Monika Heiner, Louchka Popova-Zeugmann

Worst-case Analysis of Concurrent Systems with Duration Interval Petri Nets

Reihe Informatik I-02/1996

February 1996

Worst-case Analysis of Concurrent Systems with Duration Interval Petri Nets

Monika Heiner
Brandenburg University of Technology
Dep. of Computer Science
Karl-Marx-Straße 17, D - 03013 Cottbus
mh@informatik.tu-cottbus.de

Louchka Popova-Zeugmann
Humboldt University
Dep. of Computer Science
Lindenstraße 54a, D - 10099 Berlin
popova@informatik.hu-berlin.de

Abstract:
This paper deals with computing the minimal and maximal execution durations in a given concurrent system in order to support software dependability engineering by assuring the meeting of prescribed deadlines. For that purpose, a new kind of time-dependent Petri nets - the Duration Interval Petri net - is introduced, and a dedicated reachability graph is defined in a discrete way. Using this reachability graph, shortest and largest time paths between two arbitrary states of the concurrent system, and by this way minimal and maximal software execution times, can be computed. These results have been applied to a medium-sized reactive system.

Keywords: dependable software, software validation, performance evaluation, time-dependent Petri nets, programming languages, flexible manufacturing.

1 Introduction

Starting from the taxonomy of dependability [Laprie 95], methods of software dependability engineering can be divided into two groups - methods to improve the software dependability by fault avoidance or fault tolerance (procurement) and methods to predict the reached degree of software dependability (fault forecasting). Among those methods, which aim at the improvement of software dependability, different kinds of Petri net based validation techniques to avoid faults during the development phase have attracted a lot of attention in the last decade. Within this general framework various Petri net based methodologies of parallel and distributed (shortly called concurrent) software engineering have been outlined. Because of the crucial impact of performance on concurrent systems [Jelly 94], special emphasis has been laid in the last years on incorporation of performance criteria as part of the system development cycle (see e.g. [Balbo 92], [Donatelli 94], and [Heiner 94a]).

Beyond that and maybe most importantly, the Petri net based approach to concurrent software engineering is able to combine different kinds of software validation methods as well as a structure-oriented approach to reliability prediction on the basis of a common Petri net-based intermediate representation of the concurrent software system under development.

Because of its generality, the Petri net framework can be used with concurrent systems expressed in a wide variety of specification or programming languages. The semantics of a particular language are captured by a procedure for automatically deriving Petri net representations for any parallel system expressed in this high-level language. The modelling of parallel systems by Petri nets resulting in an intermediate representation yields several advantages.

First, tools for analyzing and reduction extract information about events and their ordering directly from this intermediate representation of the system. Therefore, they do not rely on any special assumptions, but regard general features of parallel systems and can be applied to any system once a Petri net representation for the system has been obtained.

Second, due to the generality of the Petri net approach, tools based on Petri nets can be extended to provide common analysis methods across a number of phases in the software development process as soon as some formalized description of the parallel system under development is available. This might be a source of valuable commonality and integration in a software development environment.

Third, the net-based intermediate program representation serves as a common root from which different net-based software validation methods, basically on communication/synchronization level, are able to start. This diversity of methods covers (1) qualitative analysis (in terms of context checking of general semantic properties by classical Petri net theory and verification of special semantic properties especially by temporal logics), (2) monitoring and systematic testing as well as (3) quantitative analysis (in terms of performance evaluation/prediction and reliability prediction).

In [Heiner 92], a method is demonstrated how to develop at first qualitative models as place/transition nets suitable for analysis of general and special qualitative properties. In [Heiner 94b], [Wikarski 95], the validated qualitative models are transformed step-by-step by quantitative expansion and property-preserving structural compression into quantitative models (as Markovian object nets - a special type of stochastic Petri nets) in order to predict performance behaviour.

Based on this experience, the approach to integrate different methods on a common representation is extended by a formal method to derive Petri net models suitable for a structure-oriented worst-case prediction of the system’s timing behaviour. For that purpose, we are going to introduce a dedicated kind of time-dependent (non-stochastic) Petri nets, allowing definite statements, not only probability-based ones.

This paper is organized as follows. In section 2, a concise summary is given on the general framework of Petri net based software dependability engineering. After having introduced the basic terminology, the general procedure for the newly introduced approach of Petri net based worst-case prediction of concurrent software systems is outlined in section 3. Some more details of the corresponding mathematical background are presented in section 4. Using these results, a typical example of software dependability engineering is sketched in section 5. Finally, conclusions and outlook on further work will be given in section 6.

2 Petri Net Based Methods for Software Dependability - Overview

A classification of software validation techniques should separate the validation methods (main principles) on the one side and the properties to be validated on the other side. These software validation techniques should then be embedded in a process model to develop software with high dependability demands. A suitable order of validation methods takes into account that

- validation should be applied as early as possible,
- the proper functionality is a prerequisite for an evaluation of quantitative properties, and
- the expected functionality can only be guaranteed in any case if all consistency conditions of context checking have been fulfilled.

Within this general framework a Petri net based methodology of software dependability engineering can be outlined. But, different validation methods may require net models which vary partly in their level of abstraction. This variety comprises of course typical quantitative parameters as delay or branching information (which are obviously necessary in case of quantitative analysis), but also the granularity of considered control and/or data flow, i.e. the degree of details concerning structural information. Therefore, in order to integrate qualitative as well as quantitative analysis on a common intermediate software representation, an important feature of a related methodology is the ability of a controlled structural reduction, combined with compression of any quantitative parameters.

In [Heiner 95b], a method is outlined how to develop these models step-by-step (see Figure 1):

- qualitative models
  - control structure models as place/transition nets
    (Any branching information is neglected. Every structurally possible path of the model is considered to be realizable in the software. The set of execution paths of the model is greater or equal as the software's execution path set.)
  - control flow models as coloured nets
    (All (finite discrete) control variables determining the actually control flow are added to the control structure model. So, the control flow model is generally much larger than the control structure one. But model and software have the same set of execution paths.)
- quantitative models
  - performance models as duration, interval, duration interval[1)] or stochastic Petri nets
    (depending on the type of performance measures to be evaluated)
  - reliability models as stochastic Petri nets [Heiner 95a]

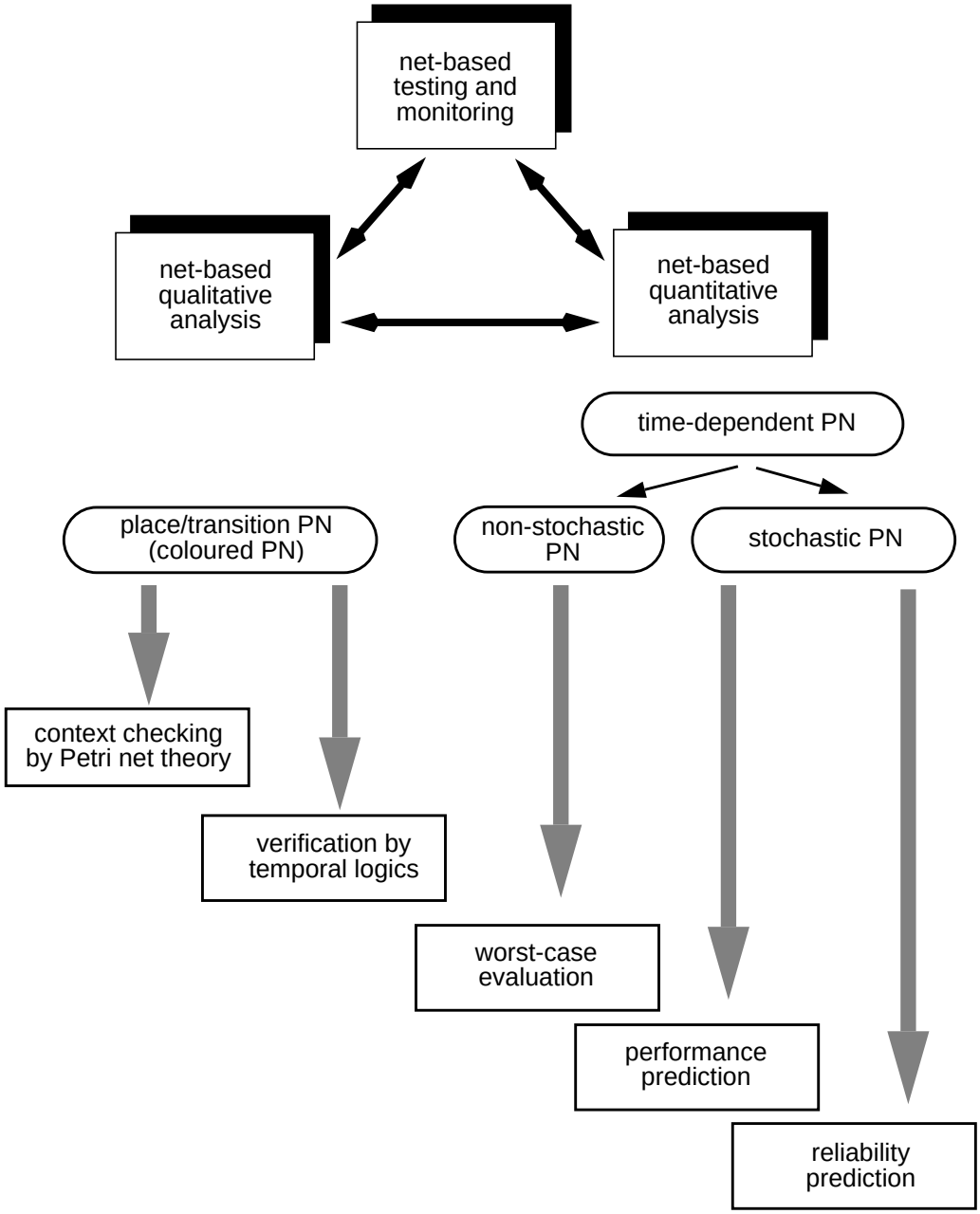
All transformations (from the concurrent software system description into a first qualitative Petri net model, and between the different kinds of net models) can be done formally, and therefore automated to a high degree.

Obviously, the individual validation techniques are essentially influenced by the analyzing possibilities available for the corresponding net classes.

(1) The validation of qualitative properties comprises two steps. At first, the context checking of general semantic properties (basically liveness properties) is done by a suitable combination of static and dynamic analysis techniques - mainly of (classical) Petri net theory. Afterwards, the verification of well-defined special semantic properties (among them safety properties) given by a separate specification of the required functionality is performed. During this second step, the power of classical Petri net theory is supplemented by the model checking approach, using temporal logic as a flexible query language for asking questions over the (complete/reduced) set of reachable states.

1) Introduced in this paper, compare section 3 and 4.

Figure 1: The Petri net approach to integrated software dependability engineering.

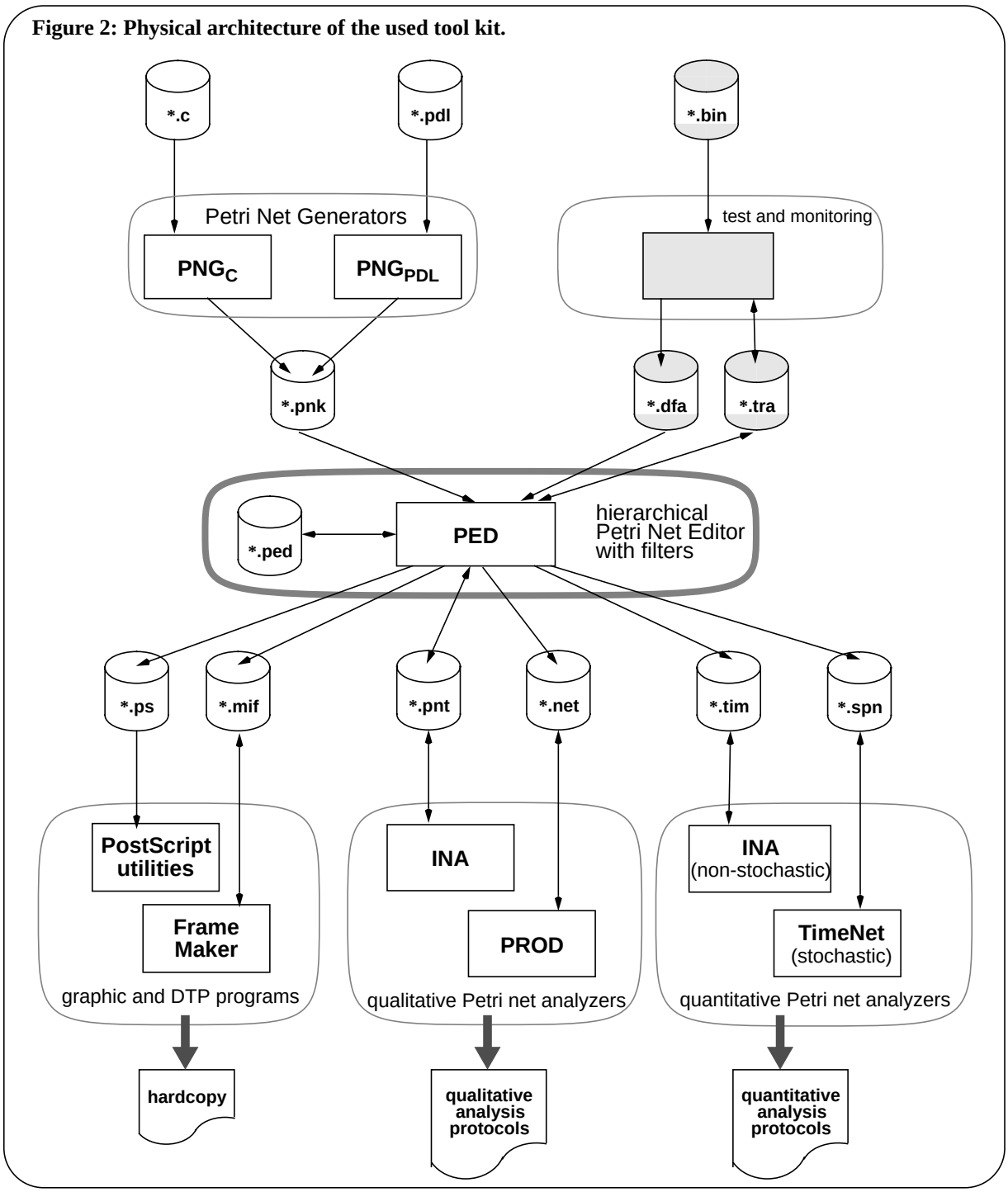


(2) The objectives of the monitoring and testing component are twofold. Besides the provision of the quantitative attributes according to the user-driven time abstraction level, the net-based testing method supports a systematic test of parallel systems. Techniques to derive automatically dedicated test suites and to measure the test coverage obtained are important features of this systematic testing.

(3) The validation of quantitative properties has to be based on quantitative net models. The frequency and maybe also delay attributes necessary to transform qualitative models into quantitative ones are provided by the monitoring and testing component or, in simple cases, calculated from the basic instruction sequences. The choice of a suitable time-dependent net type depends on the kind of properties to be validated (comp. section3).

The tool kit currently used is sketched in Figure 2. It is available on UNIX/LINUX computers. The Petri Net Generators in use (PNG_{PDL} [Grzegorek 91] and PNG_C [Kobienia 96]) produce hierarchical (place/transition) Petri net representations of the (reduced/non-reduced) control structure of a given sequential process. Basically, this generation assembles general Petri net components for all relevant language means according the syntax tree. Besides this basic functionality, further information is supplied which allows an automatic layout of the generated net afterwards, and an assessment of the program's structural complexity (Number of Acyclic Paths). For more details see also [Heiner 92].

Figure 2: Physical architecture of the used tool kit.



After that, the graphical Petri net EDitor PED [Czichy 93] is used for visualization and linking of the generated Petri net parts, and for adding any supplements that might be required, e.g.

- modelling of all control variables and related operations, upgrading the generated control structure model to a control flow one (the automatization of this step is in preparation),
- modelling of the system environment behaviour,
- fault models (e.g. message loss or duplication induced by erroneous message channels).

Finally, the filtering capabilities of PED output the particular data structures required by the analysis tools in use (INA [Starke 92], PROD [Varpaaniemi 95], TimeNet [German 94]). These analysis tools have to be understood as implementations of well-proven theorems of Petri net theory. As a consequence, the results obtained by these tools can only be in terms of Petri net theory independent of any underlying special semantics of the net or software under development.

3 A Petri Net Based Method for Worst-case Timing Prediction

We are now going to extend the approach to integrate different methods on a common representation by proposing a new kind of time-dependent Petri nets especially dedicated to worst-case prediction of a system's timing behaviour.

The choice of a net type, at best suitable for a given validation task, should be guided by the well-known engineer's basic principle to keep everything as simple as possible. So the answer to the question, which net type should be chosen, depends on the properties to be validated. As long as there are hard deadlines to meet definitely, as it should be the case for systems with predictably timing behaviour, the exact evaluation by non-stochastic nets is unavoidable. Only when average values or probability distributions of performance measures like load, throughput, utilization etc. are wanted, then the application of stochastic Petri nets becomes useful. According our analysis objective of worst-case prediction, we will restrict ourself in the following to discuss only non-stochastic Petri nets.

A search through the literature reveals a lot of different time-dependent Petri net classes, which differ essentially in the provided time concepts. For a concise summary see [Starke 95]. Due to our modelling procedure, which maps any atomic sequential parts (without any internal communication statements, for more details see [Heiner 92]) to transitions, time consumption should be connected with transition firing. Among those, the most important time-dependent net types are the following:

- **duration nets** (usually called timed nets) [Ramchandani 74]:
  constant delays with non-preemptive firing principle,
  firing consumes time,
  earliest firing rule realized by maximal step strategy;
- **interval nets** (usually called time nets) [Merlin 74]:
  interval delays with preemptive firing principle,
  firing itself happens timeless,
  latest firing rule realized by single step strategy.

The firing of a transition on model level corresponds to the execution of the actual atomic sequential program part mapped to it. Generally, such an execution cannot be interrupted again after being initiated (in case of modelling software systems without timers, interrupts etc.). So, the non-preemptive firing principle is the natural way to express the software's execution progress. On the other side, an adequate model for worst-case prediction of the timing behaviour should be

able to determine exactly the minimal and maximal time consumption of critical system parts. So what we really need for our purposes is a suitable combination of the properties listed above - *interval delays*, as used in interval nets, and *non-preemptive firing rule*, as used in duration nets. That leads us to introduce a corresponding new time-dependent Petri nets - the duration interval Petri nets. For a formal definition of the corresponding net type see section 4.2.

In the following, the quantitative parameters are time intervals of the minimal and maximal duration to execute a given (sequential) part without resource limitations (no processor sharing, no communication delays, etc.). Of course, the interval may collapse to a constant duration in case of purely linear, non-interrupted statement sequences. These execution time intervals can be measured by monitoring tools of the development environment used, or - in special cases - calculated from machine instruction sequences.

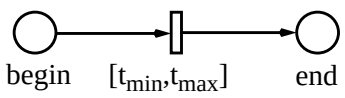
Now we apply the procedure presented in [Heiner 94b], [Wikarski 95] for the systematic, step-wise development of stochastic software performance models to derive in a similar way a Petri net model for worst-case timing prediction. Again, the model, which we need for the evaluation to be done, should not be built from the scratch, but instead of this, the timing model should be derived to a high degree automatically from that net models which we do have due to our qualitative analysis efforts done. Because we already know, how to map software onto (place/transition) Petri nets [Heiner 92], we have then altogether a formal method to derive systematically Petri net models suitable for worst-case prediction of the concurrent software system under development.

The method proposed consists basically of two components:

- a set of reduction rules describing the allowed structural reductions and the corresponding transformation rules of the quantitative parameters within any well-structured sequential substructures (see Figure 3), and
- a method to compute the execution interval of any (non-sequential) system part, including an arbitrary net structure, by length determination of the shortest and largest paths between the corresponding (begin and end) states of the reachability graph (see section 4.3).

If the structural reduction abstracts of cycles, we need the lower and upper bounds of cycle iterations in order to be able to calculate the new time interval. In case of software worst-case prediction, we can extract the necessary iteration bounds from the corresponding loop statements - provided they are static ones. But this is a well-known unavoidable restriction for systems with high dependability demands which require predictably timing behaviour (see e.g. [Kopetz 95], [Vrchoticky 94]).

The total execution time interval of given software can be immediately picked up from the reduced Petri net in case of well-structured sequential programs which have been completely reduced to[1)]



Generally, in case of concurrent programs, the time interval has to be evaluated by a suitable Petri net evaluation tool (we use INA, lately updated according the method proposed here). But a structural reduction combined with compression of quantitative parameters, done before as strong as possible, may reduce the computational costs essentially.

1) In case of “loop forever”-programs, begin and end places coincide.

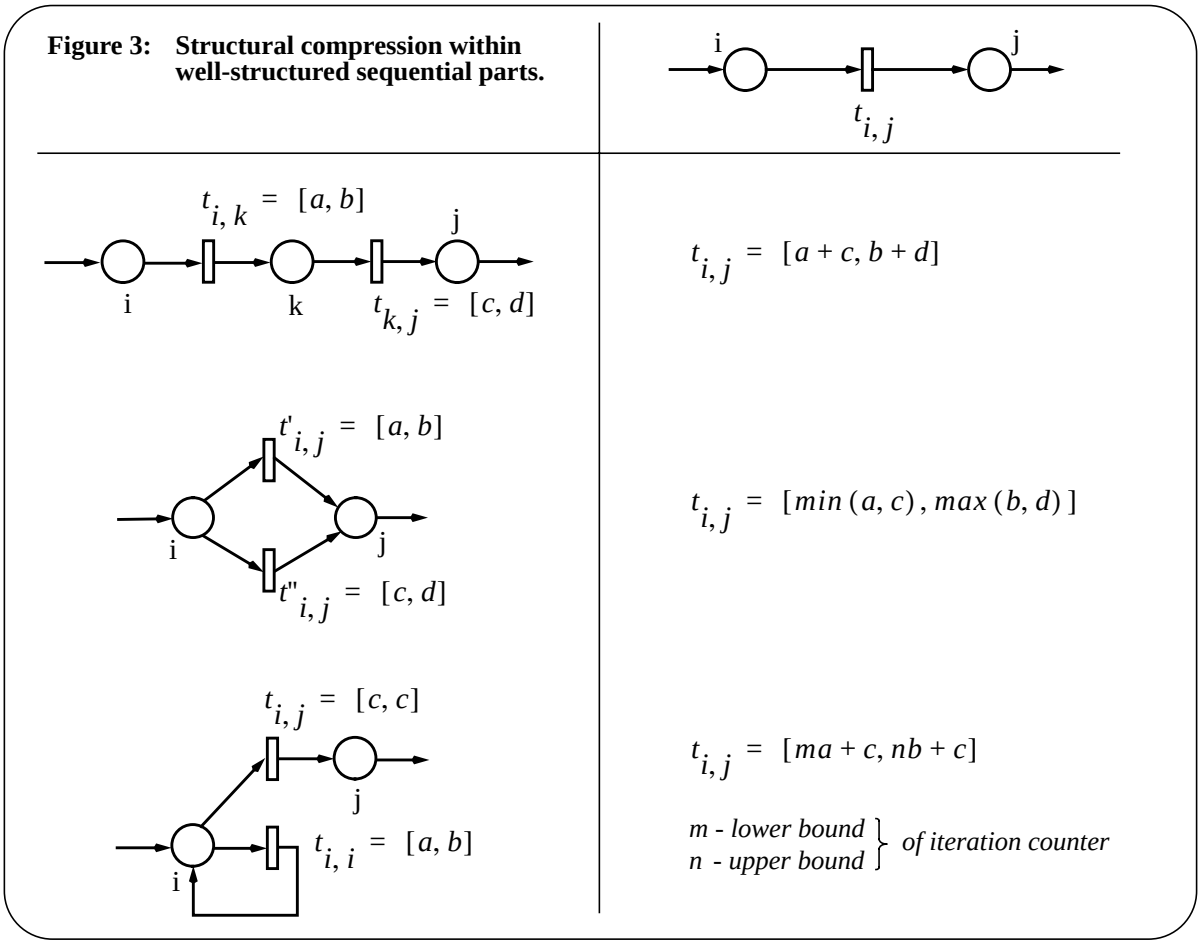


Figure 3: Structural compression within well-structured sequential parts.

4 Mathematical Background

4.1 Basic Notations and Definitions

We will use the following notations. $\boldsymbol{N}$ denotes the set of natural numbers, $\boldsymbol{N}^+$ is $\boldsymbol{N} \setminus \{0\}$. $\boldsymbol{Q_0}^+$ is the set of nonnegative rational numbers. Let g be a given function from A to B. Then, the symbol # defines a special value from the set g(A).

Definition 1:

The structure $PN = (P, T, F, m_0)$ is called **Petri net**, iff:

(1) P, T, F are finite sets. We define $X := P \cup T$. We assume
$P \cap T = \emptyset$, $P \cup T \neq \emptyset$, $F: (P \times T) \cup (T \times P) \rightarrow \boldsymbol{N}$, and
$\forall x \in X : \exists y \in X : F(x, y) \neq 0 \vee F(y, x) \neq 0$

(2) $m_0: P \rightarrow \boldsymbol{N}$ (initial marking)

A marking of a PN is a function $m: P \rightarrow \boldsymbol{N}$, where $m(p)$ denotes the number of tokens in place p. The pre- and postsets of a transition t resp. of a place p are given by

$Ft := \{p | p \in P \wedge F(p, t) \neq 0\}$ and $tF := \{p | p \in P \wedge F(t, p) \neq 0\}$ resp.
$Fp := \{t | t \in T \wedge F(t, p) \neq 0\}$ and $pF := \{t | t \in T \wedge F(p, t) \neq 0\}$.

Each transition $t \in T$ induces the marking t^- and t^+, defined as $t^-(p) := F(p,t)$ and $t^+(p) := F(t,p)$. By Δt we denote $t^+ - t^-$. A transition $t \in T$ is enabled (may fire) at a marking m iff $t^- \leq m$ (i.e. $t^-(p) \leq m(p)$ for each place $p \in P$). When an enabled transition t at a marking m fires, a new marking m' given by $m'(p) := m(p) + \Delta t(p)$ is reached.

4.2 Duration Interval Petri Nets

In order to design and analyze such systems as given above, here we present a new kind of time dependent Petri nets where the firing of each transition costs time. Generally, this time cannot be given as a fixed number but it ranges between a minimal and a maximal value. For that reason we will call these nets Duration Interval Petri nets (DIPN). The DIPN are classical Petri nets where to each transition t two nonnegative rational numbers a_t and b_t ($a_t \leq b_t$) are associated; a_t is the minimal possible value of the firing duration of t, and b_t is the maximal possible value. The times a_t and b_t are relatively to the moment at which t was enabled last. When the transition t becomes enabled it starts firing immediately, provided any dynamic conflict[1)] is resolved to its favour. The conflict resolution policy is as usual as in classical Petri nets, i.e. there is a free choice among the enabled transitions.

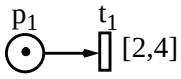
The firing rule can be basically considered as a 3-phase firing:

- The tokens are removed from the input places when a transition starts working (firing).
- The transition holds the token(s) while working (time elapse).
- The tokens are put into the output places when a transition finishes working.

The token transfer itself does not consume any time.

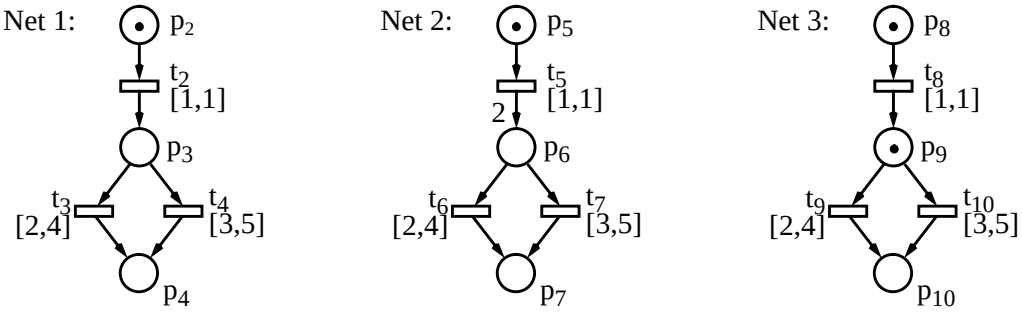
To illustrate the firing behaviour in more detail, let's consider two examples.

Example 1:



Assuming that t_1 becomes enabled at time c, then t_1 starts to fire at this moment and it finishes at the earliest at c+2 and at the latest at c+4, or it finishes at an arbitrary moment between c+2 and c+4.

Example 2:



In Net 1, only one of the transitions t_3 and t_4 can and will fire (after firing of t_2). In Net 2, after the firing of the transition t_5, the place p_6 gets two tokens. Both transitions t_6 and t_7 fire immediately (t_6 and t_7 are in static conflict[2)], but not in a dynamic one). In Net 3, transition t_9 and t_{10} are in a

1) Two transitions are in a **dynamic conflict** at the marking *m*, if both transitions are enabled at *m*, but the firing of one transition disables the other one (comp. [Starke 90]).

dynamic conflict, but the situation is another one as in the Net 2. Assume that the conflict between t_9 and t_{10} is solved in favour of transition t_9, then after one time unit, the token, which is situated in place p_8 will move to place p_9. Since transition t_9 is still firing (but not finished, yet), transition t_{10} starts to fire (one time unit later then t_9).

As demonstrated with the Net 2 and Net 3, multiple firing of a transition (concurrently to itself) is not allowed, as usual in time-less Petri nets. Moreover, in our context of software modelling this assumption reflects the fact that at any time only one process can run over a certain piece of code.

Definition 2:

The structure $D = (P, T, F, m_0, DI)$ is called **Duration Interval Petri Net** (DIPN) iff:

(1) (P, T, F, m_0) is a Petri net (skeleton of D),

(2) $DI: T \rightarrow \boldsymbol{Q_0}^+ \times \boldsymbol{Q_0}^+$ and for each $t \in T$ gilt $a_t \leq b_t$, where $DI(t) = (a_t, b_t)$.

The skeleton of D, the classical Petri net, we denote by $S(D)$. DI is the time function of D. a_t is called the shortest firing time of t (*sft(t)*), and b_t is called the largest firing time of t (*lft(t)*), resp.

Obviously, it is easier to study DIPN whose *sft*'s and *lft*'s are natural numbers. The net behaviour of an arbitrary DIPN can be traced back, without restriction of generality, to a similar DIPN with natural numbers as bounds for the time durations. Thus, in the following we always consider DIPN, whose time function DI is defined in $\boldsymbol{N} \times \boldsymbol{N}$, i.e. the *sft* and the *lft* of each transition are natural numbers.

An arbitrary situation in a given "classical" Petri net is completely described, if the number of tokens in each place in the net, i.e. the marking is known. This knowledge only is not enough for a DIPN. For a given marking we can obey which transitions are enabled and which transitions are disabled. But, for an enabled transition we cannot get any information about how many time is elapsed since the transition became enabled last. Therefore we need a carrier for the time information of each transition. Thus we define a time vector with as many components as transitions in the DIPN, we consider. The components of the time vector are rational numbers or a special sign - the sharp #. The sharp # means that the corresponding transition is disabled. The value zero shows that the corresponding transition is enabled. For a firing transition, the corresponding component of the time vector in a given situation is a rational number, which shows how many time is elapsed since this transition became enabled last.

For example, we consider Net 3 again: The time vector *tv := (#, 1.7, 0.5)* shows that the transition t8 is disabled, transition t9 and t10 are firing - t9 since 1.7 time units and t10 since 0.5 time units, resp.

The pair (marking, time vector) gives enough information about any situation in a given DIPN at each moment. Thus, this pair, here called state, is now the basic notion in our time-dependent Petri net.

Definition 3:

Let $D = (P, T, F, m_0, DI)$ be a DIPN, $tv: T \rightarrow \boldsymbol{Q_0}^+ \cup \{\#\}$, and m be a marking in $S(D)$. Then the pair $z := (m, tv)$ is called a **state** in D.

(Of course, not any state (m, tv) in D is a reachable one in D.)

---

2) Two transitions t_1, t_2 are in a **static conflict**, if they share preplaces, i.e. $Ft_1 \cap Ft_2 \neq \emptyset$ (comp. [Starke 90]).

The state $z_0 := (m_0, tv_0)$ with $tv_0(t) := 0$, iff $t^- \leq m_0$ (t is enabled at m_0), and $tv_0(t) := \#$, iff $t^- \not\leq m_0$ (t is disabled at m_0), is called the initial state of the DIPN *D*. For example, the initial state in Net 1 is

$$z_0 = (\underbrace{(1, 0, 0)}_{\text{initial marking}}, \underbrace{(0, \#, \#)}_{\text{initial time vector}}) .$$

The situation in a DIPN changes, when a transition fires or by time elapse. As already mentioned, we demand that each enabled transition starts firing immediately, possibly by solving dynamic conflicts. This means, our firing strategy is the “maximal step”, i.e. a maximal set of concurrent enabled transitions is firing or/and starts firing. A maximal step may be empty if the state is transient. In this case, time is elapsed only.

Definition 4:

A set *U* of transitions is said to be a **maximal step** in the state *(m, tv)*, iff:

(1) for each $t \in U$ it holds $tv(t) = 0$,

(2) when $U = \emptyset$, then $tv(t) \neq \#$ for at least one $t \in T$.

(3) $\left(U^- := \sum_{t \in U} t^- \right) \leq m$ and

(4) there does not exist a set U', $U \subset U'$ and U' satisfies (1), (2) and (3).

(Condition (2) means that the empty set is a maximal step, if there are transitions firing at that moment.)

In order to lay down formally the behaviour of DIPN’s we have to answer the question: what is (are) the possible successor(s) of a given state in a given DIPN, when a transition or maximal step of transitions is firing or/and when a certain time is elapsed? Because the firing duration of each transition is not a fixed number, but ranges between a given interval, the successor of the given state will vary, i.e. we get more than one successor - in general a huge amount of successor states.

Definition 5:

Let *U* be a maximal step in the state $z = (m, tv)$ in the DIPN $D = (P, T, F, m_0, DI)$. The **state *z* changes** into the state $z' = (m', tv')$ by firing *U* and the time duration $\tau \in \mathbf{Q_0}^+$, iff:

(1) $\forall t\,(tv(t) \geq 0 \rightarrow tv(t) + \tau \leq lft(t))$

(The elapse of the time τ is possible, because there is no firing transition in *z*, which reaches its largest firing time in less then τ time units.)

(2) $m' = m - U^- + \sum_{\forall t\,:\,tv(t)+\tau = lft(t)} t^+ + \sum_{t \in T_m} R(t) \cdot t^+$, where

$T_m = \{t \mid (t \in U \vee tv(t) > 0) \wedge sft(t) \leq tv(t) + \tau < lft\}$.

(T_m is the set of all transitions, which are firing at the state z and which can finish the firing after τ time units, because after τ time units they have reached at least their shortest firing time. These transitions are, on the one hand, the transitions from U just starting to fire (tv(t) = 0), and on the other hand the transitions, which started firing

before reaching the state z ($tv(t) > 0$),

$R : T_m \rightarrow \{0, 1\}$, and *R(t)* is a (possible integer) solution of the system of inequalities $\begin{cases} 0 \le \theta \le 1 \\ \theta \in \boldsymbol{N}^{card(T_m)} \end{cases}$

(*R* is an indicator for possible termination of a firing transition: when *R(t*)* = 1, it means that *t** is firing and it terminates the firing at this moment. For a given T_m we get $2^{card(T_m)}$ different solutions.)

$$(3)\ tv'(t) = \begin{cases} tv(t) + \tau & , \text{ iff } (t \in T_m \wedge R(t) = 0) \vee \\ & \quad (t \notin T_m \wedge tv(t) \ge 0) \\ 0 & , \text{ iff } (tv(t) = \# \wedge t^- \le m') \vee \\ & \quad (tv(t) \ge 0 \wedge R(t) = 1 \wedge t^- \le m') \\ \# & , \text{ else} \end{cases}$$

(The elapsed time will be increased for all firing transitions, which are not terminating at z'. These are all transitions from T_m, which do not terminate after τ time units (*R(t)* = 0), and all transitions, which are firing ($tv(t) \ge 0$), but cannot terminate at z', because after τ time units they did not yield their shortest firing time.
The components of the time vector tv' is 0 for all newly enabled transitions, which were not enabled at *z*, as well as for such transitions which were enabled in *z*, finished the firing after τ time units, and immediatley they become enabled at z' again.)

Using this rule of state changes, the state space could be generated for a given DIPN starting with the state $z_0 := (m_0, tv_0)$. But before implementing a new tool, we want to gather experience of the usefulness of the newly introduced net typ. So (as a short-term solution), we transform the DIPN into another well studied net typ with analysis tool(s) already available.

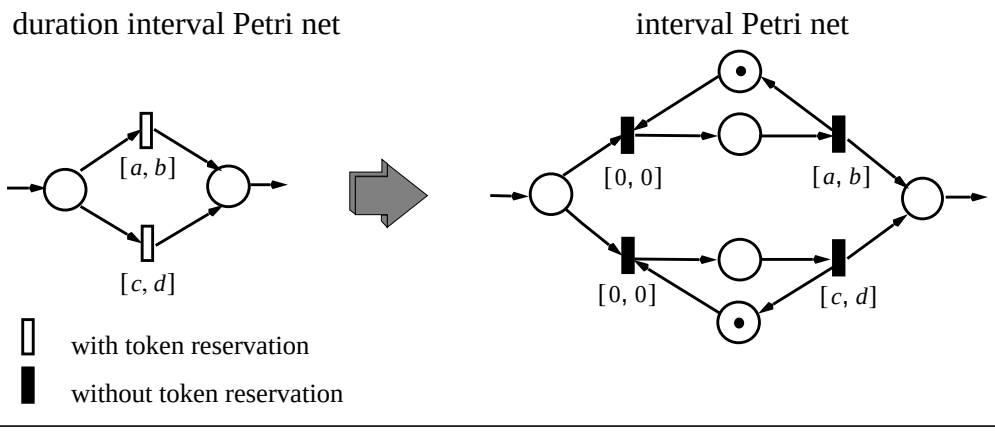
4.3 Transformation into Interval Petri nets

In order to analyze a DIPN, we translate it into an interval Petri net. We do this because:

- the interval nets are already well studied (comp. [Popova 91]),
- there exists a tool for qualitative and quantitative analysis of interval nets, INA [Starke 92],
- the transformation is easy to manage.

The transformation adds two places and one immediate transition for each transition of the original net, according the pattern given in Figure 4. It has been proven that this transformation blowing up the net structure does not enlarge the space of reachable states [Popova 96b]. Thus, we can compute a reachability graph for the given DIPN following the method to construct a reachability graph for interval nets presented in [Popova 91]. The nodes of this reachability graph are the „essential“, so-called integer states in the net, where all components of the corresponding time vector are integers or #. Obviously, this reachability graph includes only a discrete part of all the situations, which are possible in a given DIPN. But the knowledge of the net behaviour in the integer states is sufficient for knowing the whole behaviour of the net, and the computation of the integer states is much easier to manage. (For more details see [Popova 91].)

Figure 4: Transformation pattern from DIPN to interval nets.



Using this discrete reachability graph, we can analyze the DIPN in quality (boundedness, deadlock freedom, liveness, etc.) as well in quantity. Moreover, it has been shown that the time shortest and the time largest path between two reachable states resp. markings can be determined using the discrete reachability graph only [Popova 96a]. In order to be able to do it, it has been proven there that the solution of the corresponding Linear Program, which is of the following type

$$\text{(LP)} \quad \begin{cases} min\,/\,max\ t_1 + \ldots + t_n \\ b_1 \le a_{11}t_1 + \ldots + a_{1n}t_n \le c_1 \\ \quad \ldots \\ b_m \le a_{m1}t_1 + \ldots + a_{mn}t_n \le c_m \\ a_{ij} \in \{0, 1\},\ b_i \in \mathbf{N},\ c_i \in \mathbf{N} \end{cases} ,$$

is an integer. In case of computing the time largest path we disregard possible cycles. But for our applications this is not a restriction (see section 3).

5 Application Example

In [Heiner 95c] a case study is outlined aiming at Petri-net based development and step-wise validation of a medium-sized reactive system - the control software of a (really existing) production cell in a metal-processing plant [Lewerentz 95]. The case study has been designed as a basis for evaluating different kinds of software developments methods. The objective is to develop verified control software, taking into account various safety conditions and performance constraints.

The production cell considered consists of seven loosely coupled machine controllers (2 belts, 2 roboter arms, table press, crane) acting to a high degree independently of each other. The machine controllers are organized in a (closed) pipeline. Their common goal is the transport/transformation of metal plates through the established pipeline. For that purpose, neighbouring machine controllers communicate with each other according a synchronous producer/consumer pattern. There is neither a central controller of the production cell responsible for activating and deactivating the machines nor a global observer with full knowledge of the state of the production cell and of the metal plates.

Each machine follows a similar operation pattern: fetch a metal blank from the input place, process it, and deposit the plate on the output place. In order to do that, each machine performs cyclically a certain sequence of motions (of course synchronized according the states of its neighbours). Each of these motions is initiated by an actuator command (like rotate left/right, move up/down, ...) and then carried out until a sensor (like rotation angle, high, ...) yields a certain value (like load/unload position, ...). Obviously, this polling technique (compare program sketch) requires that each machine controller works fast enough in order to prevent the machines from overshooting the mark which could cause work damages or machine collisions.

```
move:
    motor!rotate left;
    while position?rotation angle /= load position
    do skip od;
    motor!stop
```

Modelling the total machine motion with Duration Interval Petri nets and assuming given time intervals for the basic mechanical motion steps, we are now able to compute the worst case of the controlling software's execution time with the (latest update of the) software package INA [Starke 92]. Due to the lack of enough space, we have to refer for more details of this case study to [Popova 96b].

Comparable situations are well-known in different kinds of hard real-time systems: if a process is caught in a livelock (i.e. blocked for longer than some critical time period), it may have the same consequences as if it were involved in a deadlock (i.e. blocked for ever).

6 Final Remarks

A new kind of time-dependent Petri net has been introduced to model and analyze concurrent systems, which require predictably timing behaviour, in a straightforward manner. In Duration Interval Petri nets, transition firing consumes time, while the firing times are given by lower and upper bounds. This net type allows to compute execution durations (of the whole software or of interesting software parts) in the best and the worst case.

Up to now, this computation is done by transforming the net models of the newly introduced net type into interval Petri nets, which are well-investigated and computer-aided analyzable by already existing tools. This transformation adds for each transition two places and one immediate transition to the original user net, resulting possibly in states, which are transient from the user's point of view. Therefore, in order to increase user friendliness of the analysis tools provided it would be helpful to analyze duration interval P nets in a direct way, without blowing up the net structure.

Further work should be done to get similar results in case of stochastic Petri nets, where the firing time is given by probability distributions (i.e. smallest and largest paths which are met with a certain probability).

Acknowledgment

We would like to thank Peter H. Starke for his immediate compliance with our wishes by extending INA with algorithms for largest paths search and by provision of an algorithm to transform duration interval Petri nets to interval Petri nets. Without his support, the practical scrutiny of our approach would not have been possible in such a short time period.

References

[Balbo 92]
Balbo, G.:
Performance Issues in Parallel Programming;
LNCS 616, 1992, pp. 1-23.

[Czichy 93]
Czichy, G.:
Design and Implementation of a Graphical Editor for Hierarchical Petri Net Models (in German);
TU Dresden und GMD/FIRST Berlin, Diploma Thesis, 6/1993.

[Donatelli 94]
Donatelli, S. et al.:
Use of GSPNs for Concurrent Software Validation in EPOCA;
Information and Software Technology 36(94)7, 443-448.

[German 94]
German, R.; Kelling C.; Zimmermann, A.; Hommel, G.:
TimeNET - A Toolkit for Evaluating Non-Markovian Stochastic Petri Nets;
Techn. Universität Berlin, Dep. of Informatics, Report 1994-19.

[Grzegorek 91]
Grzegorek, M.:
Further Development of a PDL/D Compiler (in German);
Techn. Report of Practical Studies, IIR/AdW, Berlin 1/1991.

[Heiner 92]
Heiner, M.:
Petri Net Based Software Validation - Prospects and Limitations;
ICSI Berkeley/CA, Techn. Report TR-92-022.

[Heiner 94a]
Heiner, M.; Ventre, G.; Wikarski, D.:
A Petri Net Based Methodology to Integrate Qualitative and Quantitative Analysis;
Information and Software Technology 36(94)7, 435-441.

[Heiner 94b]
Heiner, M.; Wikarski, D.:
An Approach to Petri Net Based Integration of Qualitative and Quantitative Analysis of Parallel Systems;
BTU Cottbus, Techn. Report I-09/1994.

[Heiner 95a]
Heiner, M.:
Petri Net Based Software Dependability Engineering;
Proc. RELECTRONIC ’95, Budapest, Oct. 1995, pp. 181- 186.

[Heiner 95b]
Heiner, M.:
Petri Net Based Software Dependability Engineering;
Tutorial Notes, Int. Symposium on Software Reliability Engineering, Toulouse, Oct. 1995, 101 p.

[Heiner 95c]
Heiner, M.; Deussen, P.:
Petri Net Based Qualitative Analysis - A Case Study;
BTU Cottbus, Techn. Report I-08/1995.

[Jelly 94]
Jelly, I.; Gorton, I.:
Software Engineering for Parallel Systems;
Information and Software Technology 36(94)7, 381-396.

[Kobienia 96]
Kobienia, G.:
Improvements of a Petri Net Generator for INMOS C Programs (in German);
PTB, Techn. Report of Practical Studies, Berlin 2/1996.

[Kopetz 95]
Kopetz, H.:
The Time-Triggered Approach to Real-Time System Design;
in Randell, B.; Laprie, J.-C.; Kopetz, H.; Littlewood, B. (eds.): Predictably Dependable Computing Systems, Springer 1995, pp. 53-66.

[Laprie 95]
Laprie, J.-C.:
Dependability - Its Attributes, Impairments and Means;
in Randell, B.; Laprie, J.-C.; Kopetz, H.; Littlewood, B. (eds.): Predictably Dependable Computing Systems, Springer 1995, pp. 3-24.

[Lewerentz 95]
Lewerentz, C.; Lindner, T.:
Formal Development of Reactive Systems - Case Study Production Cell;
LNCS 891, 1995.

[Merlin 74]
Merlin, P.:
A Study of the Recoverability of Communication Protocols;
Univ. of California,Computer Science Dep., PhD Thesis, Irvine, 19974.

[Popova 91]
Popova, L.:
On Time Petri Nets;
J. Information Processing and Cybernetics EIK 27(91)4, 227-244.

[Popova 96a]
Popova, L.:
An Algorithm for Computing the Shortest and the Largest Path in Time Petri Nets;
Humboldt-University zu Berlin, Informatik-Bericht, 1996 (to appear).

[Popova 96b]
Popova, L., Heiner. M.:
A Method for Analyzing Duration Interval Petri Nets;
BTU Cottbus, Techn. Report 1996 (to appear).

[Ramchandani 74]
Ramchandani, C.:
Analysis of Asynchronous Concurrrent Systems Using Petri Nets;
PhD Thesis, MIT, TR 120, Cambridge (Mass.), 1974.

[Starke 90]
Starke, P. H.:
Analysis of Petri Net Models (in German);
B. G. Teubner Stuttgart 1990.

[Starke 92]
Starke, P.:
INA - Integrated Net Analyzer; Manual;
Berlin 1992.

[Starke 95]
Starke, P.:
A Memo On Time Constraints in Petri Nets;
Humboldt-University zu Berlin, Informatik-Bericht Nr. 46, August 1995.

[Varpaaniemi 95]
Varpaaniemi, K.; Halme, J.; Hiekkanen, K.; Pyssysalo, T.:
PROD Reference Manual;
Helsinki Univ. of Technology, Digital Systems Laboratory, Series B: Techn. Report No. 13, August 1995.

[Vrchoticky 94]
Vrchoticky, A.:
The Basis for Static Execution Time Prediction;
Technical University of Vienna, PhD Thesis, 1994.

[Wikarski 95]
Wikarski, D.; Heiner, M.:
On the Application of Markovian Object Nets to Integrated Qualitative and Quantitative Software Analysis;
Fraunhofer ISST, Berlin, ISST-Berichte 29/95, Oct. 1995.