Petri Net Based Software Validation

Prospects and Limitations

Monika Heiner[1]

TR-92-022

March, 1992

1. On leave from Research Center for Innovative Computer Systems and Computer Technology (FIRST), GMD, Germany.

Petri Net Based Software Validation

Prospects and Limitations

Monika Heiner[1]
International Computer Science Institute, Berkeley, California
On leave from
GMD/FIRST[2] **at the Technical University Berlin, Germany**

Abstract:

Petri net based software validation to check the synchronization structure against some data or control flow anomalies (like unboundedness or non-liveness) has been a well-known and widely used approach for about ten years. To decrease the complexity problem and because the simpler the model, the more efficient the analysis, the validation is usually tried with the help of place transition Petri nets. However, the modelling with this Petri net class involves two important abstractions of actual software properties -- the time consumption of any action and the data dependencies among conflict decisions. Basically, this paper discusses some problems resulting from these abstractions in the models analyzed which are very often neglected and have therefore not been well understood up to now. Furthermore, discussing the pros and cons of the Petri net approach is done by offering a rough overview of the given background of dependable distributed software engineering. Suggestions for a related workstation supporting different net-based methods are outlined.

---

1. E-mail: heiner@kmx.gmd.dbp.de

2. Research Center for Innovative Computer Systems and Computer Technology

Acknowledgment:

I would like to express my thanks to the International Computer Science Institute at Berkeley for providing me an excellent working environment during my research stay. Furthermore, many experiences presented in this paper have been prepared by investigations and implementations of the students Margrit Grzegorek, TU Magdeburg, and Gunnar Czichy, TU Dresden.

Contents

1.0 Introduction 1

2.0 Overview 3

2.1 Dependability Terminology 3
2.2 Net-Based Tool Kit 10
2.3 Petri Net Framework for Context Checking 13

3.0 Modelling with Petri Nets 15

3.1 Preliminaries 15
3.2 Modelling of Sequential Processes 16
3.3 Modelling of Communication Patterns 20
3.4 Realization of Modelling 28

4.0 Modelling and Abstraction 31

4.1 Introduction 31
4.2 Abstraction of Time Consumption 33
4.2.1 Discussion 33
4.2.2 Conclusions 37
4.3 Abstraction of Data Dependencies 38
4.3.1 Discussion 38
4.3.2 Conclusions 47

5.0 Reduction of Petri Nets 48

6.0 Summary 49

6.1 Implementation Status 49
6.2 Some Hopes for the Future 49
6.3 Final Remarks 52

7.0 References 54

7.1 Software Engineering 54
7.2 Petri Nets 58
7.3 Petri Nets and Software Engineering 61

Contents of Figures

Figure 1: Taxonomy of dependable computing. 3
Figure 2: Means of software dependability procurement. 4
Figure 3: Computer-aided software validation techniques. 5
Figure 4: Horizontal and vertical validation within a layer architecture. 9
Figure 5: Overview of a net-based tool kit. 11
Figure 6: The Petri net framework. 13
Figure 7: Reduced grammar of reference language. 18
Figure 8: Petri net components for the reference language. 19
Figure 9: Basic principles of process connection. 21
Figure 10: Classification of language constructs for process communication. 22
Figure 11: Petri net components for process synchronization. 23
Figure 12: Petri net components for process addressing. 24
Figure 13: Petri net components for process waiting. 25
Figure 14: A simplified view on the influence of communication patterns on the net structure class. .. 25
Figure 15: A simple protocol in three variants /Diaz 82/. 26
Figure 16: A modified simple protocol in three variants. 27
Figure 17: Comparison of Petri nets and timed Petri nets. 33
Figure 18: Confusing combinati3on of channel and control flow conflict. 36
Figure 19: Example of a time-dependently live (timed) CSN. 36
Figure 20: The influence of communication patterns on the conflict structures. .. 37
Figure 21: State automaton of problem specification. 38
Figure 22: Supposed state automaton specification which would correspond to version 1. 39
Figure 23: Version 1 of Example 2 (E2V1). 40
Figure 24: Version 2 of Example 2 (E2V2). 41
Figure 25: Graphical solution to Example 3. 43
Figure 26: Loop structure of the theory of structured programming. 43
Figure 27: “Structured” solution to Example 3. 44
Figure 28: Favoured solution to Example 3. 45
Figure 29: Implementation status. 50
Figure 30: Implementation intentions. 51

1.0 Introduction

Distributed programs are inherently concurrent and asynchronous. Their behaviour often depends critically on the possibly unpredictable timing of their components. The large numbers of subtle interactions that can take place among the components of even a moderately-sized distributed system make it extremely difficult to evaluate certain properties of the system's behaviour or to predict its dependability in general.

Therefore, powerful techniques are needed for rigorously analyzing the possible kinds of behaviour of distributed software systems to assure that they exhibit all and at best only the properties intended. Because of the complexity of the arguments involved, developers of dependable distributed systems would greatly benefit from automated tools to aid in the analysis of the system they are going to create.

Ideally, these tools should be integrated into an environment providing tools to support all the activities required for developing and maintaining large dependable distributed software systems. Such an environment, realized by a powerful workstation, should support a diverse array of methods. In particular, it should offer tools supporting a wide variety of analysis techniques, since different techniques may have complementary strengths and weaknesses[1].

Equally important, the workstation should offer tools with strong preimplementation analysis capabilities. By providing feedback on system descriptions that arise early in the development process (i.e. requirement, design, and specification), such tools can help developers to detect errors early in that process, when their correction is cheapest.

This paper deals with one of the approaches to desirable analyzing tools, aiming basically at consistency checks of the synchronization/communication skeleton of any distributed program. These consistency checks treat properties which have to be fulfilled independently of the distributed program's special semantic and timing behaviour.

It is worth noting that the method proposed is discussed as far as possible language-independently[2]. Instead of discussing a particular language, we try to derive conclusions concerning suitable language design from the viewpoint of static analyzability. But the realization of appropriate experimental environments, making examples of practically relevant size manageable, forces implementation efforts, and any concrete implementation is always more or less strongly related to a concrete programming or specification language.

Communication protocols characterized by their special semantics and vocabulary can be comprehended as a special application area of distributed programs. So they are able to play the role of a case study standing for a wider range of applications of distributed processing. For that reason, some of the ongoing implementations are part of a general workstation for protocol engineering supporting other validation tools as well /König 88, 90/.

---

1. For a related detailed discussion see e.g. /Groz 85/, /Rudin 86/, /Sajkowski 85/, /Venkatraman 86/.
2. But up to now restricted to the family of classical imperative languages like Algol 68, Ada, CHILL or any parallel extension of originally purely sequential languages like ParFortran, ParC, etc.

In this paper, we will concentrate on the modelling aspect and related problems[1], especially with reference to the synchronization structure's static analyzability.

Furthermore, the modelling of the synchronization structure of a given distributed program by Petri nets results in an intermediate program representation from which further net-based software validation techniques are able to start (compare section 2.2):

- a systematic test approach of distributed software on communication level,
- evaluation of quantitative properties on the basis of time-based nets which are best obtained by a further semi-automatically driven compression.

Besides these validation techniques, additional net-based approaches relevant to dependability are known from literature:

- finding suitable checkpoints for recovery to establish fault-tolerant systems which avoid the domino effect,
- assessing software reliability.

According to our design objective, much attention must be paid to supporting an engineer in its task of software quality assurance. There are still a lot of problems on the methodological level. Their theoretical answers and practical solutions by implemented tools would probably have an enhancing influence on practical suitability of the Petri net approach and user's acceptance, respectively.

The paper is organized as follows.

At first, the given background of dependable distributed software engineering is briefly sketched. The introduced taxonomy allows us to define more exactly what kinds of problems we expect to become manageable by means of Petri nets and how these net-based methods must be complemented. Suggestions for a related workstation supporting different net-based methods are outlined.

Secondly, the main principles and the realization aspect of Petri net modelling are discussed in two steps: the modelling of one sequential process and the modelling of communicating sequential processes.

The modelling involves two important abstractions of actual software properties -- the time consumption of any action and the data dependencies between conflict decisions. Next, some problems and consequences resulting from these abstractions in the models analyzed are discussed in more detail. The discussion is basically done by showing examples which are expected to highlight the main points. At the same time, they are intended to give the reader an impression of what the interconnection between software program and Petri net looks like.

Finally, the current status and future intentions of ongoing tool box implementations are summarized.

1. For an overview of the whole approach applied to protocol engineering see /Heiner 89/.

2.0 Overview

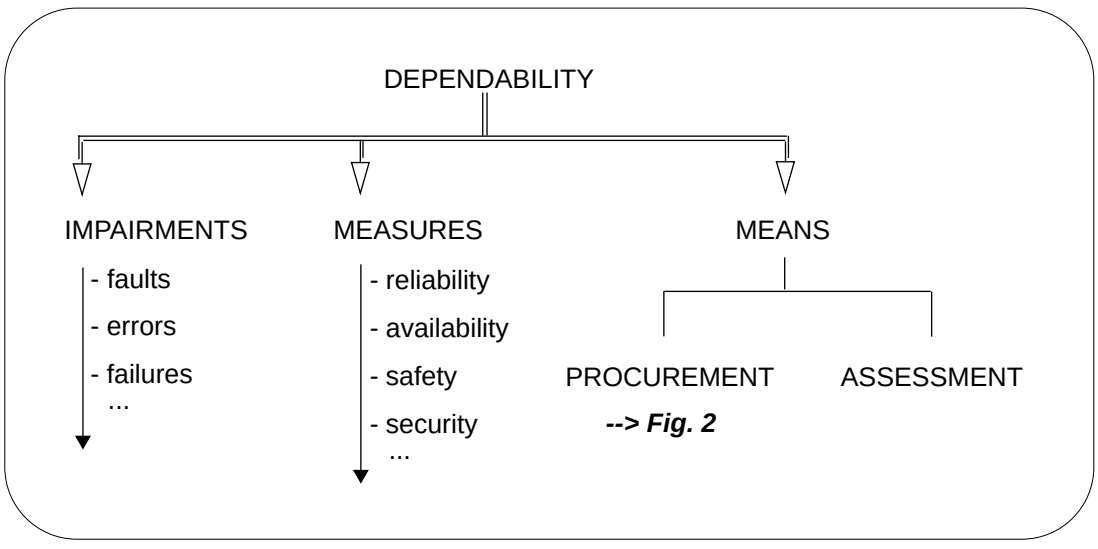
2.1 Dependability Terminology

Because of the expected expense of any validation approach based on formal methods, actual practical applications are likely to be enforced at first for computer systems with high dependability[1] demands for the services provided. For this reason, we pursue the attempt to establish a taxonomy of dependable computing undertaken within the "Reliability and Fault Tolerant Computing" scientific and technical community[2], in order to propose clear and widely acceptable definitions for some basic concepts. However, the following explanations are restricted to those notions necessary to explain the background for the approach this paper deals with. This includes a restriction to software aspects.

The concepts introduced may be gathered basically into three orthogonal main classes of attributes (see the beginning of the taxonomy shown in Figure 1):

- the **impairments** to dependability, which are undesired (unexpected) circumstances causing or resulting from undependability;
- the **measures** of dependability, enabling the service quality resulting from the impairments and the means opposing them to be appraised;
- the **means** for dependability, which are methods and general principles, perhaps implemented by tools,
  - to increase the system's dependability (**procurement**) or

Figure 1: Taxonomy of dependable computing.



1. "***Dependability*** is that property of a computer system that allows reliance to be justifiably placed on the service it delivers. The ***service*** delivered by a system is its behaviour as it is perceived by its user(s)" /Avizienis 86/.
2. See e.g. /Anderson 81/, /Avizienis 86/, /Randell 78/, /Siewiorek 91/ offering quite a few related references.

Figure 2: Means of software dependability procurement.

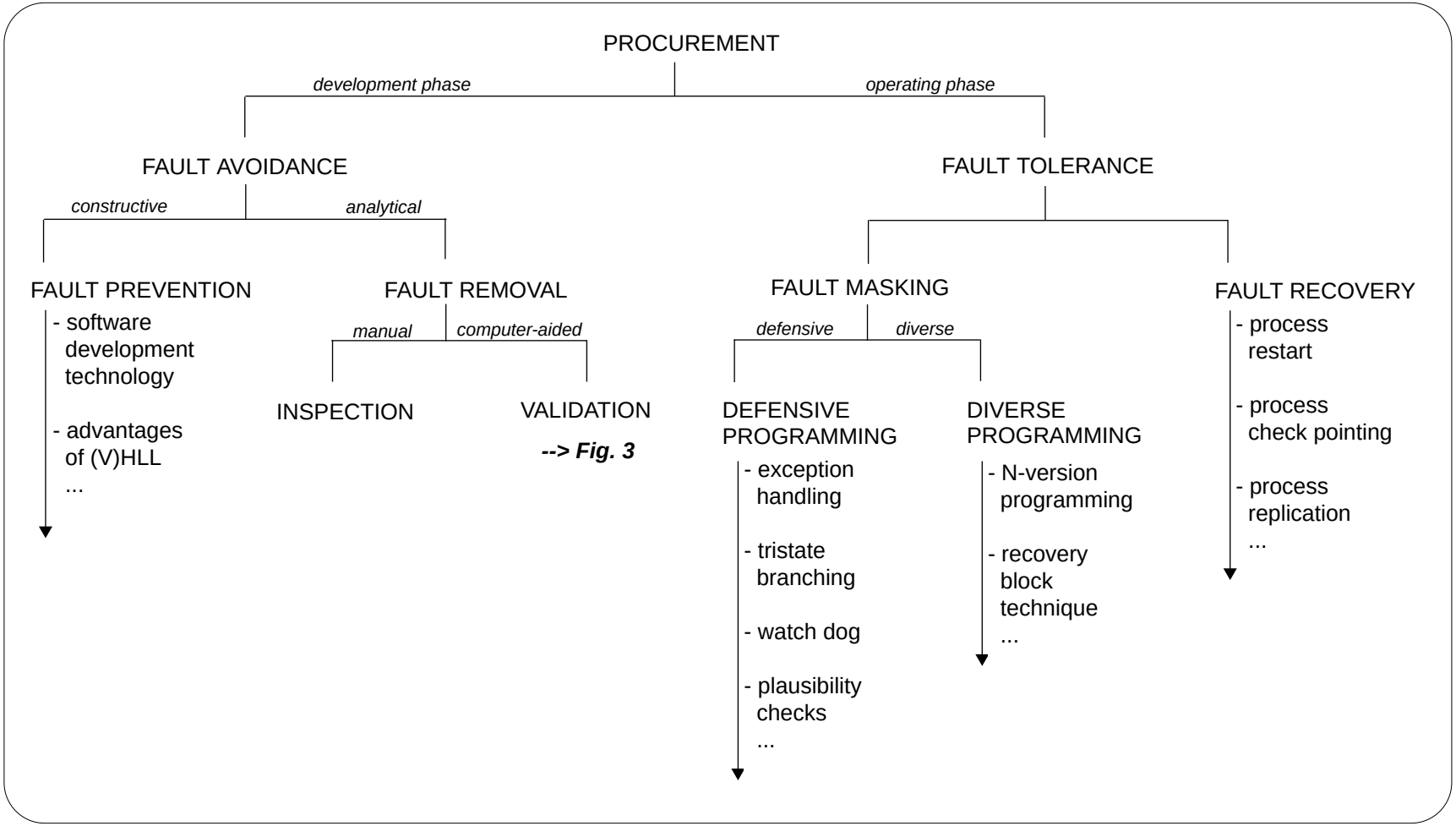
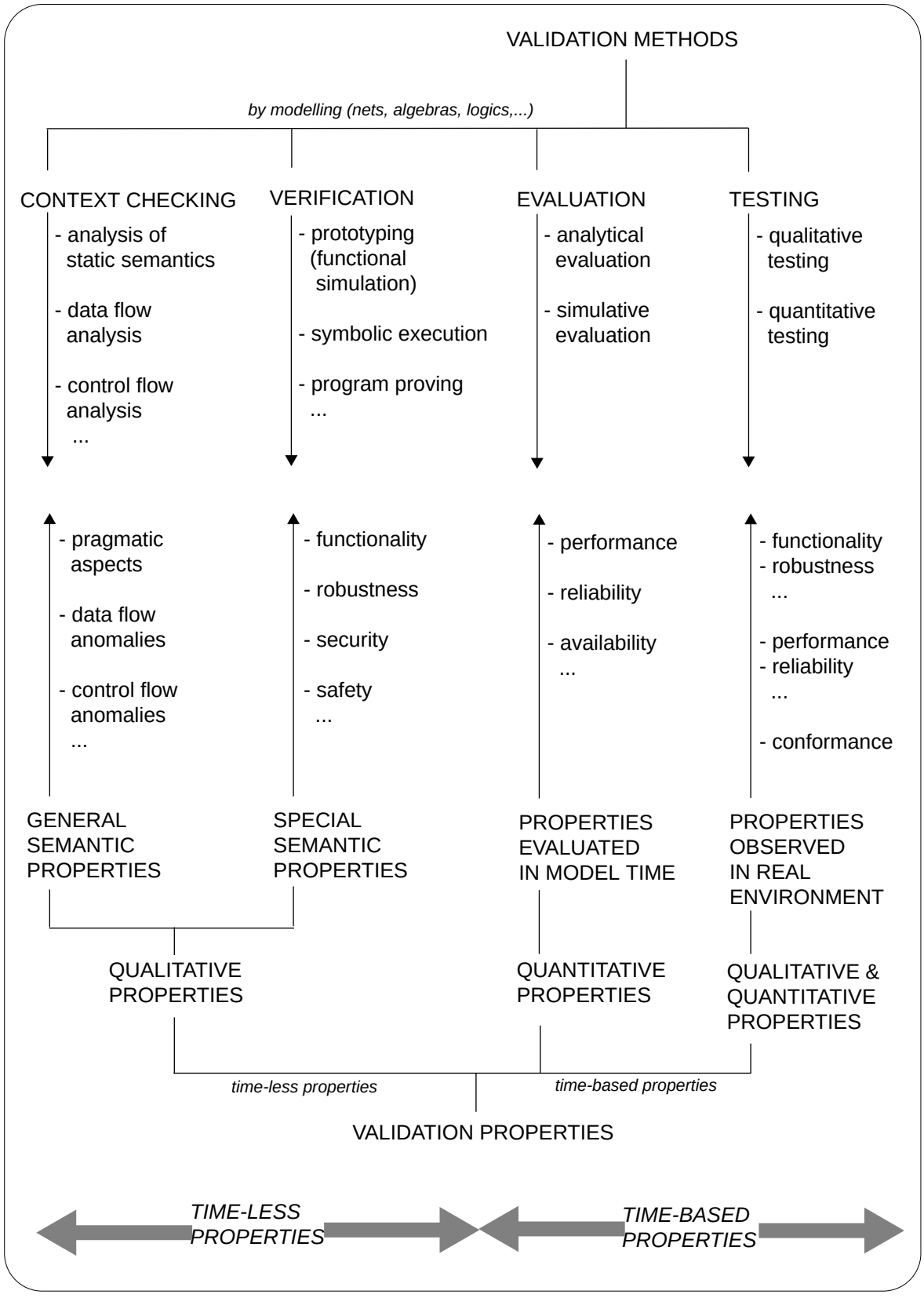


Figure 3: Computer-aided software validation techniques.



- to assess (measure) the attained degree of dependability (**assessment**).

Achieving a dependable computing system calls for the combined utilization of a set of procurement methods (Figure 2), which can be classified into:

- **fault avoidance** -- how to avoid, during the system's development phase, the occurrence of faults in the operation phase;
- **fault tolerance** -- how to tolerate faults during the operation phase by means of redundancy.

Within the fault avoidance principles, the **validation** tries to minimize the presence of faults in the operation phase by methods which are both analytical and (as far as possible) computer-aided. There is no widely accepted approach for their classification. We favour a proposal (see Figure 3), which is basically characterized by a clear separation of the validation methods (main principles) on the one side and the properties to be validated on the other side. The order in the figure from left to right is influenced by the corresponding applicability and recommended order of use during the software development process.

Because the validation methods aim at different properties, it is obvious that none of them alone is able to guarantee complete confidence in the desired total software quality. All approaches have their advantages and limitations. For this reason, they should not be viewed as competing techniques. They are, in fact, complementary methods decreasing the likelihood of software failure. That's why validation methods, more or less exhaustive checks of software products, can be comprehended as confidence-building techniques.

Basically, there are two completely different types of validation techniques:

- All **testing** techniques are based on executing the actual program system in a more or less realistic environment.
- All other validation approaches have in common that they rely on some **formal model** of the real software. These models usually reflect only certain aspects of the modelled objects in the hope that with this approach problems will become manageable. In other words, the models abstract all those properties of the software to be analyzed which are supposed not to be important for the properties under consideration.

Testing deals with any kind of qualitative and quantitative properties. There are a great number of different testing principles[1] which also partly reflect the envisaged properties. The main advantage of testing consists, in our opinion, in providing useful information about a program's actual behaviour in its intended computing environment, while the model-based validation is restricted to conclusions about the program's behaviour in a postulated environment. The most important disadvantages of testing are the following.

- In the software development process, testing can only be used very late, where the correction of any faults is already more expensive than during the specification or design phases.

1. Testing methodology has been a well liked research topic for a long time. A more detailed and comprehensive discussion of this very extensive topic would go beyond the scope of this paper. For a first overview see e.g. /Myers 87/, /Basili 87/.

- Testing is at best able to detect the presence of errors, not their absence, because an exhaustive testing is almost always impossible. That's why testing is not able to prove any properties.

The model-based validation methods try to avoid one or both of these drawbacks.

- The derivation and analysis of formal models is already applicable in the software's preimplementation phases.
- Testing is an inherently dynamic method. But with formal models, there exist usually static as well as (possibly exhaustive) dynamic methods of analysis. Perhaps the most distinguishing and significant feature of any static or exhaustive dynamic analysis is the capability of proving the absence of certain kinds of faults in a program.

An advisable combination of testing and model-based validation should lead to the consideration that it is evidently very useful to know as much as possible as surely as possible about a program before its execution.

A more detailed differentiation of the model-based methods relies on the different envisaged software properties:

- **Context checking** deals with general qualitative properties like freedom from data or control flow anomalies which must be valid in any system independent of its special semantics (for that reason, it is sometimes called general verification). These properties are generally accepted or in-house consistency conditions of the static semantics of any program structure.
- **Verification** aims at special qualitative properties like functionality or robustness which are determined by the special semantics of the system under development (to underline this fact, it is sometimes called special verification).
- **Evaluation** techniques treat quantitative properties like performance and reliability to predict the software's timing behaviour in advance, or to assess it afterwards.

While the properties the evaluation deals with are inherently time-based, both context checking and validation aim at time-less properties which should be valid time-independently. Unfortunately, that will not always be true in the case of distributed programs.

From literature it is well known that debugging of distributed programs only by means of systematical run-time tests is hardly possible. This is because some erroneous behaviour, such as synchronization errors, like total or partial system deadlocks, can be controlled by the current progress of the processes, which is generally time-dependent and non-reproducible.

Moreover, in /Gait 85, 86/ the experience is reported that some synchronization failures could not be detected by debugging (without hardware support). This phenomenon of error masking, called **probe effect**, is well known in the theory of Petri nets (see section 4.2). It emphasizes the need for another way than testing to get assurance of software quality (e.g. the impossibility of certain system states).

It is an (at least implicitly) common opinion in software development technology that the relative progress of processes with respect to each other must not have any influence on the general logical behaviour of the whole system. That's the reason why context checking

should always be done first, to prove the absence of any data or control flow anomalies in the program structure in a time-independent manner. On the basis of this confidence, a validation of the program's functionality becomes really meaningful.

There is also the aspect of practicability, recommending the separation of subquestions which are far more likely to be mastered. Therefore for the time being we will concentrate on context checking of general semantic properties in parallel program structures.

A number of description techniques for analyzing general semantic properties in distributed systems have been investigated, e.g. finite state machines /Castanet 85/, /Eckert 85/, /Sidhu 86b/, Petri nets /Diaz 82/, temporal logics /Karjoth 87/, constrained expressions /Dillon 84/, some algebraic approaches, and different hybrid ones. A formal comparison of some of these can be found in /Venkatraman 86/.

In addition to the Petri net approach, finite state machines are also favoured, for both approaches provide the formal background required for exhaustive state exploration by dynamic analysis. Beginning at an initial state, all reachable system states are generated. The resulting structure is usually represented by a so-called reachability graph of system states. This method has the great advantage of practicability, but suffers, however, from a number of shortcomings.

It generates potential behaviour in an exhaustive rather than a directed manner. This creates two problems for a software developer concerned with a specific behavioural property.

- First, these techniques usually report vast numbers of potential behaviours when applied to realistically large and complex designs. The developer is then left with the task of sorting out and interpreting the reported behaviours to determine which, if any, are relevant to the property under investigation.
- Second, the exhaustive nature of state exploration makes them particularly susceptible to the combinatorial problems inherent in analyzing distributed software systems.

Nevertheless, the power of this type of analysis for software validation objectives is illustrated in literature, see e.g. /Sidhu 86a/ where some failures in the NBS class 4 transport protocol were uncovered by using exhaustive state exploration.

Moreover, a finite state machine-based workstation of protocol engineering is sketched in /Rudin 86/. The tool kit provided comprises tools for context checking, verification, and evaluation.

While the finite state machines are completely dependent on dynamic analysis, the theory of Petri Nets also promotes, besides a lazy state evaluation[1] by reduced reachability graph construction, some static analysis methods to circumvent the problems mentioned above

- structural analysis,
- net invariants,
- property-preserving reduction.

1. Avoiding the generation of states not needed for the validation of a given property.

The reachability graph analysis -- although the most customary one -- is only one of several methods within Petri net theory.

The layered architecture often used in modelling suggests a further useful classification of validation approaches (see Figure 4):

- The **horizontal** validation investigates interconnections between processes within one layer according to process system specification.
- The **vertical** validation investigates interconnections between processes of adjacent layers according to service specification.

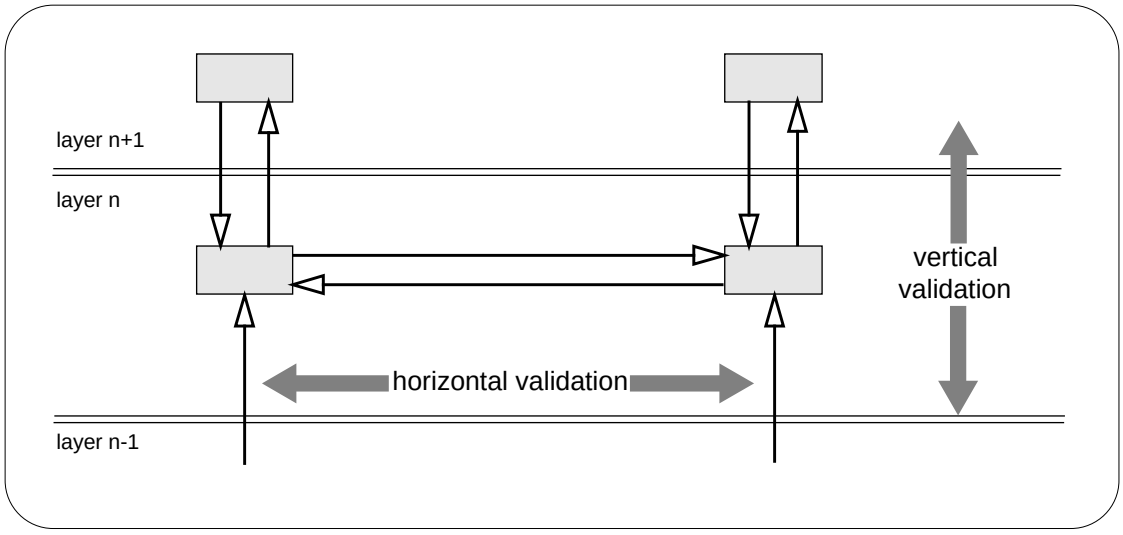
Because interactions between neighbouring layers are evidently based upon assertions about the behaviour of one layer, first the horizontal validation, and then the vertical validation have to be considered.

Petri nets are a suitable model for both validation directions. Especially important is the fact that parts of system components can already be analyzed when the essential behaviour of the environment has been modeled.

Most of the validation methods still have an experimental character and need further investigation. Some reasons must be mentioned: the absence of defining reports and manuals, similar to programming languages; the absence of support by organizations or governments.

Another important reason is the degree of acceptance of the methods by the user. The preference for, or refusal of a given tool depends essentially on the user's individual and professional experience as well as what he/she was taught. Software validation requires an adequate mathematical education.

Figure 4: Horizontal and vertical validation within a layer architecture.



Summary:

- *Validation methods can be classified according to the properties they aim at and according to the orientation of the considered process interactions within one's layer architecture.*
- *The different validation methods do not compete, but complement each other.*
- *A recommended 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.*
- *A thourough software validation is expensive and requires an adequate mathematical foundation.*

2.2 Net-Based Tool Kit

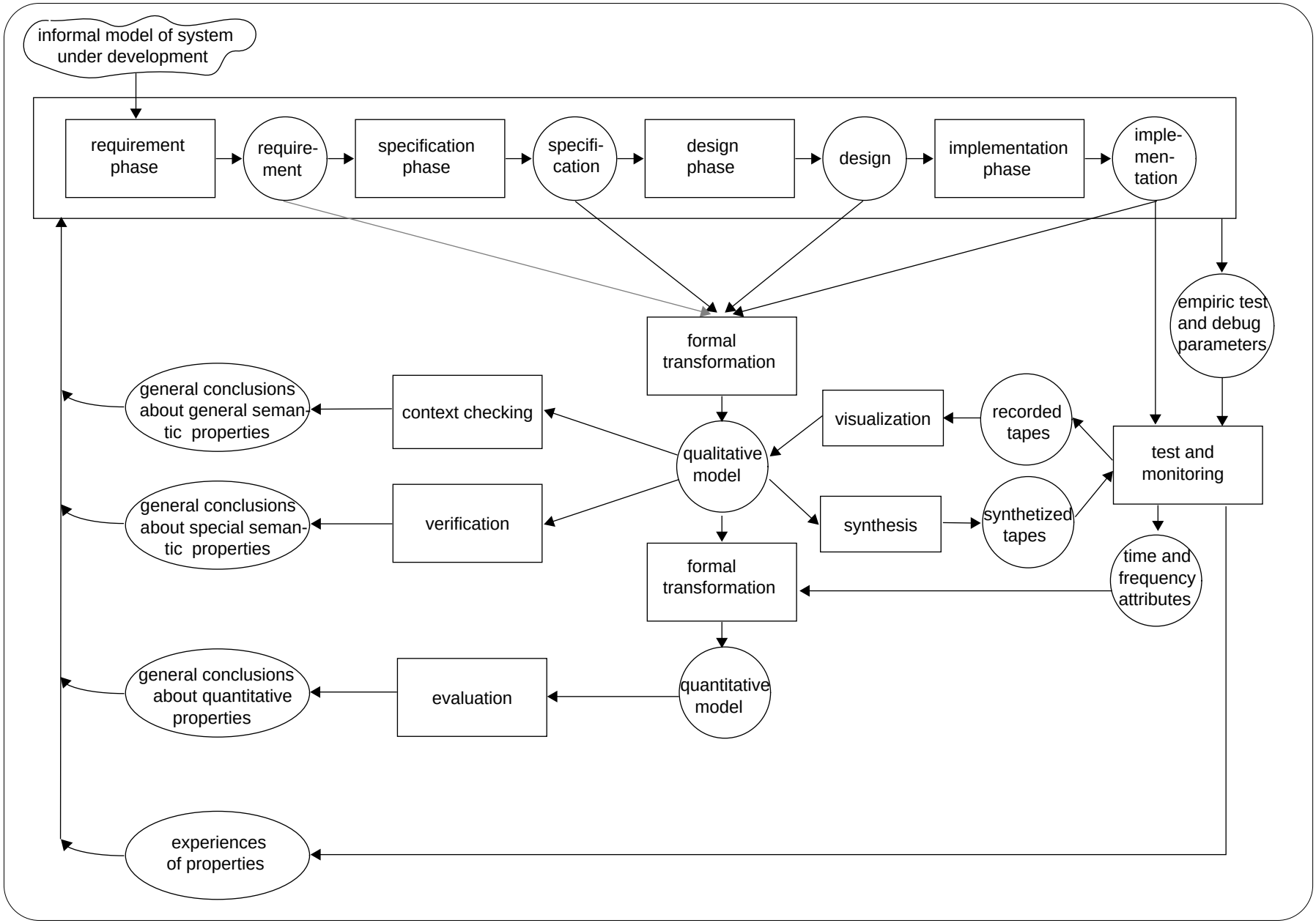
Qualitative properties (above all general properties such as mutual exclusion, deadlocks, liveness, livelock, and self-synchronization, but also functional aspects of process interactions) involve interactions among the parts of a distributed system. It is quite widely accepted that these properties are most naturally analyzed in terms of the order (and number) of events. Especially for these problems, the Petri nets offer a suitable modelling background with an already powerful theory which is still under development.

The approach of Petri net-based software validation is able to combine the advantages of high-level (specification/programming) languages with those of Petri nets theory. Figure 5 shows a proposed tool kit for net-based software validation with its components and their interconnections.

An important property of the Petri net approach is its extreme generality. It aids developers in a general way in reasoning about the behaviour of distributed systems. Because of its generality, the Petri net framework can be used with distributed systems expressed in a wide variety of specification or programming languages[1].

1. Disclaiming completeness, see e.g. for
   Ada: /Shatz 85/, /Shenker 86/,
   Algol 68: /Heiner 80/,
   INMOS-C: /Czichy 92/,
   CCS: /Goltz 88/, /Taubner 89/,
   CHILL: /Steinmetz 87a/,
   Lotos: /Barbeau 90/,
   OCCAM: /Botti 90/, /Carpenter 87, 88/, /Joosen 91/, /Steinmetz 87a, 87b/, /Tyrell 86/,
   PDL: /König 85a/, /Grzegorek 91/,
   PEARL: /Plessmann 87/,
   PL/1: /Herzog 76/,
   SDL: /Lindquist 87/, /Fischer 88/.

Figure 5: Overview of a net-based tool kit.



The semantics of a particular language are captured by a procedure for automatically deriving Petri net representations for any distributed system expressed in this high-level language (refer the Petri net generator in section 3.4). The modelling of distributed 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 assumptions, but regard general features of distributed 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 distributed 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 are able to start.

- Context checking (see section 2.3).
- Some kinds of verification by prototyping /Bruno 86/, as well as by static analyses with the help of net invariants /Lautenbach 90/ or (un-) reachability of certain system states /Ochsenschläger 88/.
- A systematic test approach of distributed software on communication level.

  A concept to integrate qualitative modelling and distributed debugging is proposed in /Dahmen 89, 91/, consisting basically in a net-controlled modification of the “instant replay” mechanism[1]. An approach with some quite similar basic ideas can be found in /Caillet 89/.
- Evaluation of quantitative properties on the basis of time-based nets which are best obtained by a further semi-automatically driven compression of the qualitative model /Wikarski 88, 91/. Time and frequency attributes necessary to generate a quantitative model should be provided by test and monitoring.

Besides these validation techniques, further net-based approaches[2] relevant to dependability are known from literature:

- finding suitable checkpoints for recovery to establish fault-tolerant systems which avoid the domino effect /Tyrell 86/, /Carpenter 88/,
- assessing software reliability /Hura 81/.

1. “Instant Replay” /Leblanc 87, 88/ is a general debugging approach making distributed program behaviour reproducible. Executing the original program, the relative order of inter-process communication is filed into a so-called process history tape for each process (record mode). Only information that identifies the messages is traced, and not the messages themselves. During the replay mode, these tapes are used to enforce exactly the same process execution order. No global time is needed. Some related topics are discussed in /Heiner 88b/ and /Dahmen89/.
2. /Pätzold 89/ contains a more general investigation of related literature about the use of Petri nets both as to design and analysis of distributed programs.

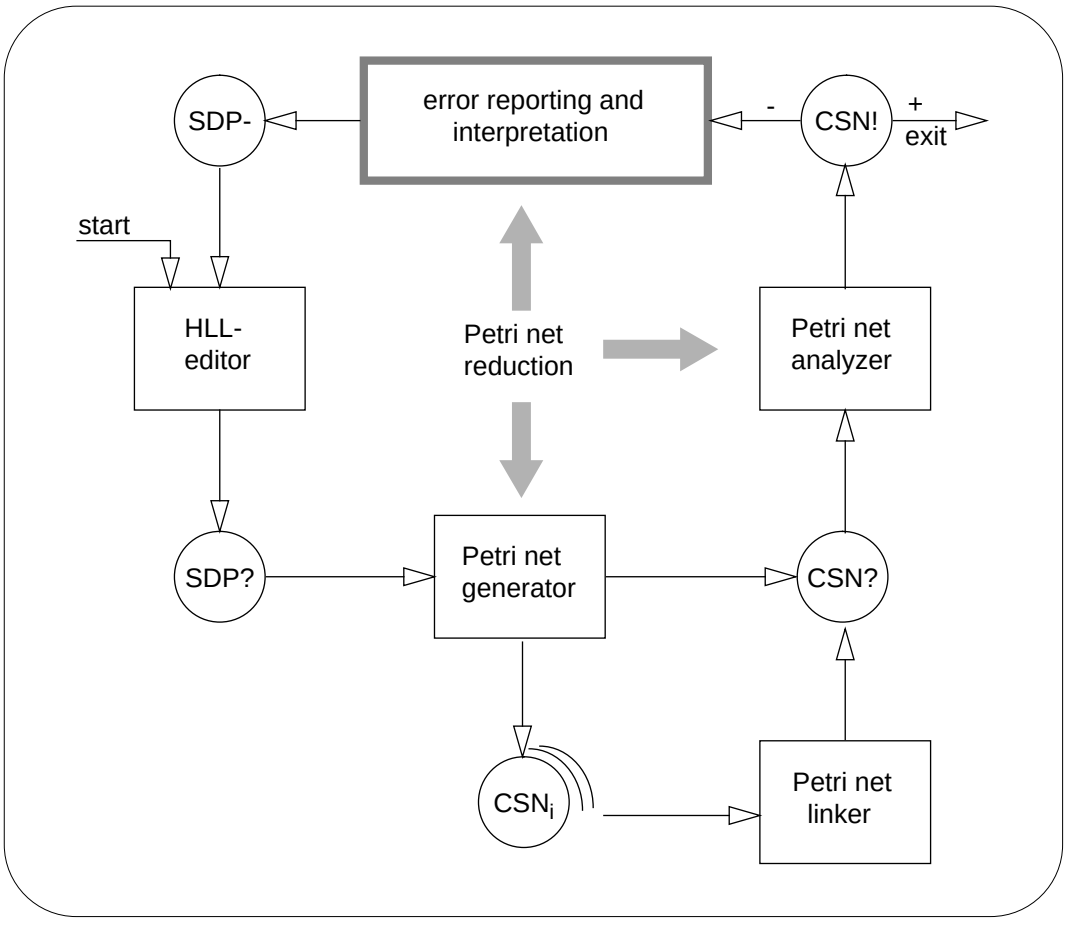
Summary:

- *Petri nets provide an adequate common basis for a general workstation that extensively supports different methods of dependable distributed software engineering.*
- *Petri nets are a suitable intermediate representation for*
  - *different languages,*
  - *different phases of software development cycle, and*
  - *different validation methods.*

2.3 Petri Net Framework for Context Checking

The components of a tool kit for net-based context checking are connected together in the so-called Petri net framework which has basically four components: the Petri net generator, linker, analyzer, and the error reporting and interpretation system. All parts are influenced by the available and exploited possibilities of Petri net reductions. We briefly describe below the approach to modelling and analyzing used in the Petri net framework (see Figure 6).

Figure 6: The Petri net framework.



The starting point for this cyclic technology in software development is an editor, which assists in creating a system of distributed processes (SDP).

To get some answers to questions concerning interesting static properties of the programmed system, it must be compiled and translated into Petri nets. This will be done with the help of the Petri net **generator**.

In the case of separate compilation of system parts, the corresponding Petri net parts must be put together by the **linker**, too.

The result of the former steps is a so-called control structure net (an interesting subclass of Petri nets) with some unknown properties (CSN?). This is the input into the Petri net analyzer. The **analyzer** is a generic name for a collection of tools implementing specific analysis techniques applicable to Petri nets. Among the tools comprising the analyzer, the available procedures for checking structural properties are of special importance. The analyzer proves the existence (or nonexistence) of some special properties of the Petri net under investigation. (The character "!" in "CSN!" stands for known properties.)

In the case of error reporting by the analyzer, now the difficult task of **error interpretation** has to be started. The difficulties result on the one hand from the need for retranslation of the analysis results in terms of Petri nets into terms of the programming source under investigation, and on the other hand from the need for decisions about the truth and gravity of the failure situation for the actual behaviour of the real system.

Basically, there are three different possibilities to interpret an error reported by the Petri net analyzer.

- The error can be traced back to missing information in the model.
- The error has been caused by the abstractions applied during modelling.
- An actual error (undesirable program behaviour) has been found.

While the former task of the retranslation of analysis results is obviously automatable, the later one of the evaluation of analysis results is still an open question and remains the intellectually demanding duty of the programming team and software quality assurance team, respectively. Ideally, the complete Petri net framework is hidden in a black box /Shatz 85/, so that the user does not need any special knowledge of mathematics. But living up to this ideal is difficult, and it seems to be an unreachable one.

3.0 Modelling with Petri Nets

3.1 Preliminaries

The understanding of this text requires only elementary knowledge of the basic notions of Petri net theory. In the following, this knowledge is assumed[1]. No special introduction into the modelling and analysis power of Petri nets is provided. Explanations for some fundamental notions are provided in footnotes.

The modelling of a distributed program's communication structure by finite Petri nets is an inherently static approach to software validation.

- The advantage of avoiding the program's execution involves the disadvantage that the modelling can exploit only those facts which are statically analyzable within the program source's context[2]. Moreover, it seems to be worth limiting the expense of a (static) data flow analysis.
- To get finite Petri nets, manageable by modelling and analysis, we are forced to restrict the modelling to programming concepts which are statically finite (i.e. independent of any execution).[3]

For these reasons, it is obvious that the more dynamic the language, the greater the gap between the model's and original program's behaviour. The static analyzability should be taken into consideration early, i.e. during the design of a new language or the selection among available ones, especially for those languages dedicated to applications with high dependability demands[4].

Besides, to keep the models as small as possible, it makes sense to abstract as many as possible of those software properties which are supposed (or at best proven) to be unimportant for the properties under consideration. Therefore we try to restrict ourselves to a time-less and (control) data-independent modelling of the communication structures[5], neglecting

- any special time assumptions about the progress of the processes relative to each other, and
- any data value dependencies among control flow branches.

Related consequences are discussed in the next chapter in more detail.

---

1. A reader not familiar with Petri nets see e.g. /Baumgarten 90/, /Reisig 85/, or /Starke 90/.
2. For an overview of resulting restrictions as to what can be analyzed statically by Petri nets see /Joosen 91/.
3. The consequences for modelling of CCS or TCSP by the restriction to finite Petri nets have been discussed on a rather formal background in /Goltz 88/, /Taubner 89/.
4. In /Hoare 73/ the opinion is emphasized that two of the most important aspects of high-level programming are simplification of program checking and improvement of error detection.
5. Like many other approaches in literature, see e.g. /Carpenter 87/, /Diaz 82/, /Herzog 76/, /Joosen 90, 91/, /Shatz 85/, /Shenker 86/, /Steinmetz 87b/.

Furthermore, it is known that the simpler the model, the more efficient the analysis. That's why we prefer using the class of ordinary (hence homogeneous) place transition nets[1], offering the richest choice of analysis possibilities -- static as well as dynamic ones.

At first, the objective is to exhaust the bounds set by Petri nets[2], and only afterwards to transfer, if necessary, to more suitable higher net classes for well-defined reasons and for well-chosen questions.

The main modelling principles of the processes within one layer (horizontal validation) are discussed in two steps,

1. the modelling of one sequential process, and then
2. the modelling of communicating sequential processes.

Up till now, we have restricted ourselves to a static set of processes -- a restriction often used in the application area of (real-time) automation techniques, which makes many problems more manageable.

There are at least three important modelling aspects which are not investigated in this text.

- In a layer system architecture, the behaviour of a process set within one layer is typically stimulated by interconnections with the layer above. For this reason, some appropriate basic **assumptions about the environment** must be made to validate the processes within one layer.

  In the case of communication protocols, all known assumptions about the behaviour of the upper layer are described in the service specification. Therefore, the modelling of environment behaviour needed for protocol validation (horizontal validation) can be extracted from the generated Petri net representation of service specification by a suitable reduction.
- Timer and **time-out mechanisms** are extensively used by communication protocols to improve fault tolerance. An adequate modelling of time-out mechanisms by Petri nets is not trivial and can best be discussed in connection with the supposed fault model of the communication medium /Grzegorek 92/.
- Especially in real-time applications, **priorities** are extensively used to force certain scheduling policies in case of concurrent resource sharing. Even though it is usually supposed that a distributed program's behaviour will not be affected by scheduling policies, it seems to be worth investigating it in more detail.

3.2 Modelling of Sequential Processes

For each process (i.e. for each protocol part in the case of communication protocols) a Petri net is constructed by using transitions to represent statements and places to represent control points.

---

1. A place transition net is called ***homogeneous***, if all (existing) edges have the same weight, and it is called ***ordinary***, if all (existing) edges have the weight 1.
2. In the following, the term "Petri net" stands shortly for ordinary place transition nets, if nothing else is noted.

More precisely, a sequence of general statements not including synchronization/communication statements (e.g. send/receive statements, wait statements) is represented by a single transition. With this modelling, the termination[1] of any synchronization-free statement sequence is assumed. The corresponding assumption in the Petri net model is the finite firing duration of any transition.

The Petri net representation of statements of the control structure, which does include synchronization/communication statements, is refined in such a way that it reflects all structurally possible threads of control. The result is called a synchronization skeleton.

The structure of the resulting synchronization skeleton may best be described in an engineer-like manner by giving the pair “reduced grammar - Petri net components” which have many strong relations among each other.

- The **reduced grammar** reflects only those context-free aspects of the language under consideration controlling the modelling. All other parts of the (full) grammar are skipped or marked as terminal symbol (by “@”). Obviously, this procedure can be applied very easily to any given language. In keeping with language independency, Figure 7 refers to an hypothetical reference language including all typical patterns of control structures[2] used in current imperative languages. A procedure mechanism has been excluded only for reasons of simplicity.
- Basically, there exist **Petri net components** for each metanotion of the reduced grammar (see Figure 8). Additionally, there are components for
  - the physical end of the process body which is used by the stop statement,
  - simple_statement@ which is required for the non-reducing operation mode of the Petri net generator.

  To support cross-referencing between Figures 7 and 8, suggestive labels have been added. These Petri net components can be combined according to the same composition rules (serialization, nesting) as are described by the rules of the grammar.

To make possible a completely free context-independent combination ability (serialization, nesting), each of these components fulfills the following design criteria:

- Each component starts with a place.
- Each component ends with a transition.

The only exception is made by the component for the physical end of the process body which must, of course, end with a place (see place P_{stop} in Figure 8).

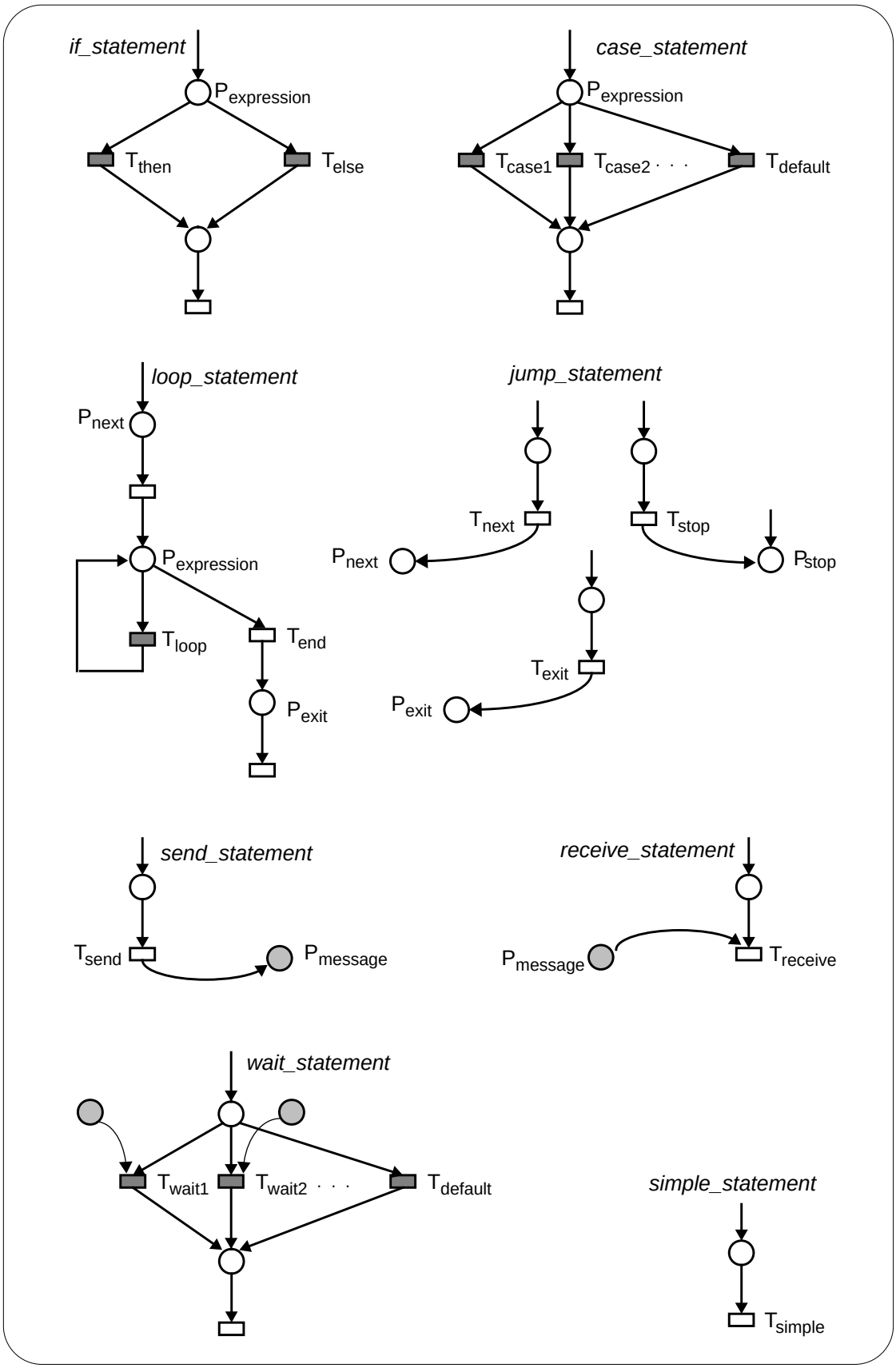
Two further conventions of graphical representation are applied, but only to enhance readability:

1. Non-termination can be interpreted as a control flow anomaly. In due of the different effects causing non-termination, it is very useful to distinguish between termination decisions within one sequential process and for a set of parallel processes.
2. The statements **next** and **exit** are only allowed within loops, compare section 4.3.

Figure 7: Reduced grammar of reference language.

```
process              ::= process_id@ ":" process
                           statement_sequence
                         #process process_id@ .

statement_sequence   ::= statement_sequence ";" statement
                       | statement .

statement            ::= if_statement
                       | case_statement
                       | loop_statement
                       | jump_statement
                       | send_statement
                       | receive_statement
                       | wait_statement
                       | simple_statement@ .

if_statement         ::= if if_expression@
                         then statement_sequence
                        [ else statement_sequence ]
                         #if .

case_statement       ::= case case_expression@ of
                           case_label@ ":" statement_sequence
                        { "|" case_label@ ":" statement_sequence }*
                        [ default ":" statement_sequence ]
                         #case .

loop_statement       ::= [ loop_ label@ ":" ]
                         loop [ loop_expression@ ]
                           statement_sequence
                         #loop [ loop_ label@ ] .

jump_statement       ::= next loop_label@
                       | exit loop_label@
                       | stop .

send_statement       ::= send message@ [ to process_id@ ] .

receive_statement    ::= receive message@ [ from process_id@] .

wait_statement       ::= wait
                           message@ [from process-id@ ] ":" statement_sequence
                        { "|" message@ [from process-id@ ] ":" statement_sequence }*
                         #wait .
```

Figure 8: Petri net components for the reference language.



- The cross-hatched transitions, and only these, can be refined according to the grammar's recursiveness. (Refinable transitions appear in the Petri net components exactly at those points where in the recursive grammar the metanotion "statement_sequence" appears.)
- Synchronization places (introduced in the next section) are highlighted by striping-hatching.

It is obvious that the modelling of one process always results in a state machine (SM) if we ignore any connections to synchronization places.

Within a state machine[1], it is structurally impossible for an existing token to disappear or to multiply. So, each sequential process (state machine), once started, has the remarkable property of containing exactly one token in only one place. Therefore, all places within a single-threaded state machine are a priori safe[2].

This single token can be interpreted as a program counter. If the state machine is strongly connected[3], this token circulates forever through the whole program, representing the infinite control flow of the modelled program and making the Petri net model live.

Structured sequential processes consisting of an infinite loop at the outermost nesting level and without any jumps are, per construction, strongly connected. This illustrates how compact control structures and restricted use of goto support simple Petri net structures, which can decrease the amount of analysis substantially.

3.3 Modelling of Communication Patterns

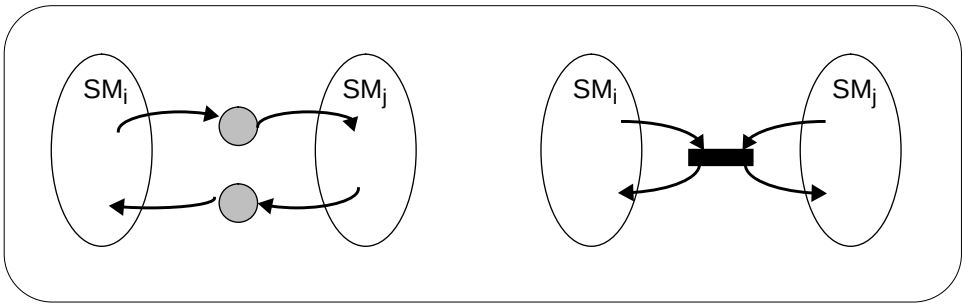
In the previous section, we developed a fixed set of (isolated) sequential processes each modelled by a state machine. If these processes communicate with each other, the state machines have to be composed according to the used communication patterns. The Petri net model of the communication patterns depend on properties of the language constructs for process communication. Figure 10 gives an overview of these properties which we suppose to be relevant for our purposes.

Basically, the connection between state machines can be realized via places or via transitions (see Figures 9 and 11).

- Obviously, the connection via transitions damages the state machine structure drastically, because it is an inherent property of the communication transitions (modelling the rendezvous synchronization avoiding any auxiliary communication medium) that they have more than one preplace and postplace (each one of a different process).
- In case of all other than rendezvous synchronization features, the state machines are far more loosely connected via synchronization places (modelling the communication medium).

1. In a ***state machine***, each transition has, per definition, exactly one preplace and one postplace.
2. ***Safe*** stands shortly for 1-bounded.
3. A directed graph is called ***strongly connected*** if there exists a directed path between each pair of nodes.

Figure 9: Basic principles of process connection.



We get a net class called communicating state machines[1] (CSM). How far the structure of communicating state machines is from a state machine structure depends again on the process patterns used to deal with the addressing and waiting scheme (see Figure 14).

In the modelling of the communication medium, the synchronization places also reflect some of their properties.

- The synchronization places can have a certain capacity as in the case of the semi-asynchronous synchronization type. Then they are a priori k-bounded to the given capacity k of the communication medium. The Petri net structures we get are locally conservative.[2]
- The synchronization places can be 1-bounded (safe), guaranteed by the construction principle, as in the case of the synchronous or remote invocation synchronization type. The Petri net structures we get are globally conservative.[3]
- Otherwise (asynchronous synchronization type), it is an interesting question whether or not an upper bound of the token numbers can be found by analyzing the net behaviour.

A customized Petri net analysis tool for communicating state machines could be made to exploit the a priori knowledge stemming from the applied construction principles in order to minimize the representation and implementation of reachable markings.[4] At the same time, this would result in a more compressed and interpretation-oriented representation which might have a positive influence on the user's acceptance.

---

1. Similar net classes can be found in literature. E.g. in /Reisig 82/ the "buffer synchronized state machines" are introduced, in /Kuse 86/ the "stream-connected concurrent processes", in /Fengler 91/ the "systems of coupled state machines", and in /Peng 91/ the "communicating finite state machines".
2. A Petri net is called ***locally conservative*** if there exists a constant sum of tokens for all reachable markings. No transition affecting the total number of tokens in the net is allowed.
3. A Petri net is called ***globally conservative*** if there exists a constant weighted sum of tokens for all reachable markings. Obviously, such a weight vector can be constructed very easily for the corresponding subset of CSM by setting the weight
   - for each sequential place (non-synchronization place) to 1, and
   - for each synchronization place to -1.
4. Obviously, each reachable marking can be represented uniquely by the current control point (marked place) of each process and the current marking of all synchronization places.

Figure 10: Classification of language constructs for process communication.

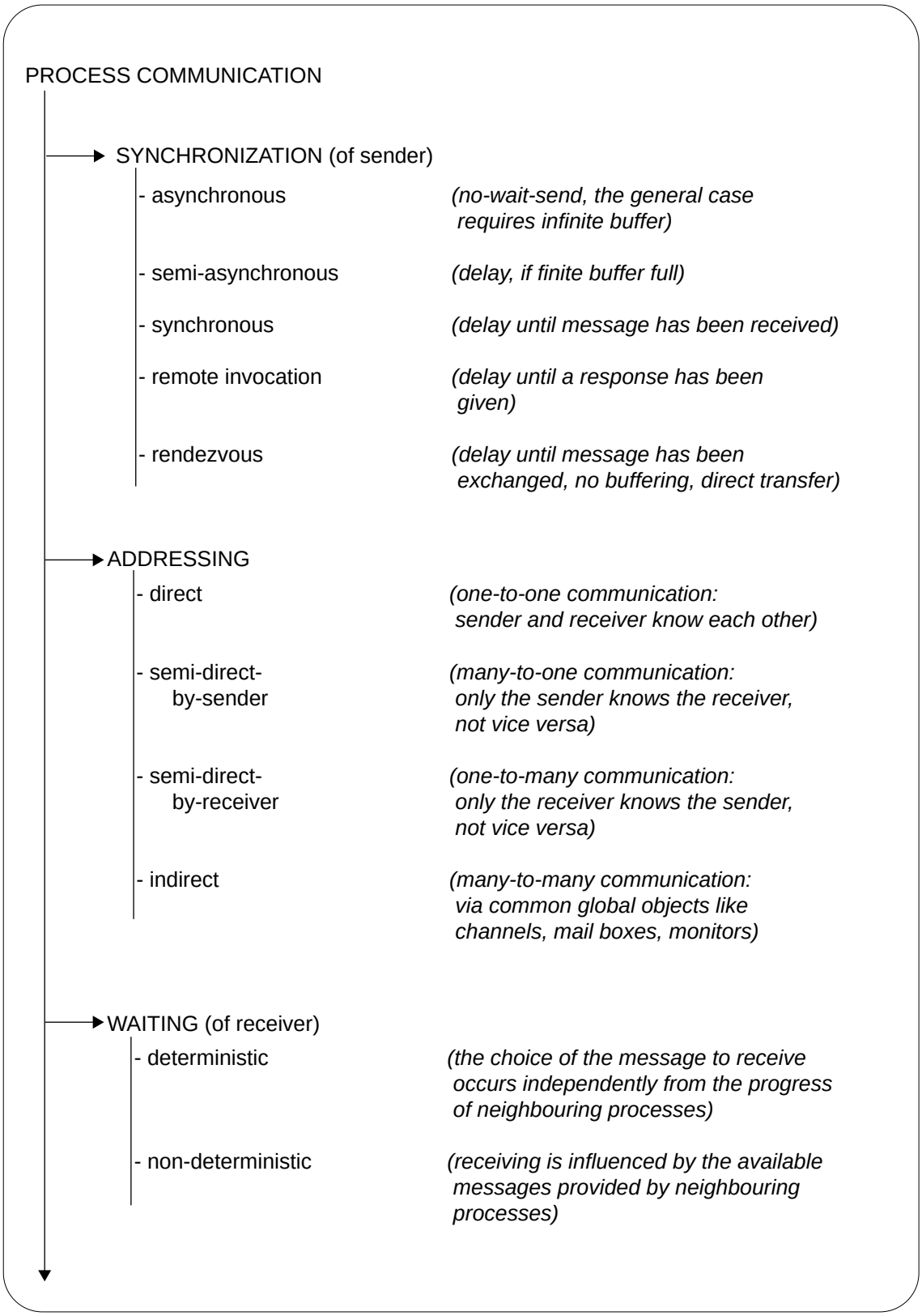


Figure 11: Petri net components for process synchronization.

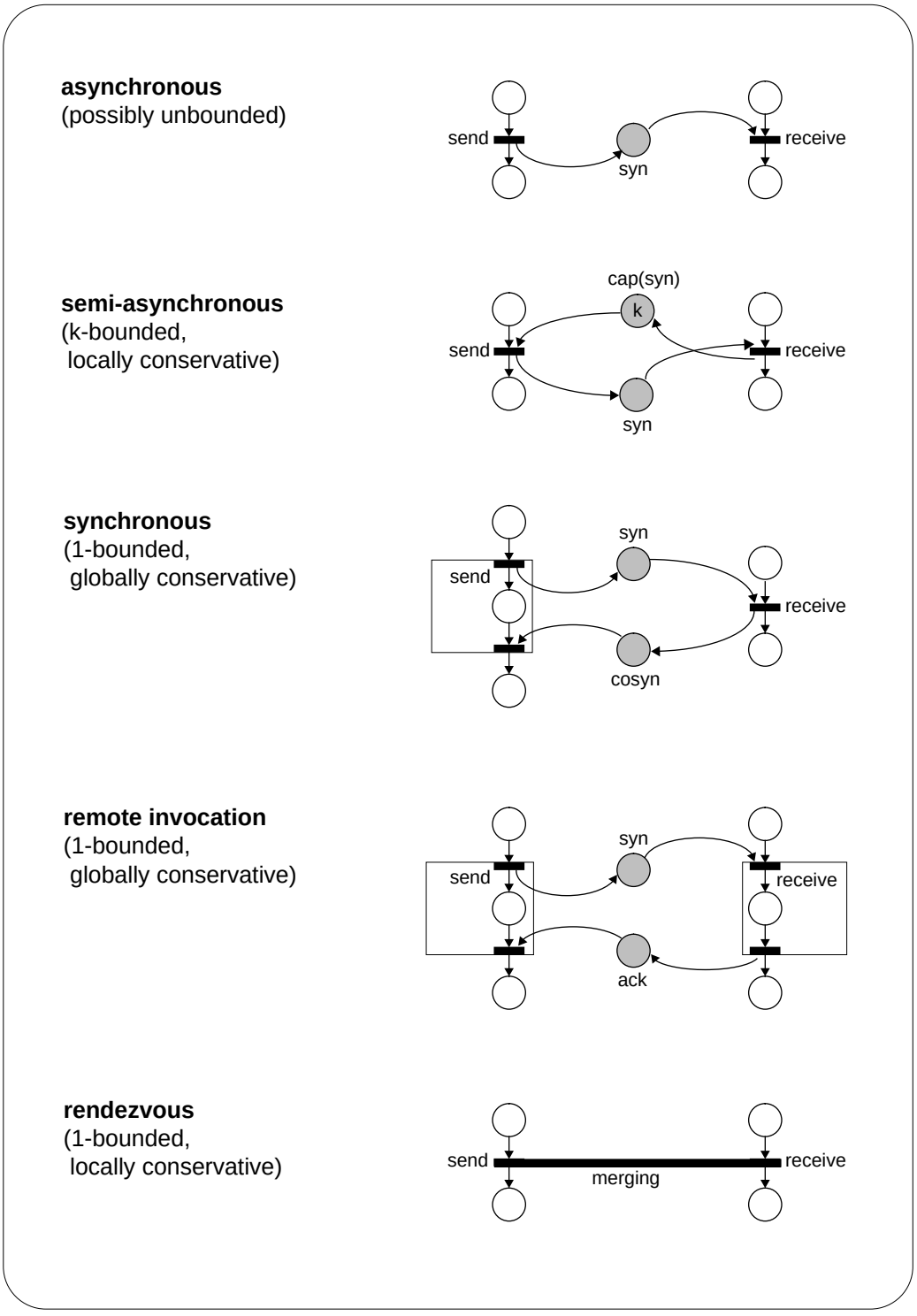
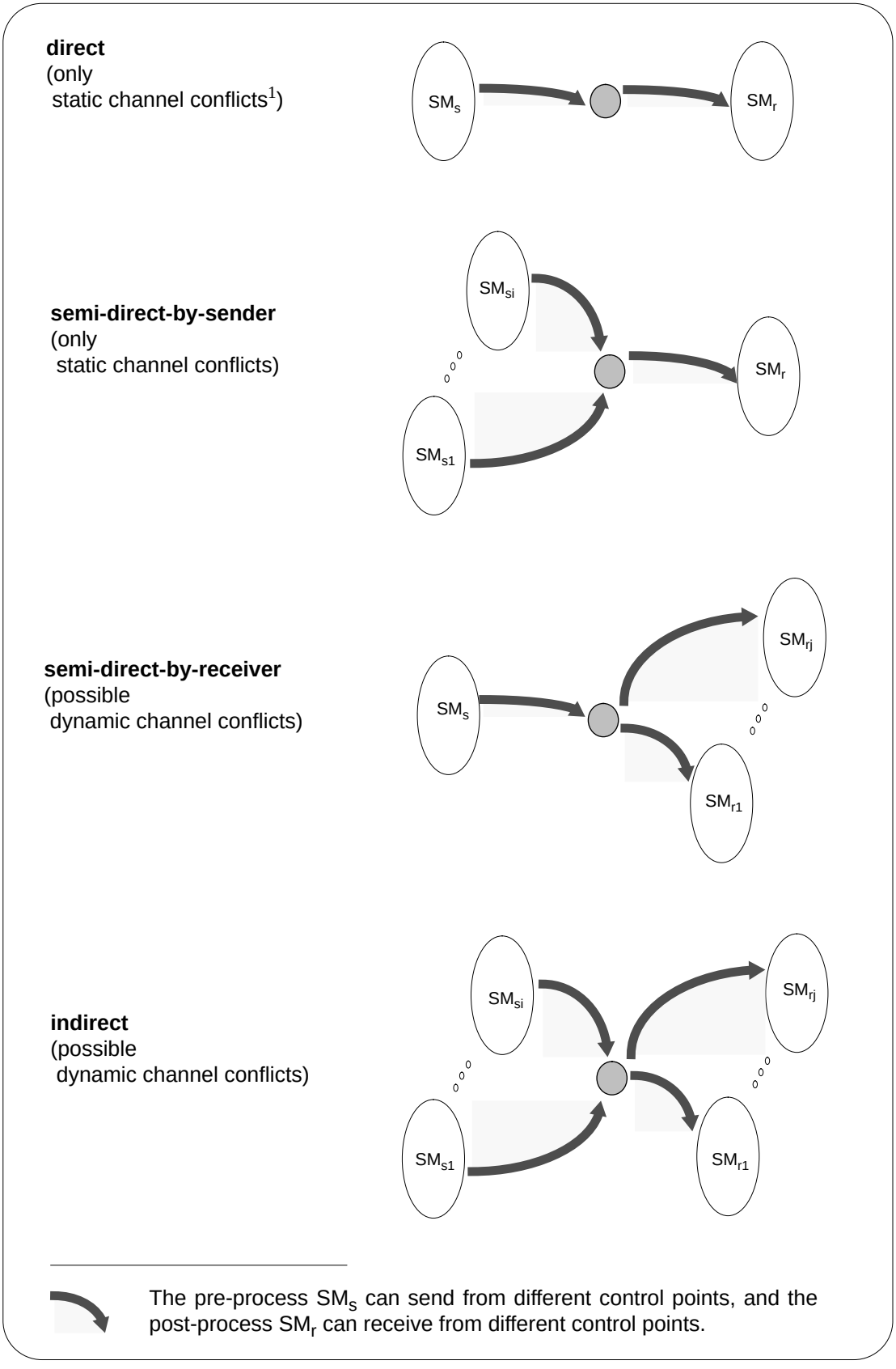
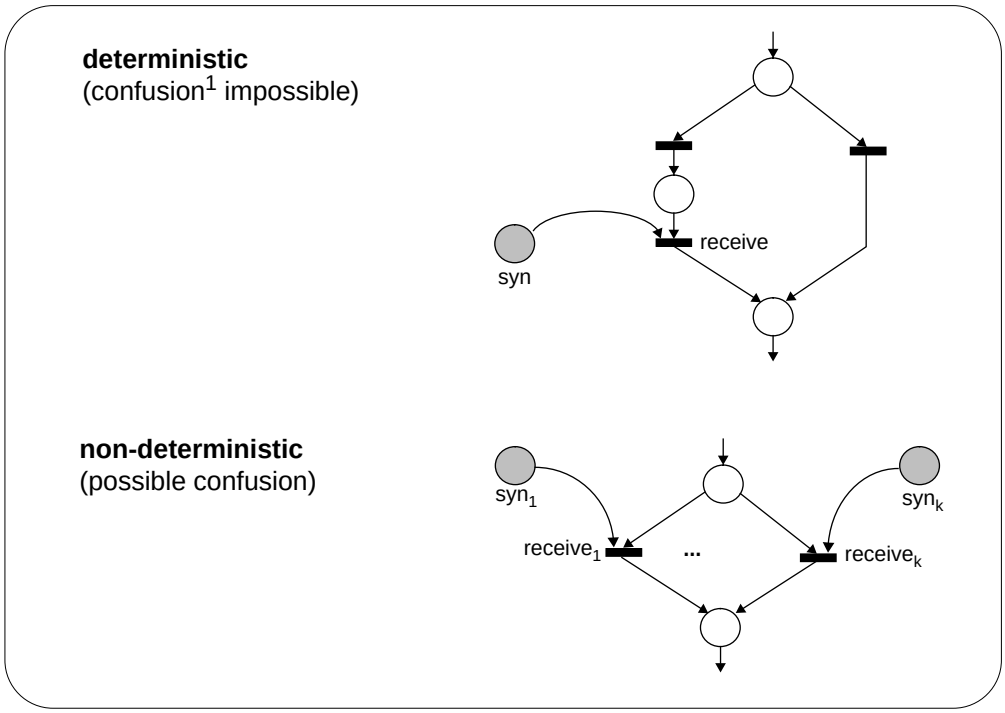


Figure 12: Petri net components for process addressing.



1. Conflicts are discussed in section 4.2.

Figure 13: Petri net components for process waiting.



1. Confusion is discussed in section 4.2.

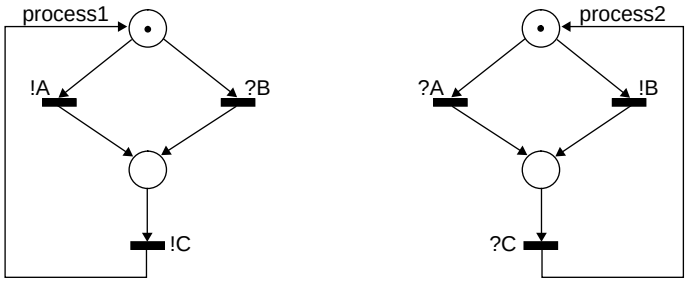
Figure 14: A simplified view[1] of the influence of communication patterns on the net structure class.

\ addressing waiting\	direct / semi-direct-by-sender	indirect / semi-direct-by-receiver
deterministic	EFC[2]	ES[3]
non-deterministic	ES	CSM

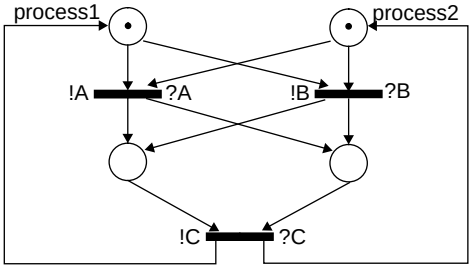
1. Provided, pre- and postprocesses do not access the same communication object from different control points.
2. A net is called ***Extended Free Choice*** net (EFC) if for all pairs of places, which do have a common posttransition, the posttransition sets are equal.
3. A net is called ***Extended Simple*** net (ES) if for all pairs of places, which do have a common posttransition, the posttransition set of the one place is a subset of the posttransition set of the other place. In /Best 86/, extended simple nets are called asymmetric choice nets.

15: A simple protocol in three variants /Diaz 82/.

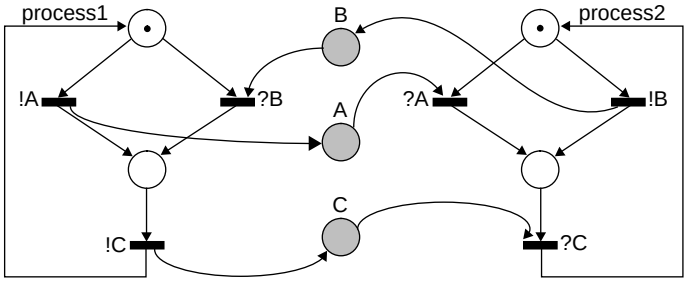
(1)
given
original
process
patterns



(1a)
rendezvous:
EFC,
safe,
live



(1b)
asynchronous:
ES,
not bounded,
live



(1c)
remote invocation:
ES,
bounded,
not live

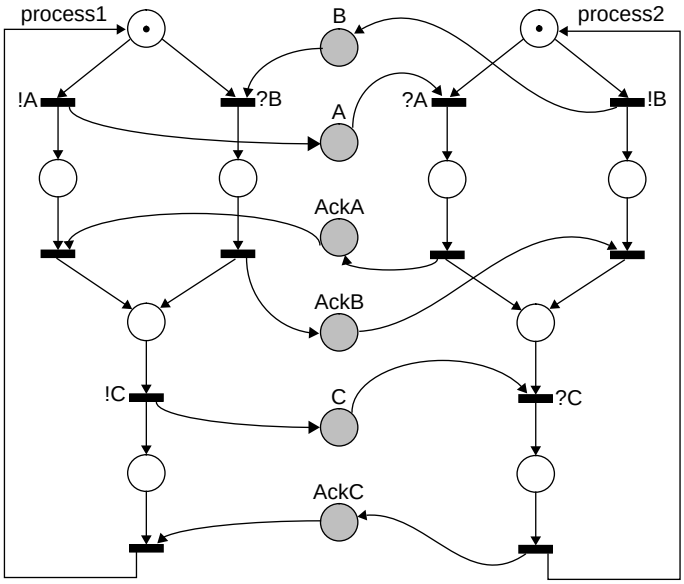
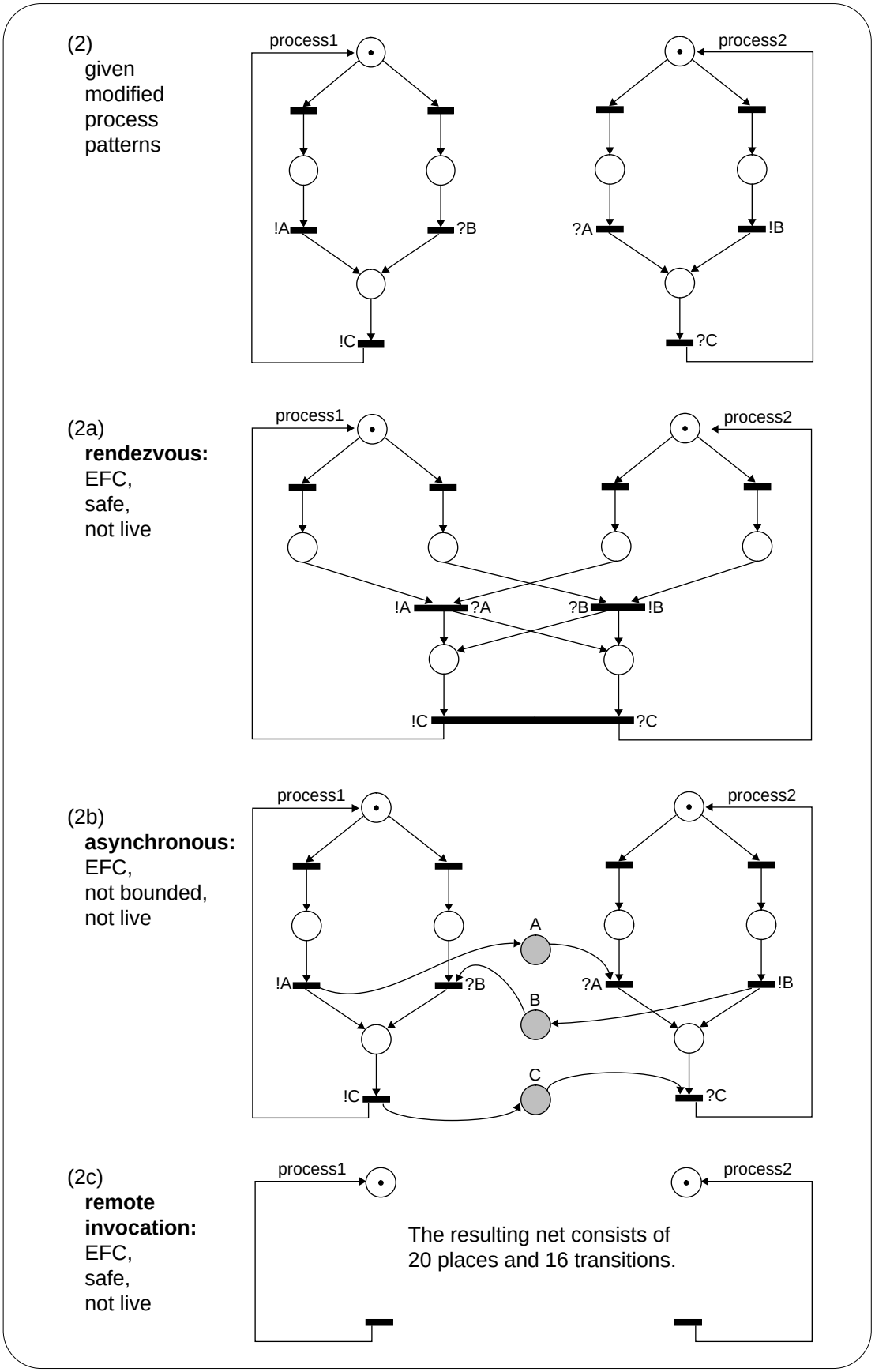


Figure 16: A modified simple protocol in three variants.



A careful design of the Petri net components is an important prerequisite for getting usable analysis results. The influence of modelling on analysis has been discussed in /Diaz 82/ with the help of a quite simple connection protocol. The given verbal problem specification is the following.

Example 1:

There are two communicating processes. Process1 or process2 can open a connection by sending messages A or B. Only process1 can disconnect by sending message C to process2.

Applying three different communication patterns to the (seemingly) same problem, it is illustrated in /Diaz 82/ that the thoughtless choice of interaction mechanism may lead to completely different analysis results (see Figure 15). But in fact, the Petri net models describe really different realizations of the original problem.

(a) In the case of rendezvous synchronization, the processes make an agreement in advance whether message A or B should open a connection.

(b) In the case of asynchronous synchronization, each process may concurrently open a connection without regarding its neighboured process.

(c) In the case of remote invocation, only one process is allowed to open a connection because of the requested acknowledgment. But if the other process does not hold this (implicit) rule, then the system will run into a classical deadlock situation.

The different analysis results disappear if the underlaying process patterns describe the same basic protocol behaviour, e.g. as demonstrated in Figure 16.

The crucial point is that the communication patterns represent an abstract view of the lower level within an hierarchical communication system. In a bottom-up design, the models of the processes' interaction mechanisms have to be derived from the behaviour of the lower level. In our case, the lower level is the language-oriented operating system layer realizing the process management and interaction features.

3.4 Realization of Modelling

There are some evident reasons for an automated mapping of the relevant program structure onto a Petri net model, especially one of the expected size of Petri net models. Besides, the model validation can be concentrated on the validation of generation procedure.

The procedure applied to generate Petri nets automatically depends on an essential general precondition the language used has to fulfill.

- There are no implicit interactions (synchronization, communication) between distributed processes made by sharing global variables. (If the language definition does not already enforce this principle, then it must be supplemented by in-house project standards.)

- Instead, all process interactions are realized explicitly by dedicated language means. The scanning of the relevant communication primitives requires special syntactical units which are treated as basic elements like **message**, **send**, **receive**, **wait**.

If these prerequisites are fulfilled by a given language, the Petri net generator needs appropriate Petri net components to map the relevant control flow for all communication primitives and control structures contained in the language (see Figure 8).

These Petri net components are composed according to the result of the parser -- the syntax tree. This means that within a general scheme of a compiler structure the Petri net generator appears as a particular semantic synthesis /Heiner 80/.

Obviously, the effort required to write a Petri net generator can be essentially reduced by using the front-end of a conventional compiler, if a sufficiently good description of the internal interface between compiler's front-end and back-end is available.

Besides the basic functionality to generate a Petri net, the Petri net generators in use /Grzegorek 91/, /Czichy 92/[1] supply further information on layout and program complexity as well as operation options.

(a) Layout:

During the Petri net generation, all nodes (places, transitions) are supplemented by x- and y-coordinates which allow an automatic layout of the generated Petri net afterwards. But because the Petri net generation is done separately for each process, this automatic layout generation covers only one process. Graphical process composition has to be done (up to now) manually by means of a suitable graphic editor.

(b) Program Complexity[2]:

We favour a complexity measure called Number of Acyclic Paths (NAP) introduced in /Nejmeh 88/. In characterizing it briefly, the following aspects are worth mentioning.

- It is a finite measure of a sequential program's execution possibilities.
  - It yields the number of structurally possible paths by neglecting any data dependencies between decision points. So, it overstates the number of realizable paths in the case of certain program structures.

---

1. They are based on experience gathered in /Heiner 80/ and /Wehrsdorfer 89/.

2. Program complexity can be interpreted as the difficulty experienced programmers have in understanding a program to test or modify it. Research on software metrics received much attention in the mid 1970s (see e.g. /McCabe 76/, /Rodriguez 87/, /Shepperd 88/).

   Since this time, the dream of finding objective measures of software products has created a wide range of complexity metrics proposing ways of identifying "complex" code with sufficient confidence. All these complexity measures have in common that they are intended to

   - predict software costs,
   - evaluate programming effort,
   - estimate program understanding,
   - calculate testability.

   For a more detailed discussion see /Heiner 88b/.

- It yields the number of “linearly independent” paths through a program by counting a single iteration of each loop.

  Because of these restrictions, the NAP complexity reflects exactly the number of paths in the Petri net model.

  This complexity metric is most closely related to some maximum strength, but finite testing efforts. The decision to use this complexity measure is based on the assumption that software with more execution paths is more difficult to understand and test than software with fewer execution paths and therefore exhibits an higher complexity.

- It can be simply calculated during compilation or afterwards by means of an attributed abstract syntax tree. The NAP complexity is additive for nesting of statements, and is multiplicative for consecutive statements.
- It supports software complexity assessment at the function, subunit or unit level by being applicable on any abstraction level of algorithm structure.

(c) Options

The Petri net generators’ operation mode can be controlled by different options providing additional helpful features (defaults are underlined).

- reducing/non-reducing mode:

  Because the graphical impression of the program structure is sometimes also useful for a separate sequential program, the reduction to the synchronization skeleton can be switched off causing any program statement to be modelled by a transition.

- comment/no-comment mode:

  In the case of comment mode, information supporting cross-referencing to the source text is provided (identifiers, source line numbers). This may be helpful in retranslating Petri net-based analysis results back to source text level.

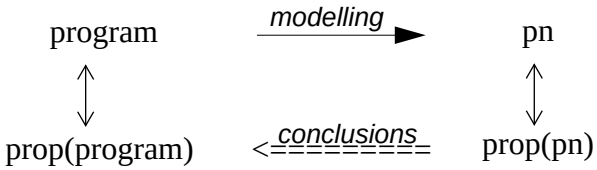
Summary:

- *The costs of static analysis and the range of results are to a high degree dependent on the used control structures and communication concepts.*
- *A restriction to a well-chosen combination of language means can possibly simplify the corresponding class of generated Petri net structures to subclasses with remarkable properties the analysis can make use of.*
- *An offered or enforced modularization supports a decomposition and reduction of the analysis.*

4.0 Modelling and Abstraction

4.1 Introduction

The approach of Petri net based software validation is characterized essentially by modelling the synchronization structures of parallel processes by Petri nets. This is done in order to conclude from the properties of the Petri net the properties of the actual behaviour of the parallel processes in a real environment.



During the modelling two important abstractions happen -- the information about the time consumption of any action (sequential parts or synchronization/communication statements) and the information concerning data dependencies between conflict decisions are neglected.

- With respect to the abstraction of time consumption, the conclusion drawn from the properties of a Petri net concerning the properties of the corresponding program class is similar to that drawn from the properties of a given Petri net concerning the properties of any correspondingly timed Petri net (which has the same structure, but each transition possesses a time constant giving the duration of firing).

  But it is known from Petri net theory that there is generally no justification for this deduction, because time constraints can restrict the reachability set.

- Similar conditions hold relating to the abstraction of data dependencies between conflict decisions. E.g., if two consecutive IF-statements decide their branching by the same Boolean expression, then there exist four structurally possible execution paths, but only two data-dependent possible paths (provided no variable of the Boolean expression will be changed in the meantime).

To summarize the consequence, static analysis of program structures by means of Petri nets considers, in general, more execution paths of the whole system than are actually possible if all (time and data) dependencies between conflict decisions are taken into account**:**

$$\mathbf{R}_{\text{program}} \subseteq \mathbf{R}_{\text{pn}} .$$

Because of this, two different types of properties have to be distinguished:

EX-property:

Properties which are fulfilled if there exists at least one system execution (path in the reachability graph) with some special condition are called EX-properties. If an EX-

property is not fulfilled in the reachability set of the underlying place transition Petri net, then it is impossible to fulfill it for any subset of this reachability set.

not prop (pn) ==> **not** prop (program)

Typical examples:

- *boundedness:* BOUND (pn) ==> BOUND (program)
  If there is no unbounded state reachable in the reachability graph of the underlying place transition Petri net, then no unbounded state is reachable for any subset of this reachability graph.
- *freedom of deadlock:* **not** DEAD (pn) ==> **not** DEAD (program)
  If there is no dead state reachable in the reachability graph of the underlying place transition Petri net, then no dead state is reachable for any subset of this reachability graph.

ALL-property:

If the fulfillment of a property depends on all (or a certain set of) possible system executions (paths in the reachability graph) with some special condition, then this property is called ALL-property. If an ALL-property is fulfilled in the reachability set of the underlying place transition Petri net, then this does not apply generally for any arbitrary subset of this reachability set, because it is possible that the subset can cut such execution paths, which are essential for the fulfillment of the property under consideration. But the reverse deduction applies. If all paths needed for a given property are already included in some set, then they are still included in any superset.

prop (pn) <== prop (program)

Typical example:

- *liveness:* LIVE (pn) <== LIVE (program)
  If the underlying place transition Petri net is live (each transition has the chance to fire infinitely often), then this must not be true for any arbitrary subset of its reachability graph.

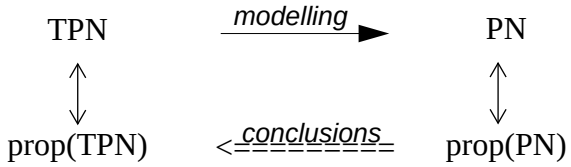
Consequently, if an EX-property is fulfilled in the Petri net model (e.g. there exists a deadlock state), then it has to be shown that this state is actually realizable in the modeled object. But, if an ALL-property is fulfilled in the Petri net model (e.g. all transitions are live), then this means, for the time being, nothing for the modeled object.

In the following, we are at first going to make the problems more evident and secondly, to discuss some possible consequences resulting from both of these abstractions in the models analyzed.

4.2 Abstraction of Time Consumption

4.2.1 Discussion

With respect to the abstraction of time consumption, the conclusion drawn from the properties of a Petri net concerning the properties of the corresponding program class is similar to that drawn from the properties of a given Petri net PN concerning the properties of any correspondingly timed Petri net TPN (which has the same structure, but each transition possesses a time constant giving the duration of firing).



Time constraints can result in a restriction of the reachability set because of the different construction rules (see the comparison given in Figure 17). To be exact, due to the time restriction, the actual possible firing sequences in a given timed Petri net are a subset of the reachability set associated with the time-less Petri net (at a given initial marking m_0).

$\mathbf{R}_{TPN(m_0)} \subseteq \mathbf{R}_{PN(m_0)}$

So, there is unfortunately no general justification for conclusions with respect to ALL-properties like liveness; rather it is true:

LIV(PN) <=/=> LIV(TPN)

The problem of time consumption can thus be rephrased: How applicable are the results of net analysis to real systems? We look for results for all possible time restrictions in order to gain independence from the imprecise time inscription of the net.

It is a common opinion in software development technology that the relative progress of processes with respect to each other must not have any influence on the general behaviour of the whole system. The following discussion has to be understood as a direct consequence of this assertion.

Figure 17: Comparison of Petri nets and timed Petri nets.

PN	TPN
firing without time	firing duration
general firing rule	earliest firing rule
single step	maximal step

In connection with the context checking intended two cases have to be distinguished.

1. **not** LIV(PN) --> LIV(TPN) ?

   A non-live Petri net becomes live by the influence of timing. This phenomenon of error masking is known as probe effect /Gait 85, 86/ and can be observed during debugging (without hardware support) of distributed software.

2. LIV(PN) --> **not** LIV(TPN) ?

   A live Petri net becomes non-live by the influence of timing. In context with software interpreted Petri nets, this case is not as comprehensible as the former one. Instead of this, the static analysis is based on the (so far implicit) guess that there are interesting subsets of Petri net structures which are time-independently live (shortly time-invariant).

The influence of timing upon liveness of a net was firstly discussed, as far as we know, in /Godberson 79, 82, 83/. He pointed out that there is no difference in the behaviour of timed Petri nets and an equivalent Petri net in the case of conflict-free nets. In the meantime, there are further results available.

In /Starke 87b/, it has been proven that Petri nets are time-independently live if they are live and (general) extended free choice. In /Starke 88/, this result is generalized to extended simple nets. In /Bause 89/ a similar result is presented for simple net structures of a Petri net extensions by queuing places.

To explain the usefulness of these results concerning software analysis, let‘s now have a somewhat closer look on the problem.

Obviously, the following statements apply:

- The Petri net mapping of a process set without interconnections produces state machines. In the case of strongly connected state machines, the given tokens (program counters) loop forever, their movement can be delayed only (see section 3.2).
- This delay is realized by dependencies from additional places, the so-called synchronization places (highlighted by striping-hatching in the graphical representation).
- The only possibility to get some time dependencies can be reduced to the situation where some (receiving) transitions of different processes (state machines) are in conflict via synchronization places (shortly channel conflict).

Two transitions are involved in a ***(static) conflict*** if they do share at least one preplace. If one of these common preplaces is a synchronization place, then the conflict is called ***channel conflict***, else ***control flow conflict***.

The degree of possible conflicts in CSM can be classified structurally in more detail by the language means available for communication (see Figure 10).

(a) ADDRESSING

(a1) direct addressing:

Sender and receiver know each other. Each synchronization place has exactly one preprocess and exactly one postprocess.

(a2) semi-direct-by-sender addressing:

Only the sender knows the receiver, not vice versa. Each synchronization place has possibly many preprocesses, but exactly one postprocess.

(a3) semi-direct-by-receiver addressing:

Only the receiver knows the sender, not vice versa. Each synchronization place has exactly one preprocess, but possibly many postprocesses.

(a4) indirect addressing:

Sender and receiver do not know each other. Each synchronization place has possibly many preprocesses and many postprocesses.

Static channel conflicts are possible for all addressing types. But only dynamic conflicts[1] are really interesting for context checking. The situation that two transitions, which are involved in a dynamic channel conflict, belong to the same state machine is generally possible according the context-free grammar, but seems to be senseless because it produces obviously a probabilistic program behaviour. The context checker will report such situations as warning (but time dependencies can not happen). Only in the cases (a3) and (a4), dynamic channel conflicts between (receiving) transitions of different state machines are possible.

(b) WAITING

(b1) deterministic waiting:

In the case of this type of waiting, within state machines it is structurally impossible to circumvent channel conflicts. A channel conflict is unable to influence the choice among paths within one state machine (and for this reason, they can‘t change liveness property). All conflict decisions to choose a path depend only on local data values (not explicitly included in the model); they are independent from the relative progress of other processes. If a process has decided to wait for some information, it has no other way out then by receiving the needed information. To change liveness property we need time-dependent conflict decisions choosing between paths of a state machine.

(b2) non-deterministic waiting

Non-deterministic waiting[2] means that the choice of the message to receive (and at the same time, the choice of the control path to follow) depends on the relative progress of neighbouring processes.

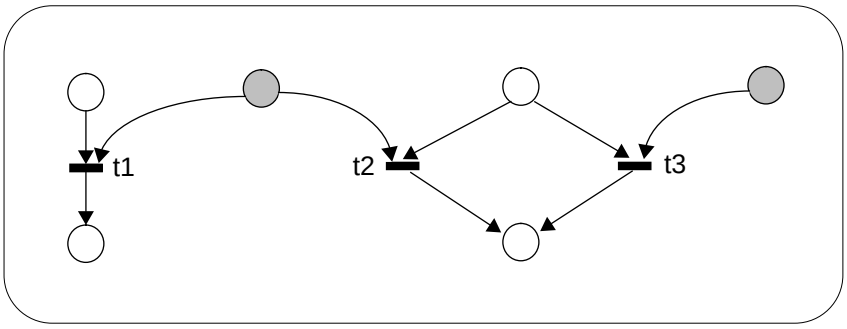
We obtain the chance of time-dependent conflict decisions if channel and control flow conflict overlap in a confusing way.

---

1. Two transitions are involved in a ***dynamic conflict***, if the are involved in a dynamically realizable conflict (i.e. there exists a reachable marking where both transitions have concession), and one transition loses its concession by the firing of the other.

2. E.g. SELECT statement in Ada, RECEIVE CASE action in CHILL, WAIT EVENT action in PDL.

Figure 18: Confusing combination of channel and control flow conflict.

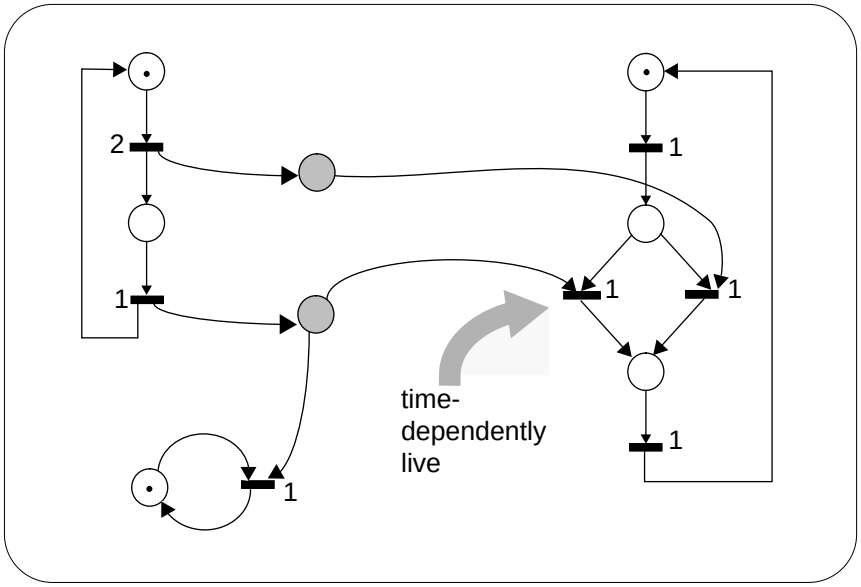


A **confusing conflict** is a situation, where channel and control flow conflict overlap (see Figure 18). Three transitions t1, t2, t3 are involved in a confusing conflict, if

- t1 and t2 are in channel conflict, and
- t2 and t3 are in control flow conflict.

For the transition (t2), which is at the same time involved in two conflicts of different type, the danger of time dependency consists. Figure 19 shows an example of a time-dependently live CSM. The transitions are labeled with (one example of possible) firing durations, which prevent one transition from firing for all the time.

Figure 19: Example of a time-dependently live (timed) CSN.



4.2.2 Conclusions

Let’s summarize the discussion above (see Figure 20).

- The term time dependency describes the situation that the process control flow may depend on the relative progress of neighbouring processes. In terms of Petri net theory, it means that there exists an inscription of firing duration changing the liveness property.
- Time dependency requires dynamic channel conflicts. Dynamic channel conflicts may happen, if more than one process is waiting for the same message.
- Time dependency becomes possible, if channel and control flow conflict establish a confusion. Confusing conflicts require the non-deterministic waiting scheme.

It has been shown informally:

- Without the possibility for non-deterministic waiting there is no chance for time dependency for all types of communication in due of the standardized conflict structures.
- The combination of non-deterministic waiting together with indirect or semi-direct-by-receiver communication bears the danger of time dependencies. This statement is proven by an example (see Figure 19).

As a side-effect, we obtained new local structures whose appearance in a Petri net model should produce a warning to the programmer. At least as long as more precise criteria to check the timing behaviour are not available.

For a further improvement of the technology of distributed software development it would be interesting to describe the border-line between time-dependent and time-independent synchronization structures in more detail.

Maybe there are further standard structures which have to be refused. Or the dangerous timing relations in synchronization structures which are potentially time-dependent can be characterized more precisely. Necessary conditions for time-independently live Petri nets could be useful steps in this direction.

Figure 20: The influence of communication patterns on the conflict structures.

<table>
<tr><th>\ addressing
waiting\</th><th>direct /
semi-direct-by-sender</th><th>indirect /
semi-direct-by-receiver</th></tr>
<tr><td>deterministic</td><td rowspan="2">no
dynamic
channel
conflicts</td><td>channel and
control flow conflicts
appear only separately</td></tr>
<tr><td>non-deterministic</td><td>confusing combination
of channel and control
flow conflicts possible</td></tr>
</table>

Summary:

- *Time independency has been proven for extended simple nets.*
- *Beyond it, it is claimed that, in the case of communicating state machines, time dependency requires a certain type of confusion.*

4.3 Abstraction of Data Dependencies

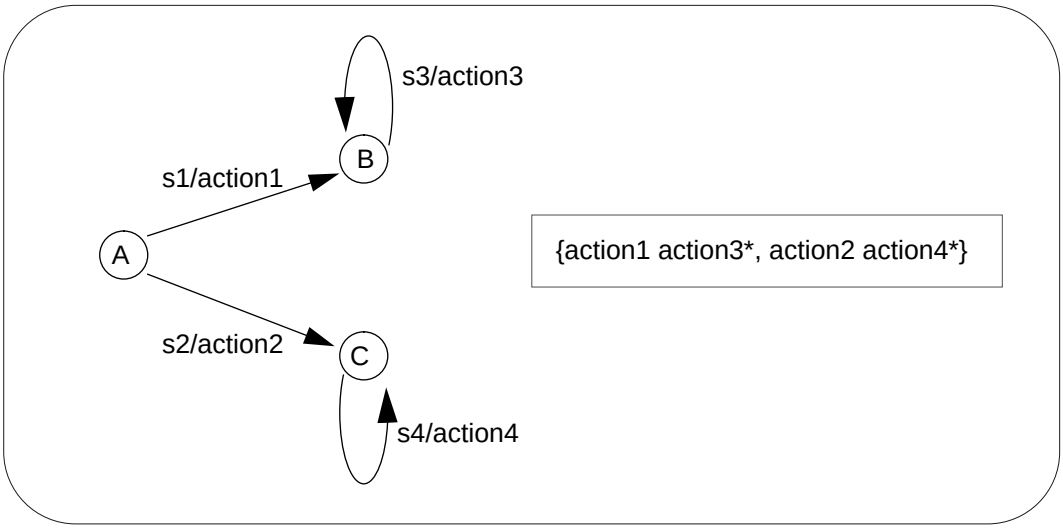
4.3.1 Discussion

To make the situation regarding data dependencies clearer, let‘s discuss Example 2.

Example 2:

Suppose, we have designed a given problem as a finite state automaton with three states and four state transitions as shown in Figure 21.

Figure 21: State automaton of problem specification.



After that, we apply an often used procedure for coding a state-oriented specification by a classical imperative programming language (see Figure 23)[1]. A variable STATE is introduced which controls the choice between the alternatives of a case-statement; each value of the discrete type of STATE corresponds to one alternative reflecting one state. The state transitions are realized by an assignment at the end of each state, followed by a repeated execution of the pattern. Obviously, that’s only a sophisticated way to express jumping. The control flow of this first program solution, E2V1, which we get following this procedure, reflects the intended state transitions only if the data values of the control variable STATE are taken into account.

1. Within example 2, the syntax of the Protocol Description Language PDL /König85a/ is used. But there is no special knowledge required. If desired, see /Heiner 89/, /König 90/ for a short introduction.

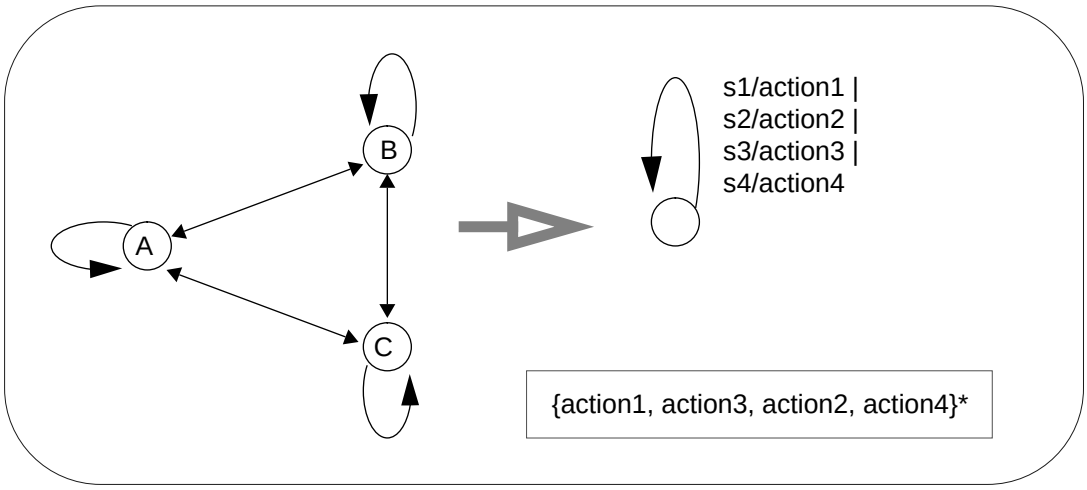
The corresponding Petri net model of the synchronization skeleton together with the program solution is shown in Figure 23 and the state automaton with the same set of state transitions in Figure 22.

Evidently, we have got a set of structurally possible paths which is (relatively) much greater than the set of data-dependently realizable paths. If we want to avoid this gap, we have to model the structure of the state automaton carefully by a slightly more demanding translation. The translation method used should correspond to structured programming practice and its familiar nested program structures. The states have to be embedded in the control flow in such a way that control flow and state transitions coincide. There is neither a need for an auxiliary control variable nor for jumping. We get a second solution, E2V2, whose control flow reflects the state transitions in a straightforward manner.

Again, the corresponding Petri net model of the synchronization skeleton together with its program origin is shown in Figure 24, and the state automaton with the same set of state transitions is already included in Figure 21. This time, there is no difference with respect to the number of execution paths. This means, that the second programming (transformation) style is at least more adequate for a static analysis than the first one[1].

To generalize this little, apparently artificially constructed example, the question arises as to wether it is possible to assess a given programming style with respect to its suitability for static analysis. In the context of our discussion a programming style is called “suitable” if it enforces as far as possible such program structures whose structurally possible path sets are exactly the same as the data-dependently realizable path sets. Such program structures are called, in the context of the envisaged validation method, well-structured ones. So let’s address a question already very popular in the programming community for about two decades: What are well-structured programs?

Figure 22: Supposed state automaton specification which would correspond to version 1.



1. In /Bochmann 87/ it is pointed out that the “structured” specification style forces the designer to consider the different circumstances more explicitly than the transition-oriented one.

Figure 23: Version 1 of Example 2 (E2V1).

Program of solution E2V1:

```
PROTOCOL PART E2V1 (ENTITY SE)                  ! head of the protocol part,
    MESSAGE S1,S2,S3,S4 FROM SE                 ! the name of the partner is SE
    BEGIN
        STATE:=A;                               ! the type of STATE enumerates
        LOOP                                    ! all state names.
            CASE STATE OF
              A: WAIT EVENT                     ! state A
                    S1 <-- SE: ACTION1;         ! the message S1 arrives from SE
                               STATE:=B;
                  | S2 <-- SE: ACTION2;
                               STATE:=C;
                #WAIT
            | B: WAIT EVENT                     ! state B
                    S3 <-- SE: ACTION3;
                               ! STATE:=B;
                #WAIT
            | C: WAIT EVENT                     ! state C
                    S4 <-- SE: ACTION4;
                               ! STATE:=C;
                #WAIT
            #CASE
        #LOOP                                   ! infinite loop
    END
```

Petri net model of solution E2V1:

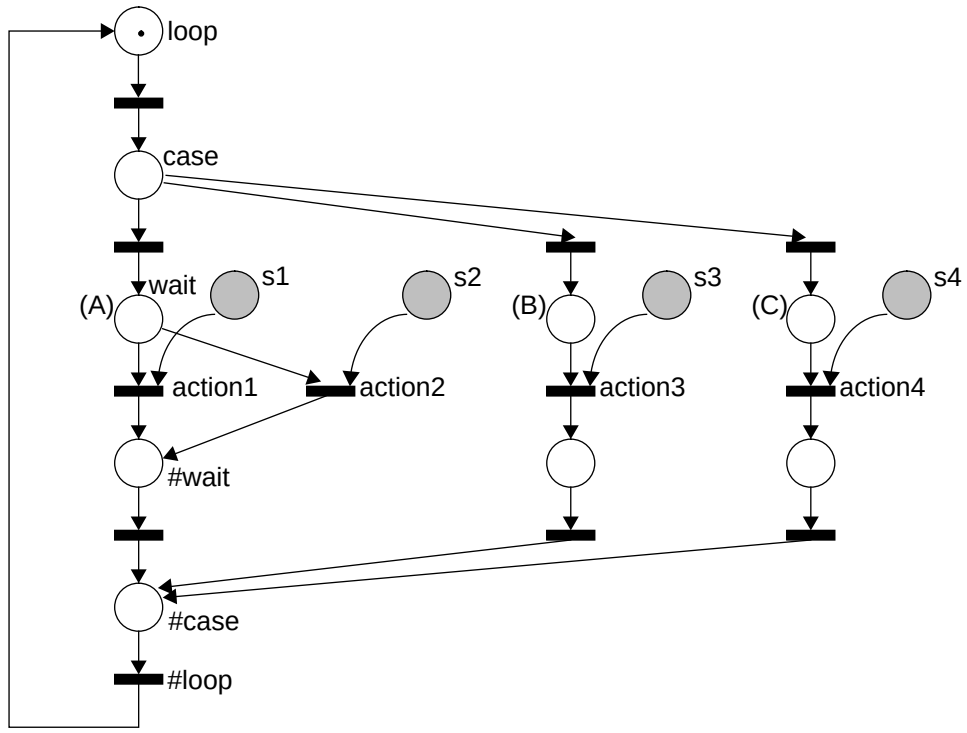
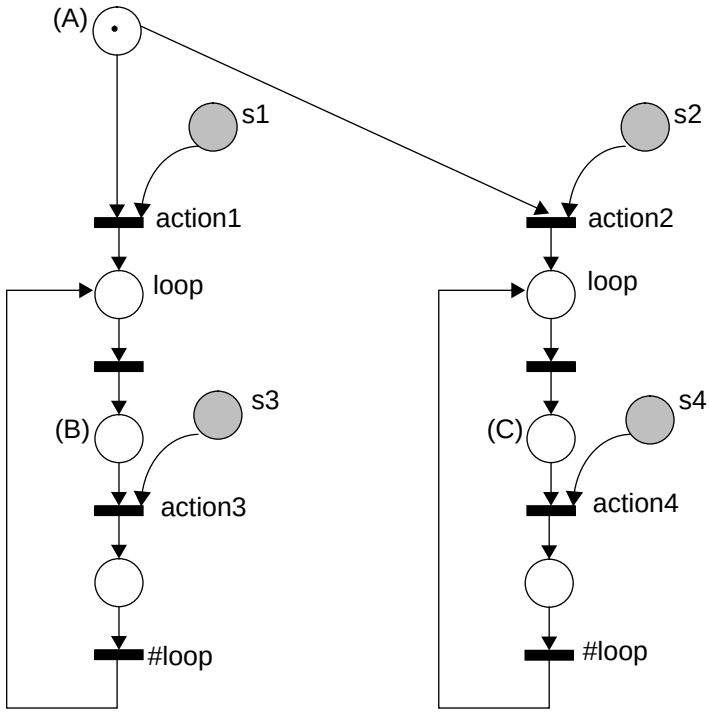


Figure 24: Version 2 of Example 2 (E2V2).

Program of solution E2V2:

```
PROTOCOL PART E2V2 (ENTITY SE)
    MESSAGE S1,S2,S3,S4 FROM SE
    BEGIN
        WAIT EVENT                                  ! state A
            S1 <-- SE:   ACTION1;
                         LOOP
                             WAIT EVENT             ! state B
                                 S3 <-- SE: ACTION3;
                             #WAIT
                         #LOOP
          |  S2 <-- SE:   ACTION2;
                         LOOP
                             WAIT EVENT             ! state C
                                 S4 <-- SE: ACTION4;
                             #WAIT
                         #LOOP
        #WAIT
    END
```

Petri net model of solution E2V2:



At very first glance, the problem seems to be solved by the rules of the theory of structured programming[1] which suggests, roughly said: “If you avoid GOTO, then you will get good programs!”. But looking more carefully, we have to realize that there are no (explicit) GOTO’s at all in program sketch E2V1. Instead, it represents a typical pattern of implicit GOTO’s.

In /Jonsson 89/, there are given several such GOTO-free goto patches. All of these have in common that they avoid explicit goto statements by using flags, auxiliary variables, repetition of code or other types of programming tricks. It can be shown easily by examples that these goto patches result in more complex program structures with a greater amount of data dependencies between control flow branches. As already mentioned above, this is equal to a decrease of static analyzability.

To demonstrate that under certain circumstances, the stubbornness of total goto avoidance leads to more complex programs with respect to NAP complexity combined with an increasing gap between structurally possible and data-dependently realizable path sets, let’s discuss a problem introduced in /Rubin 87/ as the next example.

Example 3:

Let X be an N x M matrix of integers. Write a program that will print the number of the first all-zero row of X, if any, else the information “no”[2]*.*

Forgetting all programming patterns learned from the structured programming paradigm, consider the quite simple looking graphical solution shown in Figure 25 with a NAP complexity equal to four. (Incidentally, it is supposed that there is no solution with less complexity.) But an adequate translation into programming language’s structured control statements comes along with some troubles.

The reason can be traced back to the fact that both loops in the graph of Figure 25 have ***two*** exits, but the loop structure enforced by the rules of structured programming provides only one (compare Figure 26).

Of course, the graphical solution could be coded by low-level statements like

```
i := 1; loop (i <= n) ... goto ... ; i := i+1 #loop ...
```

Or by means of one-exit loops in connection with some auxiliary variables or flags which are checked on the actual result of the loop after leaving the loop. This results evidently in a greater complexity (for this example we get the factor 3.5!); compare E3V1 in Figure 27[3].

Both solutions are roundabout ways at the expense of program clarity. What we are looking for are compact loop structures which relieve the programmer as far as possible of boring details as well as enforce well-structured programs according to the original idea of the theory of structured programming. The program structure should reflect directly the set

1. There is quite a lot of related literature, see firstly e.g. /Dijkstra 65/, /Dijkstra 68/, /Baker 72/, /Wirth 74/. A well-chosen and annotated collection of the most important papers up to 1978 is prepared in /Yourdon 79/.
2. For obvious reasons, the original problem has been expanded by an expected negative response.
3. Within example 3, the reference language (see figure 7) is used.

Figure 25: Graphical solution to Example 3.

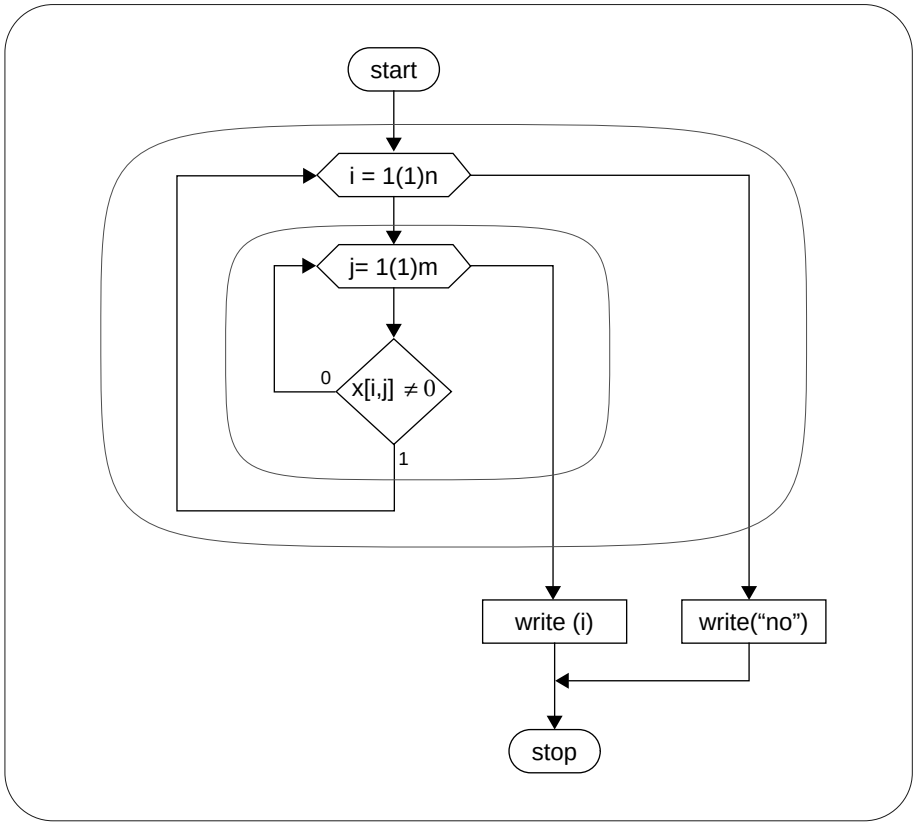


Figure 26: Loop structure of the theory of structured programming.

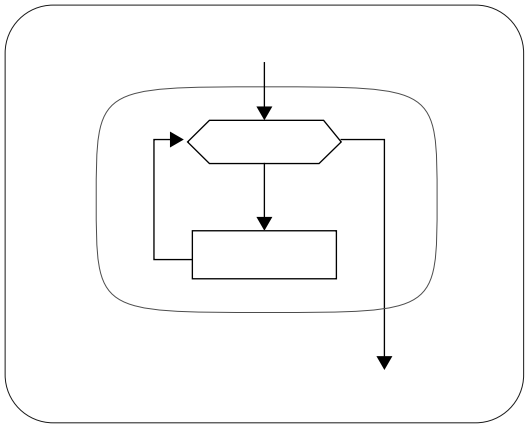


Figure 27: “Structured” solution to Example 3.

Program sketch of solution E3V1:

```
Boolean no_row_found := true,
            all_zero;
check_row: loop (all i and no_row_found)
      all_zero := true;
      check_number: loop (all j and all_zero)
         if x[i,j] ≠ 0
            then all_zero := false              ! abnormal termination of check_number loop
         #if
      #loop check_number;
      if all_zero                               ! normal termination of check_number loop
         then
            write(i);
            no_row_found := false               ! abnormal termination of check_row loop
      #if
#loop check_row;
if no_row_found                                 ! normal termination of check_row loop
   then
      write(“no”)
#if
```

Reduced Petri net model of solution E3V1 (NAP = 14)**.**

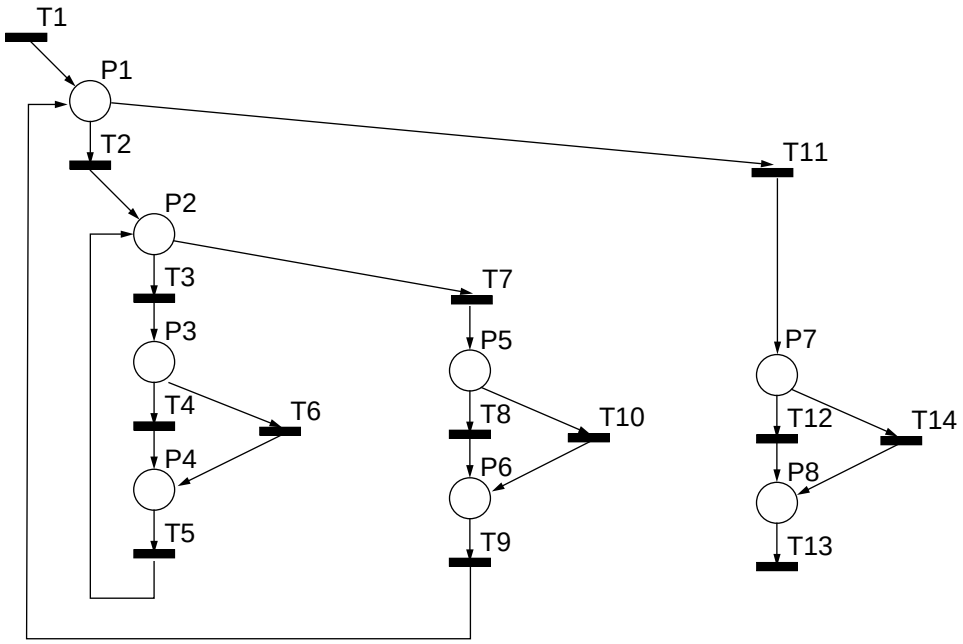
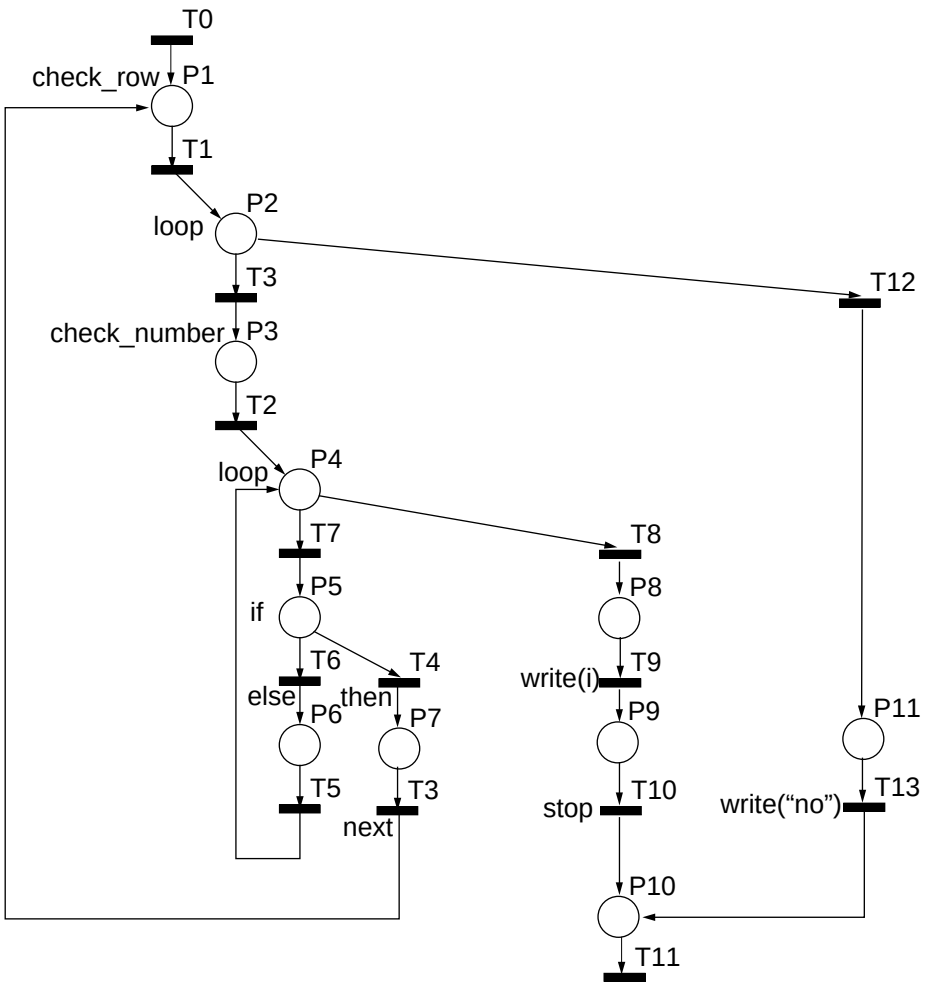


Figure 28: Favoured solution to Example 3.

Program sketch of solution E3V2.

```
check_row: loop all i
      check_number: loop all j
         if x[i,j] ≠ 0
            then next check_row        ! abnormal termination of check_number loop
         #if
      #loop check_number;
      write(i);                        ! normal termination of check_number loop
      stop;                            ! abnormal termination of check_row loop
#loop check_row;
write("no")                            ! normal termination of check_row loop
```

Petri net model of solution E3V2 (NAP = 4)**.**



of possible execution paths. The understanding of the program logic does not require the execution of the program by a trace protocol.[1]

To reach this objective, we need adequate basic control structures where theoretical demands and practical solutions coincide. For that purpose, we introduce a “multi-level multiple two-version exit loop”[2] completed by the general program stop to avoid sophisticated sequences of goto patches:

- multi-level exit feature[3]:

  The structured goto statement must show which loop is being exited. For that purpose labels are introduced which have to appear at the beginning and the end of the loop brackets (showing at the same time that it is used as a multiple exit loop.)

- multiple exit feature:

  The number of abnormal loop terminations is not restricted.

- two-version exit feature:

  Two variants of abnormal loop termination are seen to be necessary[4]:

  exit from the loop’s current iteration -> NEXT loop_label

  exit from total loop -> EXIT loop_label

- STOP -> to hold the program at all

All three structured goto statements are, of course, only syntactic sugar. But compared with the possibly unrestricted use of goto statements, any special exit loop construct imposes a constraint on the control flow. Using these constructs clarifies program logic and supports manageable correctness proofs as well as code optimization /Bochmann 73/. Motivated by the aim of strong static analyzability we come to similar conclusions about an advisable set of structured goto statements, but from a different direction.

Now the solution shown in Figure 28 becomes possible which corresponds directly to the graphical solution in Figure 25 and shares its NAP complexity 4.

To summarize, it is suggested that exactly such problem situations enforce more complex program solutions if they are constrained by the restricting rules of “strong” structured programming (each component has exactly one entry and one exit) which would require some conditional exit or stop statements to describe adequately some further (exceptional) loop exits to leave or continue some loops or to stop the whole program.

---

1. The objective of clear program logic has been strengthened by the fact that programs with unrestricted use of goto statements are hard to optimize, and a correctness proof of them can get quite complicated.
2. Different proposals with similar intentions have been published (see e.g. /Bochmann 73/, /Evans 74/, /Jonsson 89/). But none of them exactly match our demands.
3. The necessity of multi-level exits for well-formed programs is proven by mathematical studies of structured programming, refer /Peterson 73/. Two results of this paper are sharpened in /Fenton 91/.
4. NEXT and EXIT can be understood as multi-level versions of C’s single-level exit statements **continue** and **break**, respectively.

4.3.2 Conclusions

Data dependencies among consecutive path decisions restrict the analytical power of Petri nets with regard to software validation.

In practice, we have to appreciate that the static analyzability of program structures is strongly influenced by the programming paradigms used. We introduced the term “well-structuredness” to qualify program structures which are especially suitable for a static analysis.

Furthermore, arguments have been made against the thoughtless identification of (well-) structured programming with goto-less programming. Instead of this, structured GOTO‘s are considered to be more adequate in some circumstances to express the control flow in a straightforward manner without introduction of goto patches.

But a software quality measure to evaluate a given program in advance with respect to its static analyzability has not yet been found.

A way out could possibly be to improve the Petri net model of the synchronization skeleton by adding more detailed model components reflecting the current values of all control variables[1]. The difference between the structurally and the logically possible reachability set would become smaller. But on the other hand, the reachability graph would be overcrowded with useless states and transitions.

A second alternative could be the application of an higher Petri net class, e.g. some kind of predicate transition nets /Burkhardt 89/, to incorporate all information needed for data-dependent conflict decisions. But this change to higher expressiveness would have to be paid for by a smaller collection of analysis possibilities.

A third alternative, a combination of Petri net based static analysis with some kind of symbolic execution, is worth thinking over /Young 88/, at least to show that an undesirable system state of some EX-property is actually reachable.

Summary:

- *A program’s well-structuredness and static analyzability corollate.*
- *Structured programming is not equal to goto-less programming.*
- *Goto’s are sometimes able to express more directly the control flow than it would be possible according the strong rules of one-entry/one-exit structured programming.*
- *The avoidance of goto’s by goto patches leads in these cases to*
  - *more complex programs (concerning the NAP complexity)*
  - *the situation that the structurally possible path set becomes greater than the logically possible path set.*
- *We are still looking for a method to assess the static analyzability of a given program in advance in order to chose the right model class for validation.*

1. A variable is called control variable if it appears in the expression of a branching statement (in Figure 7: if_statement, case_statement, loop_statement).

5.0 Reduction of Petri Nets

It is our experience that great attention has to be focussed on a permanent reduction of the size of resulting Petri nets in order to save computing resources (memory as well as time).

For this reason, we have accepted the expenditure of separate compilation with subsequent linking. Only the relevant parts of the processes under investigation have to be linked. Open external references like synchronization connections are reported by the linker and must be resolved in a suitable way by the user.

Furthermore, only the synchronization skeleton is produced by the Petri net generator

A drawback of automated Petri net generation lies in some inevitable inefficiencies resulting from context-free assembling of universal net-building components. To overcome this disadvantage we have to continue with a reduction of the automatically generated nets.

For this purpose, the analyzer in use provides a set of local reduction rules based on /Ullrich 77/ and /Berthelot 86/. These rules preserving liveness and boundedness possess the important property that conditions for their application depend only upon local net structures. By this fashion, we can reduce the set of reachable states without knowing the complete one.

However, an uncontrolled application of these transformations destroys the synchronization structure, and an error interpretation, if necessary, on source text level is completely impossible. Therefore, a supplementing reduction set preserving the synchronization structure has been realized. The basic idea of a corresponding general implementation consists in the exception of nodes from reduction. This means, a reduction rule can be applied only, if no “excepted” node will be involved. Evidently, all synchronization places and all sending or receiving transitions have to be excepted. Our experience has proven that the value of the Petri net framework has been significantly enhanced by this supplement.

But, if the resulting total net cannot be mastered efficiently by available computing resources, then it is advisable to repeat this principle in slightly different manner. We have to concentrate on a protocol portion we want to analyze. All processes out of this portion are reduced consecutively whereby only their synchronization places are excepted. By this reduction, we get a consistent minimized description of the environment of the protocol portion under investigation.

Last not least, if in spite of all these precautions the computing resources are not sufficient for analyzing a given net, then we have reached the point where we have to reduce without any restrictions. However, the application of such a reduction should be kept as very last way out in due of the difficult error interpretation.

Unfortunately, there exist no general rules controlling a suitable sequence of the single reduction steps. Such rules would be very useful because the success of reduction (concerning the size of the reduced net) depends in the most cases on the chosen sequence of reduction steps.

6.0 Summary

6.1 Implementation Status

The current implementation status of the Petri net framework is summarized in Figure 29.

All of our protocol engineering examples start with PNG_{PDL} /Grzegorek 91/. PNG_{C} /Czichy 92/ is basically a portable kernel realizing the mapping of ANSI-C, but is open to any extensions by parallel language concepts. A very small subset of Inmos-C has been integrated.

Both Petri net generators output an edge-oriented Petri net representation which has to be converted into input data structures of the utilities used afterwards.

As analysis tool, the Petri net machine PAN /Starke 87a/ has been used extensively up to now. As a useful supplement, a customized graphical Petri net editor dedicated to the special purposes and demands of the net-based software validation process is under development on the basis of DESIGN/OA. The designed properties of the graph editor are especially expected to support the retranslation of Petri net-oriented analysis results back to software-oriented ones.

The total result of the whole validation process is, at least currently, a more or less unstructured set of single step analysis results in terms of the general Petri net theory. The choice and sequence of user-driven analysis steps depend heavenly on experience and conjectures of the tool user.

6.2 Some Hopes for the Future

Our investigations are greatly influenced by the goal of providing a workbench which can be applied by an engineer engaged more in software quality assurance than in Petri net theory.

In our opinion, the level of practicability of a method like this is strongly influenced by the level of an engineer-like preparation of the tool set provided. To fill the gap between theory and application, a lot of work has still to be done in the future to support engineers in handling the underlying model in an effective and easy way.

In this connection, four remarks to Figure 30, “Implementation intentions”, seem to be worth mentioning.

1. The use of further complementing analysis tools, e.g. Product net machine /Ochsenschläger 91/ is under investigation. But we still try to restrict ourself to such Petri net classes which allow also some type of exhaustive analysis (as opposed to functional simulation).

Figure 29: Implementation status.

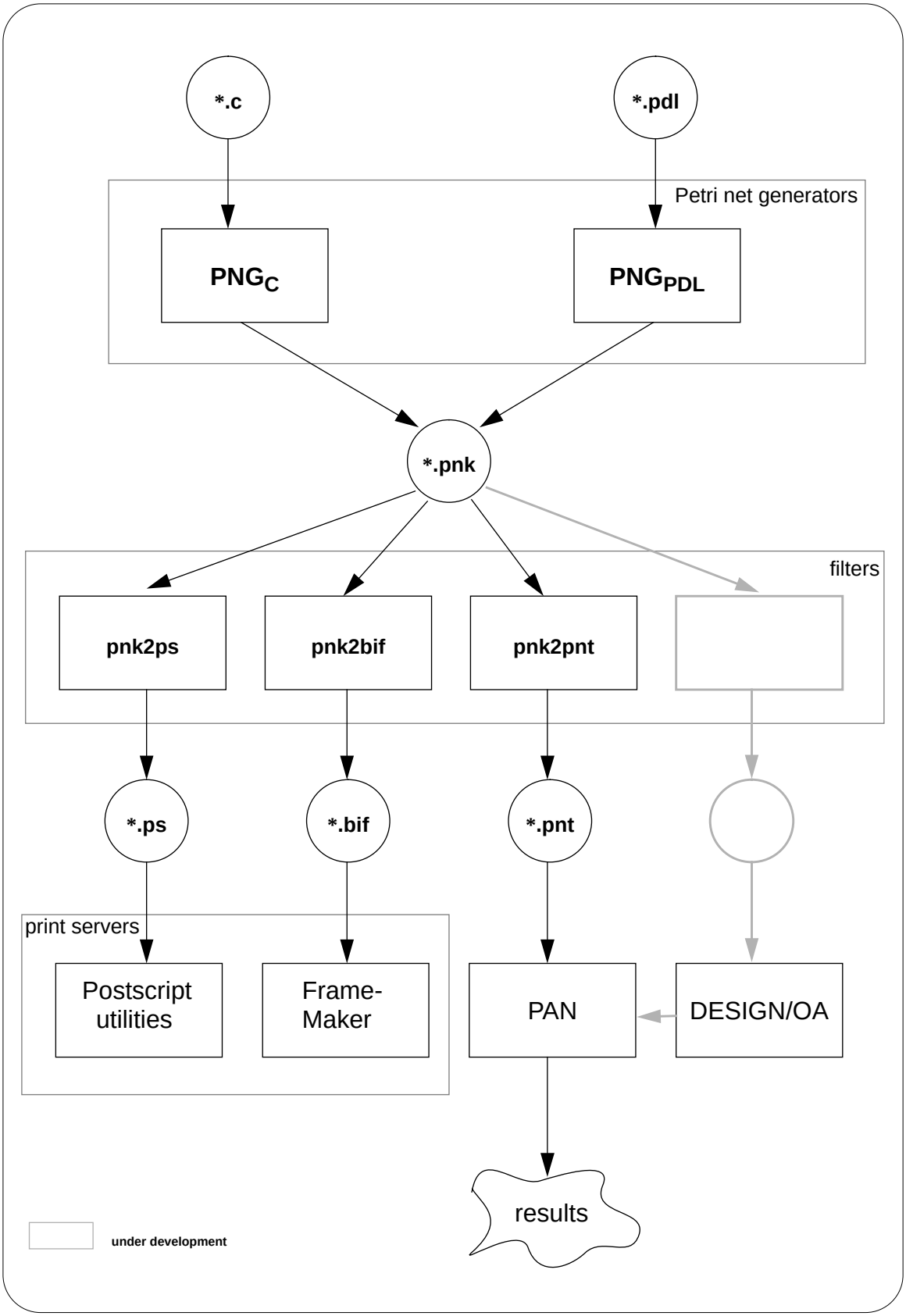
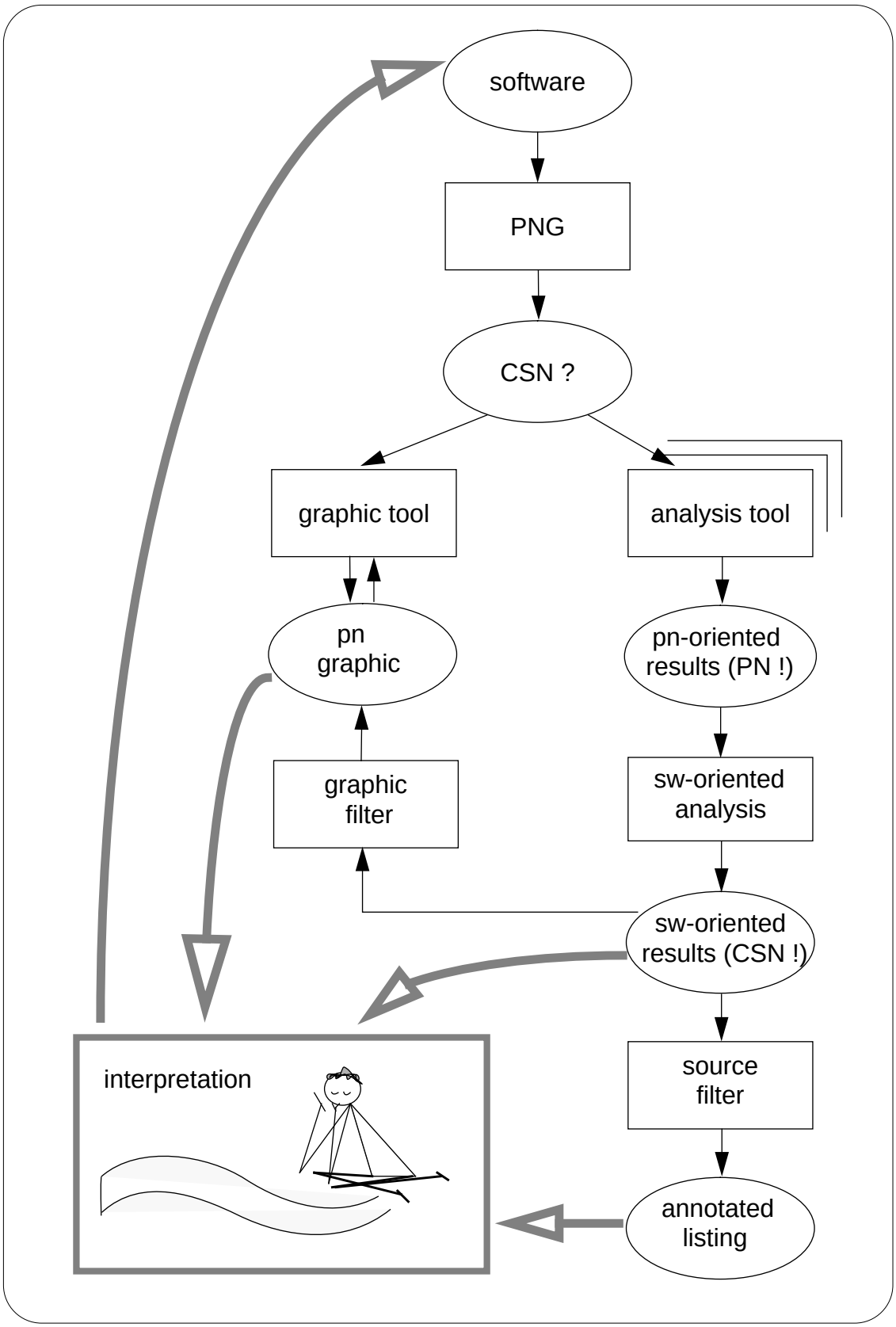


Figure 30: Implementation intentions.



2. All used analysis tools have to be understood as general implementations of well-proved theorems of Petri net theory. As a consequence, the results gained by these tools can only be in terms of general Petri net theory free from any underlying special semantics of the nets under test.

   On the other hand, we do have a lot of detailed information about special properties of the nets to be analyzed because we know where the nets come from and how the translation procedure works.

   Furthermore, it becomes useful to raise several additional questions in connection with software-interpreted Petri nets, questions which cannot be expressed by the fixed set of analysis possibilities stipulated by general analysis tools[1].

   To close this gap, we intend a further knowledge-based component called “software-oriented analysis” to compress the Petri net-oriented results into software-oriented ones as well as to add further flexible possibilities to raise questions about the collected analysis data. At the same time, such a component may be helpful for the validation of special semantic properties.
3. To exploit the modelling power of Petri nets as a tool of reasoning out problems, a significant improvement of the user interface is necessary and seems to be reachable step-wise in two directions:

   - A graphical representation of analysis results, as far as possible,

     either statically, after termination of an analysis step,

     or dynamically, in real-time accordance with analysis progress.
   - An interpretation of analysis results on source text level would be useful, especially with regard to error localization. Two approaches seem to be possible.

     Firstly, all necessary source text information (denotations of program states, process and communication object identifiers) are taken over into the Petri net model. The results of Petri net analysis are expressed in terms of these notions. (For an example of a corresponding reachability graph representation of SDL programs see /Fischer 88/.)

     Secondly, the relation between source text structure and Petri net structure is recorded as well in order to retranslate the analysis results into source text level (annotated listing).
4. The interpretation of analysis results, preferably with the help of well-prepared result representations, is still supposed to be a human activity.

6.3 Final Remarks

The Petri net based software validation as an analytical approach suffers from the disadvantage that it is always done a posteriori. Independent of the validation success, the following advantages of the approach are seen.

1. E.g., if there are dynamic channel conflicts in the net.

Any validation method directs the programmer to rethink the program design.

Furthermore, the lessons learned trying the validation can be turned into useful hints concerning how to construct distributed programs in general, and help us to understand better the problems inherent in distributed programs.

We hope these are little steps in the right direction to extend the so-called theory of structured programming (of sequential systems) to a theory of structured programming of (dependable) distributed systems, providing a discipline to design a priori distributed programs with certain useful properties like dependability or correctness -- whatever this could mean.

But any design discipline can guarantee proper properties only if it is applied properly. So finally, a thorough validation process including excessive testing efforts is still necessary anyway.

7.0 References

CACM - Communication of the ACM
IFB - Informatik-Fachberichte, Springer-Verlag
LNCS - Lecture Notes in Computer Sciences, Springer-Verlag
PSTV - Protocol Specification, Testing and Verification, North-Holland Comp.

7.1 Software Engineering

/Anderson 81/
Anderson, T.; Lee, P. A.:
Fault Tolerance: Principles and Practice;
Prentice Hall 1981.

/Avizienis 86/
Avizienis, A.; Laprie, J.-C.:
Dependable Computing: From Concepts to Design Diversity;
Proc. of the IEEE 74(86)5, pp. 629638.

/Baker 72/
Baker, F. T.:
Chief Programmer Team Management of Production Programming;
IBM Systems Journal 11(72)1, pp. 56-73.

/Bail 88/
Bail, W. G.; Zelkowitz, M. V.:
Program Complexity Using Hierarchical Abstract Computers;
Comp. Lang. 13(88)3/4, pp. 109-123.

/Bal 89/
Bal, H. E.; Steiner, J. G.; Tanenbaum, A. S.:
Programming Languages for Distributed Computing Systems;
ACM Computing Surveys 21(89)3, pp. 261-322.

/Basili 87/
Basili, V. R.; Selby R. W.:
Comparing the Effectiveness of Software Testing Strategies;
IEEE Trans. on SE 13(87)12, pp. 1278-1296.

/Belli 88/
Belli, F.; Jedrzejowicz, P:
Fault-Tolerant Programs;
Angewandte Informatik (88)12, pp. 533-538.

/Bochmann 73/
Bochmann, G. v.:
Multiple Exits from a Loop Without the GOTO;
CACM 16(73)7, pp. 443-444.

/Bochmann 87/
Bochmann, G. v.; Verjus, J. P.:
Some Comments on Transition-Oriented Versus Structured Specifications of Distributed Algorithms and Protocols;
IEEE Trans. on SE 13(87)4, pp. 501-505.

/Caillet 87/
Caillet, J.-F.; Bonnet, C.; Raither, B.:
High Level Interpretation of Execution Traces of Ada Tasks;
Proc. 1st ESEC, Strasbourg, Sept. 1987, LNCS 289, pp. 309-317.

/Castanet 85/
Castanet, R.; Guitton, P.; Rafiq, O.:
An Automatic System for the Study of Protocols: A Presentation and Critique Based on a Worked Example;
PSTV IV, 1985, pp. 111-125.

/Dijkstra 65/
Dijkstra, E. W.:
Programming Considered as a Human Activity;
Proc. of the IFIP Congress, 1965, pp. 213-217.

/Dijkstra 68/
Dijkstra, E. W.:
Go To Statement Considered Harmful;
CACM 11(68)3, pp. 147-148.

/Dillon 84/
Dillon, L. K.:
Analysis of Distributed Systems Using Constrained Expressions;
COINS Technical Report 84-18.

/Eckert 85/
Eckert, H.:
Ein mathematisches Verfahren zur automatisierten Verifikation von Kommunikationsprotokollen;
GMD-Berichte 144; 1985.

/Evans 74/
Evens, R. V.:
Multiple Exits from a Loop Using Neither GO TO nor Labels;
CACM 17(74)11, p. 650.

/Fagerström 88/
Fagerström, J.:
A paradigm and system for design of distributed systems;
Linkping Univ., Diss. No 174, 1988.

/Fenton 87/
Fenton, N. E.; Kaposi, A. A.:
Metrics and Software Structure;
Inf. and Softw. Technology 29(87)6, pp. 301-320.

/Fenton 91/
Fenton, N. E.; Whitty, R. W.:
Program Structures: Some New Characterizations;
Journal of Computer System Sciences (91)143, pp. 467-483

/Gait 85/
Gait, J.:
A Debugger for Concurrent Programs;
Software - Practice and Experience 15(85)6, pp. 539-554.

/Gait 86/
Gait, J.:
A Probe Effect in Concurrent Programs;
Software - Practice and Experience 16(86)3, pp. 225-233.

/Groz 85/
Groz, R.; Jard, C.; Lassudrie, C.:
Attacking a Complex Distributed Algorithm from Different Sides: an Experience with Complementary Validation Tools;
Computer Networks and ISDN Systems 10(85)Dec., pp. 245-257.

/Hoare 73/
Hoare, C.A.R.
High Level Programming Languages - the Way Behind;
Proc. British Computing Society Conf. 1973.

/Hoare 78/
Hoare, C. A. R.:
Communicating Sequential Processes;
CACM 21(78)8, pp. 666-677.

/Jonsson 89/
Jonsson, D.:
Next: The Elimination of Goto-Patches?
SIGPLAN Notices 24(89)3, pp. 85-92.

/Karjoth 87/
Karjoth, G.:
Prozeßalgebra und temporale Logik - angewandt zur Spezifikation und Analyse von komplexen Protokollen;
Univ. Stuttgart, Diss., Jan. 1987.

/Kernighan 90/
Kernighan, B. W.; Ritchie, D. M.:
The Programming Language C;
Prentice Hall 1988.

/Kernighan 90/
Kernighan, B. W.; Plauger, P. J.:
The Elements of Programming Style;
McGraw-Hill Book Company, 1978.

/König 85a/
König, H.; Heiner, M.; Onisseit, J.:
Definierender Bericht der Protokollbeschreibungssprache PDL;
TU Dresden, Sektion IV, Forschungsbericht TU 08 RS-L/LN-K5/015, 1985.

/König 85b/
König, H.; Heiner, M.:
The PDL System - An Unified Approach to the Specification, Verification, and Implementation of Protocols;
in Csaba, L. et al. (eds.): COMPUTER NETWORK USAGE, North-Holland Comp., Amsterdam 1986, pp. 567-582.

/König 88/
König, H.; Tarnay, K.:
A Workstation for Protocol Engineering;
Proc. Network Information Processing Systems, Sofia, 5/1988, part II, pp. 213-224.

/König 90/
König, H.:
Kommunikationsprotokolle;
Akademie-Verlag, Berlin 1990.

/Laprie 87/
Laprie, J. C.:
Dependable Computing: Concepts and Terminology;
Proc. 10th FTSD Int. Conference on Fault-Tolerant Systems and Diagnostics, Varna, 1987, pp. 44-53.

/Leblanc 87/
Leblanc, T.; Mellor-Crummey, J. M.:
Debugging Parallel Programs with Instant Replay;
IEEE Trans. on Computers 36(87)4, pp. 471-482.

/Leblanc 88/
Leblanc, T. J.; Miller, B. P.:
Summary of ACM Workshop on Parallel and Distributed Debugging;
Operating System Review, 22(88)4,.

/McCabe 76/
McCabe, T.J.:
A Complexity Measure;
IEEE Trans. on SE 2(76)4, pp. 308-320.

/Myers 87/
Myers, G. J.:
Methodisches Testen von Programmen;
Oldenbourg Verlag 1987.

/Nejmeh 88/
Nejmeh, B. A.:
NPATH: A Measure of Execution Path Complexity and its Applications;
CACM 31(88)2, pp. 188-200.

/Neusser 88/
Neusser, H.-J.; Schwamborn, U.:
Wissensbasierte Software-Validation unter Unix;
Arbeitspapiere der GMD 289, Feb. 1988.

/Peterson 73/
Peterson, W. W.; Kasami, T.; Tokura, N.:
On the Capabilities of While, Repeat, and Exit Statements;
CACM 16(73)8, pp. 503-512.

/Randell 78/
Randell, B.; Lee, P. A.; Treleaven, P. C.:
Reliability issues in computing system design;
Computing Surveys 10(78)2, 123-165.

/Rodriguez 87/
Rodriguez, V.; Tsai, W. T.:
A Tool for Discriminant Analysis and Classification of Software Metrics;
Information and Software Technology 29(87)3, pp. 137-150.

/Rubin 87/
Rubin, F.:
“GOTO Considered Harmful” Considered Harmful;
CACM 30(87)3, pp. 195-196.

/Rudin 86/
Rudin, H.:
Tools for Protocols Driven by Formal Specification;
LNCS 284, 1986, pp. 127-152.

/Sajkowski 85/
Sajkowski, M.:
Protocol Verification Techniques: Status Quo and Perspectives;
PSTV IV, 1985, pp. 697-720.

/Shepperd 88/
Shepperd, M.:
A Critique of Cyclomatic Complexity as a Software Metric;
Software Engineering Journal (88)3, pp. 30-35.

/Sidhu 86a/
Sidhu, D. P.; Blumer, T. P.:
Verification of NBS Class 4 Transport Protocol;
IEEE Trans. on Comm. 34(86)8, pp. 781-789.

/Sidhu 86b/
Sidhu, D. P.:
Authentication Protocols for Computer Networks: I;
Computer Networks and ISDN Systems 11(86), pp. 297-310.

/Siewiorek 91/
Siewiorek, D. P.:
Architecture of Fault-Tolerant Computers: An Historical Perspective;
Proc. of the IEEE 79(91)12, 1710-1734.

/Thomsen 87/
Thomsen, K. S.; Knudsen, J. L.:
A Taxonomy for Programming Languages with Multisequential Processes;
The Journal of Systems and Software 7(87), 127-140.

/Tyrrell 86/
Tyrrell, A. M.; Holding, D.:
Design of Reliable Software in Distributed Systems Using the Conversation Scheme;
IEEE Trans. on SE 12(86)9, pp. 921-928.

/Venkatraman 86/
Venkatraman, R. C.; Piatkowski, T. F.:
A Formal Comparison of Formal Protocol Specification Techniques;
PSTV V, 1986, pp. 401-420.

/Wirth 74/
Wirth, N.:
On the Composition of Well-Structured Programs;
Computing Surveys 6(74)4, pp. 247-259.

/Young 88/
Young, M.; Taylor, R. N.:
Combining Static Concurrency Analysis with Symbolic Execution;
IEEE Trans. on SE 14(88)10, pp. 1499- 1511.

/Yourdon 79/
Yourdon, E. N.:
Classics in Software Engineering;
Yourdon Press 1979.

7.2 Petri Nets

/Baumgarten 90/
Baumgarten, B.:
Petri-Netze, Grundlagen und Anwendungen;
B.I.-Wissenschaftsverlag 1990.

/Bause 89/
Bause, F.; Beilner, H.:
Eine Modellwelt zur Integration von Warteschlangen- und Petri-Netz-Modellen;
IFB 218, 1989, pp. 190-204.

/Berthelot 86/
Berthelot, G.:
Checking Properties of Nets Using Transformations;
LNCS 222, 1986, pp. 19-40.

/Best 86/
Best, E.; Fernandez, C.:
Notations and Terminology of Petri Net Theory;
Arbeitspapiere der GMD 195, 1/1986.

/Best 89a/
Best, E.; Desel, J.:
Partial Order Behaviour and Structure of Petri Nets;
Arbeitspapiere der GMD 373, 1/1989.

/Best 89b/
Best, E.:
Kausale Semantik nicht sequentieller Programme;
GMD-Bericht Nr. 174, 1989.

/Burkhardt 89/
Burkhardt, H. J.; Ochsenschläger, P.; Prinoth, R.:
Product Nets - A Formal Description Technique for Cooperating Systems;
GMD-Studien 165, 9/1989.

/Desel 89/
Desel, J.:
AC/DC-Systems;
Petri Net Newsletter 32/1989, pp. 3-8.

/Desel 91/
Desel, J.; Esparza, J.:
Structure Theory of Free Choice Petri Nets;
Progress Report of the EBRA project 3148 DEMON, DEMON Workshop, Gjern/DK 6/1991.

/Esparza 90/
Esparza, J.:
Synthesis Rules for Petri Nets, and How they Lead to New Results;
Hildesheimer Informatik-Berichte 5/1990.

/Geissler 85/
Geissler, J.:
Zerlegung von diskreten Systemen mit Petri-Netzen;
Diss., Univ. Kaiserslautern, FB Elektrotechnik, 1985.

/Godbersen 79/
Godbersen, H. P.:
Funktionsnetze - Ein Ansatz zur Beschreibung, Analyse und Simulation von sozio-technischen Systemen;
IFB 21, 1979, pp. 246-265.

/Godbersen 82/
Godbersen, H. P.:
On the Problem of Time in Nets;
IFB 52, 1982, pp. 23-30.

/Godbersen 83/
Godbersen, H. P.:
Funktionsnetze - Eine Modellierungskonzeption zur Entwurfs- und Entscheidungsunterstützung;
Diss. TU Berlin, Ladewig- Verlag, Reihe IV, Band 10, 1983.

/Kuse 86/
Kuse, K.; Sassa, M.; Nakata, I.:
Modelling and Analysis of Concurrent Processes Connected by Streams;
Journal of Information Processing 9(86)3, pp. 148-158.

/Lautenbach 90/
Lautenbach, K.:
Untersuchung der Anwendbarkeit von Invarianten auf die Verifikation eines Kommunikationsprotokolls gegen die berandeten Dienste. Arbeitspapiere der GMD 441, 4/1990.

/Murata 89/
Murata, T.:
Petri Nets: Properties, Analysis and Applications;
Proc. of the IEEE 77(89)4, pp. 541-580.

/Nitsche 88/
Nitsche, U.:
Erreichbarkeitsanalyse von Produktnetzen und ihre Auswertung in PROLOG;
Arbeitspapiere der GMD 330, 8/1988.

/Ochsenschläger 88/
Ochsenschläger, P.:
Projektionen und reduzierte Erreichbarkeitsgraphen;
Arbeitspapiere der GMD 349, 12/1988.

/Ochsenschläger 90/
Ochsenschläger, P.:
Modulhomomorphismen;
Arbeitspapiere der GMD 494, 12/1990.

/Ochsenschläger 91/
Ochsenschläger, P.:
Die Produktnetzmaschine - Eine Übersicht;
Arbeitspapiere der GMD 505, 1/1991.

/Reisig 82/
Reisig, W.:
Deterministic Buffer Synchronization of Sequential Machines;
Acta Informatica 18(82), pp. 117-134.

/Reisig 85/
Reisig, W.:
Petri Nets, An Introduction;
Springer-Verlag 1985.

/Relewicz 88/
Relewicz, C., Franzen, H.:
Konzepte zur systematischen Systemanalyse mit Petri-Netzen;
IFB 187, 1988, pp. 580 - 590.

/Röhrich 86/
Röhrich, J.:
Parallele Systeme;
IFB 117, 1986.

/Starke 87a/
Starke, P.:
Petri Net Machine;
LNCS 255, 1987, pp. 43-44.

/Starke 87b/
Starke, P.:
Remarks on Timed Nets;
Petri Net Newsletter 27/1987, pp. 37-48.

/Starke 88/
Starke, P.:
Some Properties of Timed Nets under the Earliest Firing Rule;
LNCS 424, 1990, pp. 418-432.

/Starke 90/
Starke, P. H.:
Analyse von Petri-Netz-Modellen;
B.G. Teubner Stuttgart 1990.

/Souissi 89/
Souissi, Y.; Memmi, G.:
Composition of Nets via a Communication Medium;
Proc. 10th Int. Conf. on PN, Bonn, 6/1989, 292-311.

/Tankoano 89/
Tankoano, J.; Derniame, J. C.:
Structure Design of Distributed Systems Using Interpreted Petri Nets;
Proc. 10th Int. Conference on Petri Nets, Bonn, 6/1989, 329-347.

/Ullrich 77/
Ullrich, G.:
Der Entwurf von Steuerstrukturen für parallele Abläufe mit Hilfe von Petri-Netzen;
Univ. Hamburg, IFI-HH-B-36/77, 1977.

/Valmari 89/
Valmari, A.:
Stubborn Sets for Reduced State Space Generation;
Supplement to the Proc. 10th. Int. Conf. on Petri Nets, Bonn, 6/1989.

/Wikarski 88/
Wikarski, D.:
Evaluation of Distributed System‘s Behaviour from Formal Models - some Proposals;
Proc. 2nd Int. Seminar on Modelling and Performance Evaluation, Wendisch- Rietz, 11/1988, Informatik-Reporte/IIR.

/Wikarski 91/
Wikarski, D.:
Zur Modellierung des Zeitverhaltens verteilter Systeme auf der Basis von Netzen;
Interner Arbeitsbericht NAQ03, 12/1991.

7.3 Petri Nets and Software Engineering

/Abel 90/
Abel, D.:
Petri-Netze für Ingenieure, Modellbildung und Analyse diskret gesteuerter Systeme;
Springer-Verlag 1990.

/Arendt 89/
Arendt, F.; Klühe, B.:
Modelling and Verification of Real-Time Software Using Interpreted Petri Nets;
WRTP ‘89, Berlin 10/1989.

/Barbeau 90/
Barbeau, M.; Bochmann, G. v.:
Deriving Analyzable Petri Nets from Lotos Specification;
Univ. de Montreal, Publication # 707, 1/1990.

/Botti 90/
Botti, O.; Hall, J.; Hopkins, R.:
A Petri Net Semantics of Occam 2;
Materialiensammlung Sommerschule Petrinetze, Hildesheim, 9/1990.

/Bourguet 86/
Bourguet, A.:
A Petri Net Tool for Service Validation in Protocol;
PSTV VI, 1986, pp. 8/17-8/28.

/Bruno 86/
Bruno, G.; Marchetto, G.:
Process-Translatable Petri Nets for the Rapid Prototyping of Process Control Systems;
IEEE Trans. on SE 12(86)2, pp. 346-357.

/Carpenter 87/
Carpenter, G. F.:
The Use of Occam and Petri Nets in the Simulation of Logic Structures for the Control of Loosely Coupled Distributed Systems;
Proc. UKSC Conf. on Computer Simulation 1987, pp. 30-35.

/Carpenter 88/
Carpenter, G. F.; Holding, D. J.; Tyrell, A. M.:
The Design and Simulation of Software Fault Tolerant Mechanism for Application in Distributed Processing Systems;
Microprocessing and Microprogramming 22(88), pp. 175-185.

/Carpenter 90/
Carpenter, G. F.:
The Synthesis of Deadlock-free Interprocess Communications;
Microprocessing and Microprogramming 30(90), 695-701.

/Czichy 92/
Czichy, G.:
Implementierung eines Petri-Netz-Generators für INMOS-C-Programme;
Praktikumsbeleg, GMD/FIRST, Berlin 2/1992.

/Dahmen 89/
Dahmen, J.; Heiner, M.:
Ein Ansatz zum systematischen Testen verteilter Software;
Proc. Problemseminar “Programmiersysteme für Mikrorechner”, Bad Saarow, 11/1989.

/Dahmen 90/
Dahmen, J. H.:
A Concept for the Integration of Modelling and Debugging;
Proc. 3rd Int. Seminar on Modelling, Evaluation and Optimization of Dependable Computer Systems, Wendisch Rietz 11/1990, Informatik- Reporte/IIR 12/90, pp. 9-14

/Dahmen 91/
Dahmen, J. H.:
Konzeption zur Kombination von Petri-Netzen und Instant-Replay;
Interner Arbeitsbericht NAQ02, 7/1991.

/Diaz 82/
Diaz, M.:
Modelling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models;
Computer Networks 6(82), pp. 419-441.

/Fengler 91/
Fengler, W.; Philippow, I.:
Entwicklung industrieller Mikrocomputersysteme;
Calr Hanser Verlag, 1991.

/Goltz 88/
Goltz, U.:
Über die Darstellung von CCS-Programmen durch Petrinetze;
Diss. TH Aachen, GMD-Bericht 172, 1988.

/Grzegorek 91/
Grzegorek, M.:
Weiterentwicklung eines PDL/D-Compilers;
Ingenieurbeleg, IIR/AdW, Berlin, 1/1991.

/Grzegorek 92/
Grzegorek, M.:
Statische Analyse verteilter Prozesse mittels Petri-Netzen;
Diplomarbeit, TU Magdeburg, 2/1992.

/Heiner 80/
Heiner, M.:
Ein Beitrag zur Deadlockanalyse anhand einer sprachlich geführten Programmiermethodik;
Diss. TU Dresden, Sektion Informationsverarbeitung, 9/1980.

/Heiner 88a/
Heiner, M.:
Some Remarks on Time-Independently Live Petri Nets;
Petri Net Newsletter 30/1988, pp. 10-17.

/Heiner 88b/
Heiner, M.:
A Complexity Measure of Distributed Programs;
Proc. 2nd Int. Seminar on Modelling and Performance Evaluation, Wendisch- Rietz, 11/1988, Informatik-Reporte/IIR.

/Heiner 89/
Heiner, M.:
Petri Net Based Verification of Communication Protocols specified by Language Means;
Informatik-Reporte/IIR 2/89, 52 p. .

/Heiner 90/
Heiner, M.:
Prospects and Limitations of Petri Net Based Software Validation;
Proc. of the 3rd Seminar on Modelling, Evaluation and Optimization of Dependable Computer Systems, Wendisch Rietz 11/1990, Informatik-Reporte/IIR 12/90, pp. 39-48.

/Hura 81/
Hura, G. S.; Singh, H.; Nanda, N. K.:
A Petri Net Approach to the Evaluation of the Complexity of a Program;
Int. J. Electronics 51(81)1, pp. 79-85.

/Herzog 76/
Herzog, O.:
Zur Analyse der Kontrollstruktur paralleler Proramme mit Hilfe von Petri-Netzen;
Univ. Dortmund, Diss., 1976.

/Hura 84/
Hura, G.S.:
Performance Model of Software Systems Using Petri Nets;
Microelectron. Reliability 24(84)3, pp. 391-393.

/Hura 88/
Hura, G. S.; Atwood, J. W.:
The Use of Petri Nets to Analyze Coherent Fault Trees;

/Joosen 90/
Joosen, W.; Berbers, Y.; Verbaeten, P.:
Towards Parallel Compile Time Debugging of Parallel Applications;
Proc. Int. Conf. on Parallel Computing, Paris 12/1990, pp. 172-180.

/Joosen 91/
Joosen, W.; Verbaeten, P.:
A Deadlock Detection Tool for Occam;
Preprint 2/1991, 19 p.

/Juergensen 85/
Juergensen, W.; Vuong, S. T.:
CSP and CSP Nets: a Dual Model for Protocol Specification and Verification;
PSTV IV, 1985, pp. 253-277.

/Lindquist 87/
Lindquist, M.:
Translating SDL into PrT-Nets;

/Pätzold 89/
Pätzold, P.:
Der Einsatz von PN-Modellen zum Entwerfen und Bewerten verteilter Programme;
Informatik-Reporte/IIR 1/1989.

/Peng 91/
Peng, W.; Puroshothaman, S.:
Data Flow Analysis of Communicating Finite State Machines;
ACM Trans. on Programming Languages and Systems 13(91)3, 399-442.

/Plessmann 87/
Plessmann, K. W.; Wyes, J.:
Veranschaulichung der Echtzeitkonzepte in PEARL anhand von Petri-Netzen;
Angewandte Informatik (87)7, pp. 296-304.

/Shatz 85/
Shatz, S. M.; Cheng, W. K.:
Static Analysis of Ada Programs Using the Petri Net Model;
Proc. of ISCAS 1985, pp. 719-722.

/Shatz 88/
Shatz, S. M.:
Towards Complexity Metrics for Ada Tasking;
IEEE Trans. on SE 14(88)8, pp. 1122-1127.

/Shenker 86/
Shenker, B.; Murata, T.; Shatz, S. M.:
Use of Petri Net Invariants to Detect Static Deadlocks in Ada Programs;
Proc. Fall Joint Comp. Conf., Dallas, Texas, Nov. 1986, pp. 1072-1081.

/Steinmetz 87a/
Steinmetz, R.:
Relationship between Petri Nets and the Synchronization Mechanism of the Concurrent Languages Chill and OCCAM;
8. European Workshop on Application and Theory of PN, Zaragoza, 1987.

/Steinmetz 87b/
Steinmetz, R.:
OCCAM2;
Dr. Alfred Hüthig Verlag, 1987.

/Taubner 89/
Taubner, D.:
Finite Representations of CCS and TCSP Programs by Automata and Petri Nets;
LNCS 369, 1989.

/Wehrsdorfer 89/
Wehrsdorfer, G.:
Implementierung und Nutzung eines Petri-Netz-Generators für Modula-2-Programme;
Ingenieurbeleg, IIR/AdW, Berlin 1/1989.