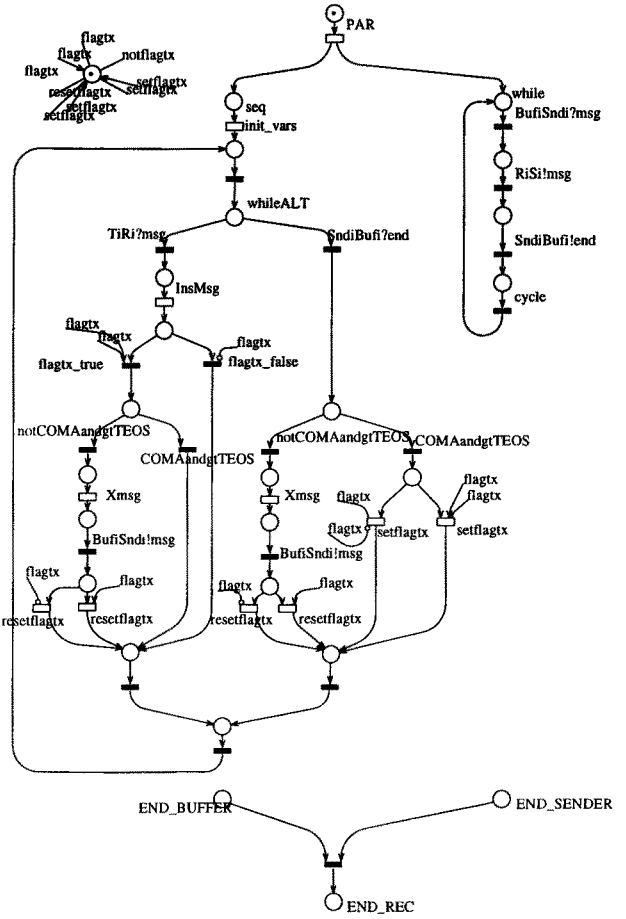


Fig. 15. Receiver model after step 3

hard to read. Actually the aim in practical use of the technique described in this chapter is to allow the user, i.e. the program developer, to avoid ever seeing the automatically generated GSPN model unless he/she wishes to do so.

This goal can be achieved by introducing the possibility of obtaining performance indices on the program automatically: for the case of general performance indices, this is possible since given their definition it is not difficult to produce a formula expressing the performance indices of interest as a function of the basic results obtained from the steady state solution of the CTMC underlying the GSPN model, written in the correct syntax accepted by the tool used to get the solution. Examples of general (and automatically generated) performance indices have been given in Section 4.3.

In the specific case of the time warp simulation model however, there are some specific performance indices of interest, like for example the ratio between the useful work (throughput of transitions representing an advance of the simulation) and useless work (transitions representing a rollback) as a function of the probability of getting messages with an *old* timestamp (weight of transitions $LVT > msg.time$ and $LVT < msg.time$), or the probability for a process of type *Sim* of sitting idle waiting for a new event coming from another simulator (probability of markings in which the marking of the simulator process submodel enables a transition representing an input message statement, but the marking of the corresponding sender submodel is such that the synchronization is not possible).

However the definition of performance indices specific of the particular program under study requires to directly deal with the GSPN model of the program, since at the moment the technique does not include a system to define program specific performance indices directly on the program code.

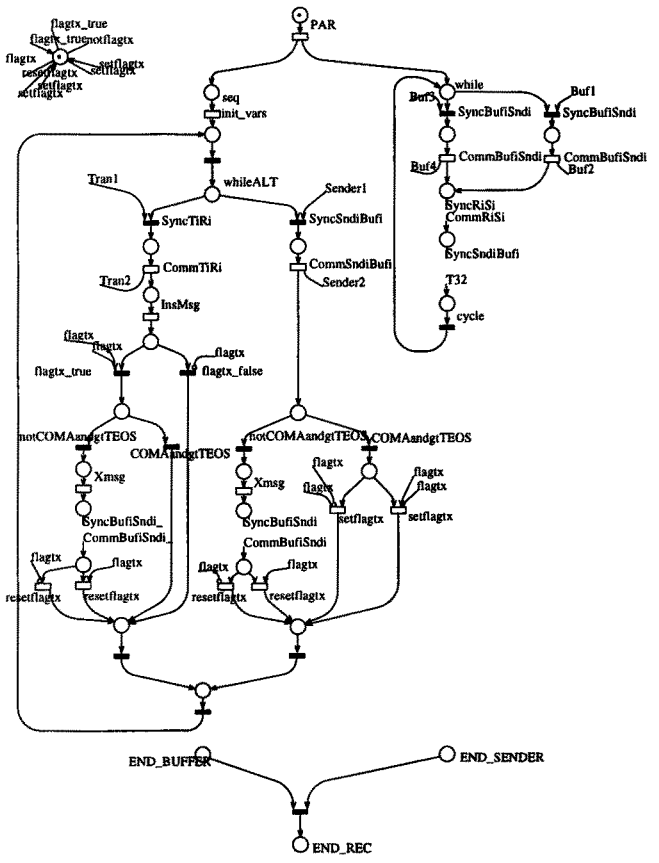


Fig. 16. Receiver model after step 4

One last remark is important about the GSPN model of the distributed simulator program: as observed at the beginning of this section, the program can reach a deadlock state corresponding to the situation in which all simulators are in the COMA state, and no messages are circulating in the network. This situation corresponds to the termination of the program (that could be detected by activating a distributed termination detection protocol). This state is indeed the only deadlock of both the program and the GSPN model. In order to compute the steady state performance indices we need to make the GSPN model cyclic (*ergodic*), this is done by adding a *restart* transition enabled only in the deadlock state and bringing the model back in the initial marking.

6 Conclusions

In this chapter we have discussed the role of GSPN for the integrated qualitative and quantitative analysis of distributed software, to help the programmer decide whether a program is correct and it does meet the required performance. The quantitative analysis is performed under an infinite resource hypothesis, hence it allows to establish the best performance that the program can ever achieve at run-time. Moreover it allows to compute program performance estimates that can be used as input data of classical algorithms for mapping and load balancing. The analysis performed here belongs to the class of "static" approaches, that typically model only the flow of programs, nevertheless we have shown that variables can play an important role in increasing the efficacy of the approach, and we have discussed a translation schema that allows them to be included in the GSPN model of the program.

Despite the fact that GSPN modelling has been illustrated here has a "final" step in the software development activity, it can indeed be seen as a help to the software designer or programmer during the whole development cycle: since it is based on the construction of an abstract model of the program, it does not require the program to be completely defined for the model to be built. Starting from a (partial) specification of the code of the distributed application, a corresponding abstract, formal GSPN model of the application can be constructed and analyzed. So GSPN modelling can be seen as a support tool of the whole program development cycle.

A very important factor for the applicability of the analysis in contexts where there is a low expertise in modelling or in performance evaluation and/or in Petri nets, is the possibility of automatically producing the model and of automatically compute program related quantities. In the translation process presented, a number of program performance indices are generated so that the programmer gets directly program related quantitative results. Qualitative properties instead, are expressed in terms of the Petri net model (e.g., a deadlock is a deadlock in the model and no translation into a program state deadlock is given, liveness is liveness of the net transitions, etc.). A necessary step towards a more complete translation schema is that of defining and automatically producing a set of *program qualitative properties*, in terms of net qualitative properties,

so that the analysis may produce both quantitative and qualitative properties of the program (rather than of the corresponding GSPN model). The GreatSPNtool that we have adopted for analysis, does not allow to define and check general qualitative properties, but it is not difficult to integrate it with tools like PROD [17, 21] that include facilities for checking general qualitative properties expressed through temporal logic formulas, using a model checking approach. The PEP tool project [13] represents an interesting and successful attempt of automatic translation of concurrent programs into PN models with the possibility for the user to perform the analysis at its favorite representation level (either the program or the PN), however this tool covers only the qualitative analysis aspects not allowing any type of timing specification and quantitative analysis.

The distributed simulation example has shown the need to compute, in addition to predefined program indices, also a set of program-specific ad hoc quantities, that should as well be defined without passing through the GSPN model representation (whose graphical representation might be simply too big to fit on a screen). Since in general we cannot assume that the programmer is an expert in nets and performance, and since anyway complex programs are represented by complex GSPN models which are difficult to deal with also by an expert, it is not possible to leave this formidable task to the programmer because this is an error prone activity: the solution could be the definition of a language for the specification of qualitative and quantitative indices in program terms, and in the automatic translation of this expressions in net terms.

References

1. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets.* J. Wiley, 1995.
2. M. Ajmone Marsan, A. Bobbio, and S. Donatelli. *Petri Nets in Performance Analysis: an Introduction.* In this book.
3. G. Balbo, S. Donatelli, and G. Franceschinis. Understanding parallel program behavior through Petri net models. *Journal of Parallel and Distributed Computing*, 15(3):171–187, July 1992.
4. G. Balbo, S. Donatelli, G. Franceschinis, A. Mazzeo, N. Mazzocca, and M. Ribaudo. On the computation of performance characteristics of concurrent programs using GSPNs. *Performance Evaluation*, 19:195–222, 1994.
5. 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.
6. F. De Cindio and O. Botti. Comparing Occam2 program placements by a GSPN model. In *Proc. 4^{th} Intern. Workshop on Petri Nets and Performance Models*, pages 216–221, Melbourne, Australia, December 1991. IEEE-CS Press.
7. S. Donatelli and G. Franceschinis. The PSR methodology: integrating hardware and software models. In *Proc. of the 17^{th} International Conference in Application and Theory of Petri Nets, ICATPN '96*, Osaka, Japan, june 1996. Springer Verlag. LNCS, Vol 1091.

8. S. Donatelli, G. Franceschinis, N. Mazzocca, and S. Russo. Software architecture of the EPOCA integrated environment. In *Proc. of the 7^{th} Intern. Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, May 1994.
9. S. Donatelli, G. Franceschinis, M. Ribaudo, and S. Russo. Use of GSPNs for concurrent software validation in EPOCA. *Information and Software Technology*, 36(7):443–448, 1994.
10. G. Franceschinis and M. Ribaudo. *Efficient Performance Analysis Techniques for Stochastic Well-formed Nets and Stochastic Process Algebras.* In this book.
11. U. Goltz and W. Reisig. CSP programs as nets with individual tokens. In *Proc. 5^{th} Intern. Conference on Application and Theory of Petri Nets*, Aarhus, Denmark, June 1984.
12. Ursula Goltz. On representing CCS programs by finite Petri nets. In *Mathematical Foundations of Computer Science*, volume 324 of *LNCS*, 1988.
13. Bernd Grahlmann. The Reference Component of PEP. In Ed Brinksma, editor, *Proceedings of TACAS'97 (Tools and Algorithms for the Construction and Analysis of Systems)*, volume 1217 of *Lecture Notes in Computer Science*, pages 65–80. Springer-Verlag, April 1997.
14. C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
15. R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer-Verlag, Berlin, Germany, 1980.
16. T. Murata, B. Shenker, and S. Shatz. Detection of Ada static deadlocks using Petri nets invariants. *IEEE Transactions on Software Engineering*, 15(3):314–326, March 1989.
17. Leo Ojala and Johan Lilius. Development and reachability analysis tools at Helsinki University of Technology. *Petri Net Newsletter*, (47):36–42, 1994.
18. Lucia Ragliani. Analisi di programmi paralleli mediante modelli GSPN, june 1994.
19. S.M. Shatz and W.K. Cheng. A Petri net framework for automated static analysis of Ada tasking. *The Journal of Systems and Software*, 8:343–359, October 1987.
20. R. Taylor. A general purpose algorithm for analyzing concurrent programs. *Communications of ACM*, 26, May 1983.
21. Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, and Tino Pyssysalo. Prod reference manual. Technical Report B13, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, August 1995. PROD home page URL: http://saturn.hut.fi/pub/prod/index.html.