EINDHOVEN UNIVERSITY OF TECHNOLOGY

The use of Petri net theory for simplexys expert systems protocol checking

Citation for published version (APA):
Lammers, J. O., & Technische Universiteit Eindhoven (TUE). Stan Ackermans Instituut. Information and Communication Technology (ICT) (1990). *The use of Petri net theory for simplexys expert systems protocol checking*. [Pd Eng Thesis]. Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/1990

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 22. Aug. 2022

Research Report
ISSN 0167-9708
Coden: TEUEDE

Eindhoven
University of Technology
Netherlands

Faculty of Electrical Engineering

The Use of Petri Net Theory for Simplexys Expert Systems Protocol Checking

by
J.O. Lammers

EUT Report 90-E-238
ISBN 90-6144-238-9

June 1990

Eindhoven University of Technology Research Reports

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
Eindhoven The Netherlands

ISSN 0167- 9708 Coden: TEUEDE

THE USE OF PETRI NET THEORY FOR SIMPLEXYS EXPERT SYSTEMS PROTOCOL CHECKING

by

J.O. Lammers

EUT Report 90-E-238
ISBN 90-6144-238-9

Eindhoven
June 1990

Final report of the post-graduate course "Information and Communication Engineering" of the Institute for Continuing Education (IVO) of the Eindhoven University of Technology, followed in the period May 1988 till May 1990.

Supervisors: Prof.dr.ir. J.E.W. Beneken
and
Dr.ir. J.A. Blom,
Division of Medical Electrical Engineering,
Faculty of Electrical Engineering,
Eindhoven University of Technology,
P.O. Box 513,
5600 MB Eindhoven,
The Netherlands

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Lammers, J.O.

The use of Petri net theory for Simplexys expert systems protocol checking / by J.O. Lammers. - Eindhoven: Eindhoven University of Technology, Faculty of Electrical Enginnering. - Fig., tab. - (EUT report, ISSN 0167-9708; 90-E-238)
Thesis, Institute for Continuing Education (IVO), Eindhoven. - Met lit. opg., reg.
ISBN 90-6144-238-9
SISO 608.1 UDC 616-089.5 NUGI 742
Trefw.: Petrinetten; expertsystemen / patiëntbewaking; expertsystemen.

Abstract

The Simplexys real time expert systems toolbox allows the design of expert systems that are so compact and efficient that they allow real time operation even on a small computer such as a PC. Simplexys expert systems are also reliable, because the logical correctness and consistency of the knowledge base is checked using a variety of techniques. Due to this knowledge checking before the final expert system is built, little checking is necessary at run time.

A Simplexys knowledge base contains a *protocol* that describes the time-sequencing knowledge that is part of the knowledge base. This protocol closely resembles a Petri net. Petri net theory is used to realize an algorithm that checks the correctness of protocols.

Lammers, J.O.
THE USE OF PETRI NET THEORY FOR SIMPLEXYS EXPERT SYSTEMS PROTOCOL CHECKING.
Faculty of Electrical Engineering, Eindhoven University of Technology, The Netherlands, 1990.
EUT Report 90-E-238

Contents

Abstract 3

Summary 6

1 Introduction 8

2 Condition/Event nets 11
- 2.1 Introduction 11
- 2.2 Graphical representation of a Petri net 11
- 2.3 State transitions 12
- 2.4 Other Petri net classes 14

3 Place/Transition nets 16
- 3.1 Introduction 16
- 3.2 Basic definitions 16
- 3.3 Enabling and firing of transitions 17
- 3.4 Interaction between individual transitions 19
- 3.5 Properties of P/T-nets 21

4 The protocol part of a Simplexys knowledge base 25
- 4.1 The Simplexys toolbox 25
- 4.2 The Simplexys knowledge base 27
- 4.3 The correspondence of protocols and Petri nets 30

5 The Petri net class that represents correct protocols 32
- 5.1 Introduction 32
- 5.2 Requirements for correct protocols 33
- 5.3 Safe Place/Transition nets versus Condition/Event nets 35
- 5.4 Liveness of the protocol 35
- 5.5 A correct protocol represents a Live Safe P/T-net 37

6 Analysis of Petri nets 39
6.1 Introduction 39
6.2 Terminology and theorems 39
6.3 Decomposition of the analysis 43
6.4 Analysis of strong connectedness 44
6.5 Simultaneous firing of transitions 44
6.6 Analysis of liveness and safeness 47
6.7 Methods that reduce the protocol's complexity 49

7 Realization of the protocol checker 52
7.1 Introduction 52
7.2 The data structure 52
7.3 Checking of strong connectedness 53
7.4 Checking of liveness and safeness 54
7.5 Construction of the reachability tree 55
7.6 The data structure of the reachability tree 60
7.7 Error messages 62

8 The user interface of the protocol checker 63
8.1 Introduction 63
8.2 A protocol represents a Petri Net 63
8.3 Syntax checking 65
8.4 Topologic checking 68
8.5 Checking of the dynamic behavior 71
8.6 Properties of a correct protocol 77
8.7 Properties of an incorrect protocol 77
8.8 List of warning and error messages 79

Conclusions 81

References 83

Appendix A Free Choice nets 86

Summary

Many programs in the medical domain contain much very specific heuristic knowledge. In standard software, the medical knowledge would mostly be spread out all through the program, which makes it difficult to change and maintain those programs. In expert systems, the knowledge is separately specified in a knowledge base which is easy to understand, self-documenting, and thus easy to modify and update.

The Simplexys real time expert system toolbox is written in Pascal. The knowledge in a Simplexys knowledge base is compiled by a *knowledge compiler* program into Pascal code. This code results in a runnable expert system when it is linked with the *inference engine*, a Pascal program that peruses the knowledge base in order to derive conclusions from input data. Compilation has two functions: the creation of more efficient code and the verification of the correctness of the code in order to prevent run time errors.

Correctness checking of the knowledge base is performed before the final expert system is built. It entails two different mechanisms. First, semantic checking of the knowledge base involves an analysis of the interaction between the individual chunks of knowledge. Second, *protocol checking* involves the examination of the correctness of the time-sequencing knowledge that is contained in the knowledge base. In a monitoring situation, the notion of *context* is essential; the meaning of the data may depend on the circumstances. The context information and the *transitions* between contexts are implemented in a protocol that closely resembles a Petri net.

The set of active *states* in a Petri net represents a *context* in Simplexys; a context specifies the final conclusions or *goals* which the inference engine must derive from the input data. A context represents the *current situation*, a *transition* represents *progress* from one context to another. In every context, the inference engine will evaluate the conditions that must hold before a transition to a new context can take place. The Petri net denotes a *protocol*: progress by a sequence of transitions from the initial context (protocol start) through other contexts toward the final context (protocol end). Given the close resemblance between Simplexys protocols and Petri nets, the design and realization of the protocol checker is based upon Petri net theory.

In order to be able to use Petri net theory to check the correctness of Simplexys protocols, the correspondence between the protocol and a Petri net class must be established. In Simplexys, a context is a set of *active* states; a state is either true (active) or false (inactive). In a Petri net, an active state is marked with one or more *tokens*. In Simplexys, multi-token states, generally possible in Petri nets, are thus meaningless. Therefore a Petri net class in which there is at most one token per state is needed to represent the protocol; the Petri net classes Condition/Event nets and Safe Place/Transition nets are suitable for this.

P/T-nets rather than C/E-nets are chosen to represent protocols because transitions in a C/E-nets have a restriction that is not implemented in Simplexys. *Safeness* of the P/T-net is checked to guarantee at most one token per state. *Liveness* is checked to guarantee that the protocol never comes to a *deadlock*: a context from which no further progress is possible because no transition can occur. Furthermore, liveness guarantees that the protocol can always *terminate*, independent of the transitions that occurred before.

Finally there is a check for *conflicts* between transitions that can occur at the same time. Conflict checking is only partial, because during checking generally the knowledge about the exact conditions for the occurrence of a transition is missing.

The main goal of checking is to test whether the semantics of the knowledge base agree with the assumed intentions of the knowledge engineer. Where a Simplexys protocol differs from a Live Safe P/T-net, an appropriate warning message is given. Due to checking, knowledge engineers have a tendency to learn to compose their protocols in the form of correct Petri nets; checking assists the knowledge engineer in understanding the knowledge base during its development. If it were possible to execute a knowledge base with a protocol that the checker finds incorrect, run time errors like a deadlock context, a non-terminating context or a conflict could occur.

1 Introduction

This document is the final report of an investigation concerning the use of Petri nets in Simplexys, the real time expert system toolbox that has been developed by the Medical Electrical Engineering division of the Eindhoven University of Technology [Blom, 1990].

The Simplexys expert system toolbox is used to build real time expert systems. Major applications of Simplexys-generated expert systems are in patient monitoring. One of the applications that is currently under development, is a rule based alarms system for a ventilator (respirator; anesthesia machine). The sensor outputs are interpreted by a Simplexys knowledge base in order to generate specific alarm messages [van der Aa, 1990]. Another application is an automatic blood pressure controller for patients undergoing cardiac surgery [Zwart, 1990; Lammers, 1990].

A *protocol* is a description of the *time sequencing knowledge* part of a Simplexys knowledge base. A protocol closely resembles a Petri net, and Petri net theory is used to develop and realize a program that analyzes the correctness of the protocol. Important aspects of protocols in Simplexys knowledge bases are:

A. Validity
The (semantic) meaning of the protocol agrees with the intentions of the knowledge engineer.

B. Correctness
The protocol has a unique meaning; it is executable, and does not come to a deadlock.

C. Documentation
The knowledge base is a documentation of the knowledge that is incorporated in the system; the protocol is part of this documentation.

In order to use Petri net theory to check the correctness of protocols, the correspondence between protocols and a certain Petri net class must be established first.

For the choice of the kind of Petri net that represents correct Simplexys protocols, the following are important:

1. The Petri net *class* that represents protocols.
2. The *properties* that a correct protocol should have.
3. The *interpretation* of a protocol.
4. Software *tools* for protocol design and analysis.

1. The Petri Net class that represents a protocol

There are several families, classes and sub-classes of Petri nets. A Petri net class that is suitable to represent a protocol supports sophisticated constructs, so that a compact and clear specification is possible. However, not every construct should be allowed: the knowledge engineers should be convinced to formulate the problem in such a way that a correct Petri net results.

Furthermore the Petri net class should make checking possible. The significance of the features that can be checked, and the methods through which checking is performed, depends strongly upon the Petri net class that is used.

2. The properties that a correct protocol should have

Each Petri net of some class has its specific properties. Correct protocols should have certain properties; otherwise an appropriate and specific error message should be reported. Furthermore, in order to assist the knowledge engineer with the design and development of the protocol, additional properties can be checked. The following questions must be answered:

A. Which properties must a *correct* protocol have.
B. Which additional properties can be checked in order to *assist* the knowledge engineer.
C. What is the *meaning* of these properties.
D. Through which *methods* can these properties be analyzed.
E. How are *conflicts* against these properties reported.

3. The Simplexys interpretation of a protocol

After the expert system has been built, the Simplexys inference engine will execute the protocol. Important issues are then:

A. The execution of the protocol must agree with Petri net theory.
B. In a real time expert system the execution must be fast enough.
C. Little error checking can be afforded at run time.
D. Run time errors must be reported and if possible solved without a system stop.

4. Software tools for protocol design and analysis

The Simplexys toolbox contains several programs that work together. The *protocol checker* checks whether a protocol represents a correct Petri net. The *interpretation program* is that part of the Simplexys inference engine that executes the protocol. For development and debugging, a *simulation* tool is available.

This document is built up as follows. Chapter 2 is a general introduction to Petri nets. Necessary formal definitions follow in chapter 3. Chapter 4 describes the relation between protocols in a Simplexys knowledge base and Petri nets, and the interaction between the protocol and the remainder of the knowledge base.

In chapter 5 the properties of a correct protocol are specified, taking into account the function of a protocol in a Simplexys knowledge base. Chapter 6 describes the algorithms through which a protocol is analyzed, and chapter 7 describes the implementation of these algorithms. Finally chapter 8 explains the error and warning messages that are reported by the checker.

2 Condition/Event nets

2.1 Introduction

Carl Adam Petri first described Petri nets [Petri, 1962]. During the last 25 years, Petri net theory has been developed further and many new results have been derived. A Petri net models the *time sequential* or dynamic behavior of a system or process, and Petri net theory supports the analysis of such systems or processes. An important characteristic of a Petri net is its *concurrency*: parts of the model can operate concurrently (simultaneously) with other parts [Petri, 1986; Reisig, 1985].

This chapter explains the concepts and terminology of Petri nets. Section 2.2 describes the graphical representation of Petri nets. In section 2.3 the dynamic behavior of an elementary net class (Condition/Event nets) is discussed. Finally chapter 2.4 introduces Place/Transition nets, which are a generalization of Condition/Event nets. Formal definitions will be given in chapter 3.

2.2 Graphical representation of a Petri net

A Petri net contains four parts:
(a) A set of *states*, also called "S-elements", "places" or "conditions", drawn as circles.
(b) A set of *transitions*, also called "T-elements" or "events", drawn as boxes or bars.
(c) A set of *arcs* which represents the relation between states and transitions, also called the "flow relation".
(d) A set of *tokens*, collectively called the "marking".

It depends on the Petri net class which terminology is used. In this chapter Condition/Events nets (C/E-nets for short) are considered [Thiagarajan, 1986; Rozenberg, 1986]; the next chapter treats Place/Transition nets (P/T-nets). According to the Petri net class, S-elements are called "conditions" or "places" and T-elements are called "events" or "transitions"; in Simplexys protocols, the names "states" and "transitions" are used instead. To prevent any confusion, for an S-element the name *state* and for an T-element the name *transition* is used throughout the whole document.

An *active* state is marked with a token; the composition of one or more active states is called the *marking* or the *context* in Simplexys. Generally, the initial marking is indicated when a Petri net is drawn; figure 2.1 shows a C/E-net with its initial marking.

2.3 State transitions

A *marking* in a Petri net represents the set of active states; the *initial marking* is the set of states that is active when the system starts up. In figure 2.1, s0 is the only state in the initial marking.

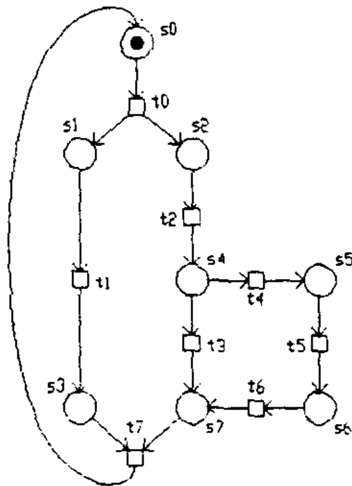

Figure 2.1
A Condition/Event net and its initial marking

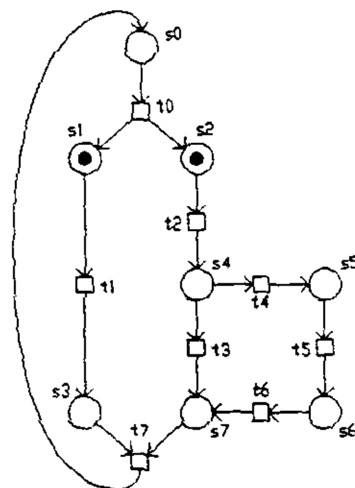

Figure 2.2
A step is enabled

In figure 2.1, transition t0 is *enabled* by the initial marking s0, because all states that have an arc to t0 contain a token. When t0 *fires*, there is a change of state; the *upstream state* s0 looses its token and the *downstream states* s1 and s2 get one (figure 2.2).

In figure 2.2, both t1 and t2 are enabled because the upstream states s1 and s2 are marked. Transitions t1 and t2 can fire *independently* of each other: if t1 fires, t2 is still enabled, and the other way around. There is *concurrency*: the left part of the net can operate independently of the right part. Transitions that do not influence each other's firing, are called a *step*.

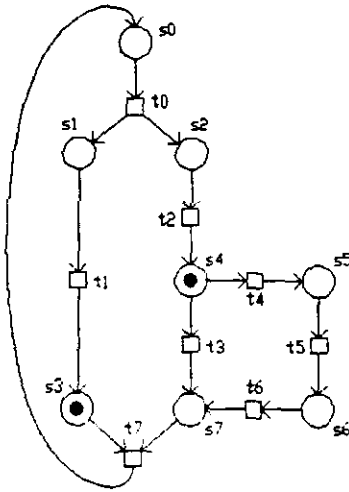

Figure 2.3
A choice is enabled

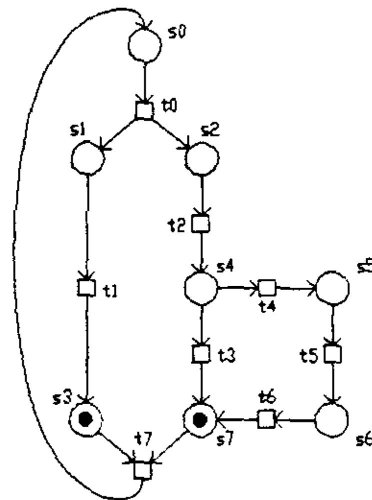

Figure 2.4
Result of a sequence

Figure 2.3 results when both t1 and t2 have fired, in an arbitrary order or at the same time. Now transition t7 is *not* enabled because not all its upstream states are marked with a token: s7 must also be marked to enable t7.

As in figure 2.2, in figure 2.3 two transitions enabled, too. The transitions t3 and t4 are not a step like t1 and t2, because after firing of t3, transition t4 is not enabled anymore and vice versa. There is a *choice* to fire t3 or to fire t4, but not both.

If transition t3 fires, the token flows from s4 to s7. If t4 fires instead, the token flows to s5, and t5 is enabled. When t5 fires, the token flows to s6, and t6 is enabled; when t6 fires, the token flows to s7. The context that results in either case is (s3, s7), as shown in figure 2.4. This context finally enables t7; if that transition fires, the net returns to its initial marking, the one shown in figure 2.1.

In summary, *initially* s0 is marked; by firing t0, two net parts are made active, which can operate independently. *Finally*, when t7 has fired, these net parts are made inactive again and the system returns to its initial context; thus the *initial context* (s0) equals the *final context*. In the right part of the net, there is a choice at s4 between t3 and t4; choosing either path will finally result in s7.

Important (and desired) properties of this net are that both choices at s4 result in s7, and furthermore that after splitting up at t0, the two tokens merge when t7 fires.

The dynamic behavior of a Petri net depends strongly upon the *initial marking*. The same C/E-net as figure 2.1, with another initial marking, for instance (s4, s5) will show a completely different dynamic behavior. When (s4, s5) is the initial marking, the net will sooner or later come to a deadlock because state s3 will never get a token, and thus t7 is never enabled.

The terminology of Petri nets and the way in which transitions fire has been introduced now; only the concept *state capacity* is left. The state capacity defines the maximum number of tokens that a state can carry. In C/E-nets, the state capacity for each state equals one; a state is either active or not, and a state cannot contain two or more tokens. As a consequence, a transition is *not* enabled if one of its downstream states already carries a token. For example, suppose that the initial marking of the net in figure 2.1 equals (s4, s5) rather than (s0). Then the initial marking does not enable transition t4 because state s5 is already marked.

Note that in the Simplexys inference engine the restriction that every output state must be unmarked to enable a transition is not implemented; instead the tokens *merge*. When t4 fires from context (s4, s5), the new context is just (s5). This is discussed in further detail in chapter 5.

2.4 Other Petri net classes

For Condition/Event nets, the state capacity equals one for every state; Place/Transition nets [Reisig, 1986] allow more than one token per state. The *state capacity* is generally not the same for every state, and can be larger than one, up to infinity. Transitions can fire only if the output states have enough capacity left to store the transported tokens.

The *arc weight* specifies the number of tokens that is transported when a transition fires. The P/T-net of figure 2.5 describes a sender (states s1 and s2) that produces 3 messages when t2 fires; the receiver (states s3 and s4) gets 2 messages from buffer s when t3 fires (if a state capacity or an arc weight equals one, the number 1 is not indicated). The state capacity of the buffer equals 8; when the buffer contains more than 5 messages, t2 is not enabled and when it contains less than 2 messages, t3 is not enabled.

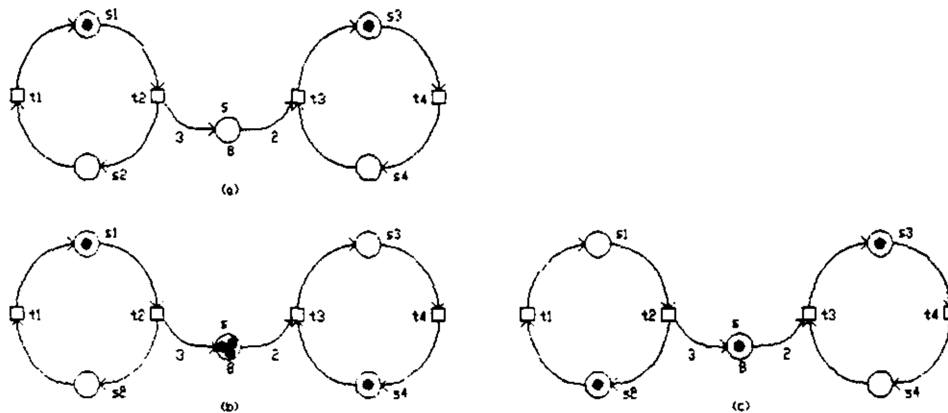

Figure 2.5
Firing of transitions in a P/T-net

A C/E-net is a special kind of P/T-net: all state capacities and arc weights equal one. That is the reason that a transition is not enabled if one of its output states already contains a token.

A further generalization of P/T-nets are Predicate/Event nets (Pr/E-nets for short) [Genrich, 1986], where tokens and arcs are labeled; tokens can only flow along arcs with a corresponding label. Pr/E-nets are not considered in further detail.

Another type of net that is used to specify the function of Programmable Logic Controllers (PLCs) are *Grafcets* [Blanchard, 1977; Valette, 1986]. Since Grafcets disagree with some of the basic Petri net properties, these are not discussed in this document.

In the next chapter, the formal definitions for P/T-nets are given. It may seem curious to study a Petri nets class that allows more than one token per state, but there is a sub-class of P/T-nets that has attractive properties to represent Simplexys protocols.

3 Place/Transition nets

3.1 Introduction

In chapter 2, Condition/Event-nets were introduced. This chapter describes a more universal class of Petri nets: Place/Transition nets (P/T-nets for short). In order to let the terminology in this chapter agree with the Simplexys terminology, the word *state* is used rather than place.

Most of the Petri net terminology that was introduced briefly in chapter 2, is formally defined in section 3.2 and 3.3. Section 3.4 discusses the different ways in which transitions can interact, and section 3.5 finally describes some of the basic properties of Petri nets.

3.2 Basic definitions

A P/T-net contains the following parts [Reisig, 1985].

Definition [3.1] *P/T-net*
N = (S, T; F, K, M, W) (notation)
S : set of *states* or set of S-elements
T : set of *transitions* or set of T-elements
F : $F \subseteq (SxT) \cup (TxS)$, the set of arcs or the *flow relation*
K : the *state capacity* of every state
W : the *arc weight* of every arc
M : the *initial marking* or initial token distribution

The set (SxT) contains all couples {s, t} where $s \in S$ and $t \in T$. The *flow relation* F is the set of couples {s, t} and {t, s}, where s and t are connected by a directed arc. The *inverse flow relation* F^{-1} contains all couples {s, t} where {t, s} is in F, and all couples {t, s} where {s, t} is in F. The *trace* F^* denotes a sequence of elements of F.

The *state capacity* k(s) of a state is the maximum number of tokens that this state can contain; state capacities need not be the same for every state. Special cases occur if all state capacities are equal to one, or if all state capacities are infinite.

The *arc weight* w(s,t) or w(t,s) is the number of tokens that is transported when a transition fires; the arc weight is generally different for every arc. A special case occurs if all arc weights are equal to one.

Definition [3.2] *C/E-net*
A C/E-net as an special kind of a P/T-net: all state capacities $k \in K$ and all arc weights $w \in W$ are equal to 1. The sets K and W are omitted in the definition.

A transition is *enabled* if (1) its input states contain at least the number of tokens of the corresponding arc weight w(s,t) and (2) if its output states have enough free capacity to carry the number of tokens specified by the arc weight w(t,s). The formal definition for an enabled transition is given in the next section.

Definition [3.3] *The matrix representation*
(a) N = (S, T; F, K, W, M) (N represents a P/T-net)
(b) $N_{ij} = w(t_i, s_j) - w(s_i, t_j)$ (contents of the matrix)
(c) $N = |S| \times |T|$ (matrix dimensions)
(d) The marking M is represented by a vector

An example of a P/T-net is given in figure 3.1; its matrix representation is given in figure 3.2.

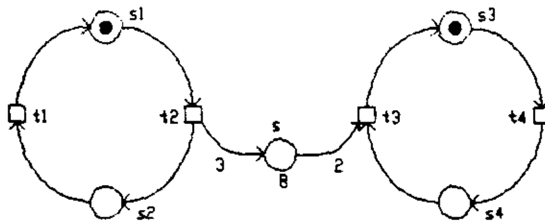

Figure 3.1
A Place/Transition net

$$
\begin{array}{c} \\ s1 \\ s2 \\ s3 \\ s4 \\ s \end{array}
\begin{array}{c} \begin{matrix} t1 & t2 & t3 & t4 \end{matrix} \\ \begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 1 \\ 0 & 3 & -2 & 0 \end{pmatrix} \\ N \end{array}
\qquad M = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}
$$

Figure 3.2
Matrix representation of figure 3.1

3.3 Enabling and firing of transitions

The preset and postset of a net element are defined as follows:

Definition [3.4] *Preset and postset*
(a) $x \in X$ where $X = (S \cup T)$ (x can be any S- or T-element)
(b) Preset $\cdot x = \{y \in X \mid (y,x) \in F\}$
(c) Postset $x \cdot = \{y \in X \mid (x,y) \in F\}$

The preset of a net element is the set of *upstream* net elements or *input* net elements; the postset the set of *downstream* or *output* net elements. The preset or postset of a state is a set of transitions; the preset or postset of a transition is a set of states. Some presets and postsets of the net of figure 3.1 are:

$\cdot t1 = s2$	$t1\cdot = s1$	$\cdot t2 = s1$	$t2\cdot = \{s2, s\}$
$\cdot s = t2$	$s\cdot = t3$	$\cdot s1 = t1$	$s1\cdot = t2$

Further definitions are:

Definition [3.5] *Markings, arc weight and state capacities*
(a) If M is the current marking, M(s) denotes the number of tokens at state s
(b) w(s,t) denotes the weight of the arc from state s to transition t; w(t,s) is defined similarly.
(c) k(s) is the capacity of state s.

The conditions for an *enabled* transition are:

Definition [3.6] *Enabled transition*
M enables t $\Leftrightarrow$
(a) $\forall s \in \cdot t: M(s) \geq w(s,t)$ *and*
(b) $\forall s \in t\cdot: M(s) + w(t,s) \leq k(s)$
(c) Notation M[t> (M enables t)
(d) For a C/E-net: $\forall s \in \cdot t: M(s) = 1$ *and* $\forall s \in t\cdot: M(s) = 0$

A transition is enabled by a certain marking if every state in the preset contains enough tokens (condition a) and if every state in the postset has enough state capacity left to carry the transported tokens (condition b). The conditions for an enabled transition in a C/E-net follow if w(s,t), w(t,s) and k(s) are all set to 1. When an enabled transition *fires*, the old marking is transformed into a new marking according to the following definition.

Definition [3.7] *Firing of a transition*
(a) M is the old marking, M" is the new marking
(b) Transition t is enabled
(c) Firing $\forall s \in \cdot t: M''(s) = M(s) - w(s,t)$ *and*
(d) $\forall s \in t\cdot: M''(s) = M(s) + w(t,s)$
(e) In matrix notation: $M'' = M + Nf$, where $f(t) = 1$ if t fires, else 0
(f) Notation M[t> M" (M is transformed to M" when t fires)
(g) For a C/E-net: $\forall s \in \cdot t: M''(s) = 0$ *and* $\forall s \in t\cdot: M''(s) = 1$

When a transition fires, every input state looses the number of tokens that is specified by the weight of the arc to that transition, and every output state gets the number of tokens given by the weight of the arc from that transition.

3.4 Interaction between individual transitions

The following definitions are illustrated with figure 3.3. A marking can enable more than one transition concurrently. *Independent* transitions are called a step.

Definition [3.8] *Step* transitions t1 and t2 form a step ⇔
(a) Both t1 and t2 are enabled *and*
(b) If t1 fires, then t2 is still enabled *and*
(c) If t2 fires, then t1 is still enabled

Figure 3.4 contains a number of steps. Nets that do not contain any step have no concurrency because there is never more than one active state; such net types are called S-graphs (definition A.1a) and fulfill special topological properties. Two transitions that are concurrently enabled, are called a choice if the transitions are *dependent*.

Definition [3.9] *Choice* transitions t1 and t2 are in choice ⇔
(a) Both t1 and t2 are enabled *and*
(b) If t1 fires, then t2 is not enabled *or*
(c) If t2 fires, then t1 is not enabled

If there is a choice between transitions, they cannot fire simultaneously; if they do, a *conflict* occurs and the new marking is undefined. Figure 3.3 shows some transitions that are in choice. Net types that have no choice are deterministic because it is known for certain what the new context will be; such net types are called T-graphs (definition A.1b) and, like S-graphs, these fulfill special topological properties.

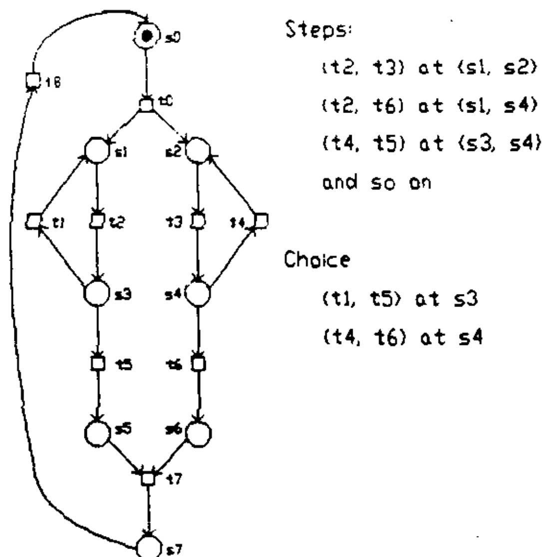

Figure 3.3
Illustration of step and choice

Firing a sequence of transitions transforms a certain marking into another:

Definition [3.10] *Firing sequence*

The set $\{t_1, t_2, t_2, .. t_n\}$ is a firing sequence at marking M_1 ⇔
(a) $M_1[t_1>$ (M_1 enables t_1)
(b) $M_i[t_i>M_{i+1}$ where $M_{i+1}[t_{i+1}>$ (after firing t_i, t_{i+1} is enabled)
(c) The firing sequence $\{t_1, t_2, t_3, .. t_n\}$ results from M_1 to M_n.
(d) Notation $M_1[\{t_1, t_2, t_3, .. t_n\}>M_n$

In figure 3.3 at marking s0, the firing sequence {t0, t2, t3, t6, t5, t7} results in marking s7, but {t0, t3, t4, t2, t5, t3, t6, t7} and many other firing sequences also transform s0 into s7.

The definition of confusion is illustrated with figure 3.4. Confusion is a difficult concept to explain.

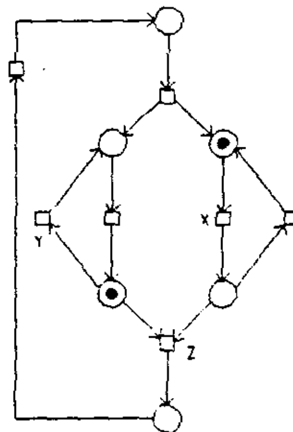

Figure 3.4 Confusion between X, Y and Z

Definition [3.11] *Confusion*

The set of transitions {X, Y, Z} forms a confusion ⇔
(a) Both X and Y are enabled and not in choice with Z at marking M.
(b) After the firing of transition X from marking M, transition Y is in choice with Z, but on the other hand
(c) After the firing of transition Y (instead of X) from marking M, transition X is not in choice with Z.

In other words: confusion (X, Y, Z) means that whether or not there is a choice between two transitions (Y and Z) depends on the firing of another transition (X). Net types that are confusion-free are called Free Choice nets (definition A.2).

3.5 Properties of P/T-nets

Topological properties result from the way in which the states and transitions are connected. Some topological properties like step, choice and confusion have already been introduced. Further topological properties are:

Definition [3.12] *Pure and simple*

(a) Pure: $\forall x \in S \cup T : {}^{\cdot}x \cap x^{\cdot} = \emptyset$ (no self loops)
(b) Simple: $\forall x, y \in S \cup T : {}^{\cdot}x = {}^{\cdot}y$ and $x^{\cdot} = y^{\cdot} \Rightarrow x = y$

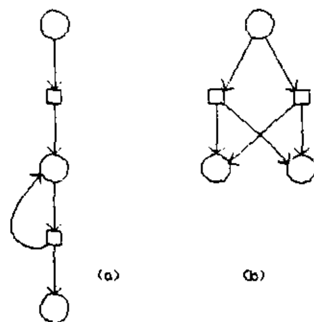

Figure 3.5
(a) A non-pure net
(b) A non-simple net

A net is pure if there is no arc from a transition to a state if there is an arc from that state to that transition. Figure 3.5a gives a non-pure net. A net is simple unless there are states or transitions that have equal upstream and equal downstream transitions or states. Figure 3.5b gives a non simple-net.

Definition [3.13] *Weak and strong connectedness*

(a) A net is weakly connected $\Leftrightarrow$ all $x,y \in S \cup T$ are in the relation $(F \cup F^{-1})^*$. In other words: there is a path along forward or backward arcs $(F \cup F^{-1})^*$ from every S- or T-element to every other S- or T-element.
(b) A net is strongly connected $\Leftrightarrow$ all $x,y \in S \cup T$ are in the relation F^*. In other words: there is a forward path from every S- or T-element to every other S- or T-element.
(c) A strongly connected net is also weakly connected. The converse is generally not true.

Figure 3.6a-c illustrates this definition.

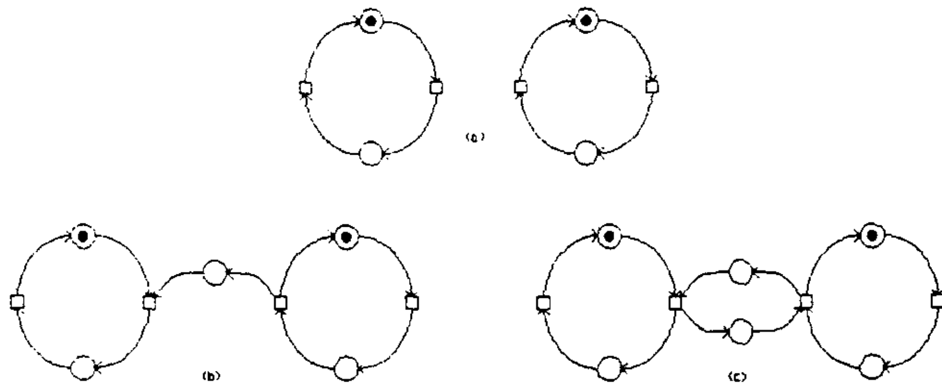

Figure 3.6
(a) Not a weakly connected net
(b) A weakly connected net,
(c) A strongly connected net

Finally the dynamic properties liveness and safeness are introduced. A *reachable marking* is a marking that can result from the initial marking after some firing sequence. The set of all reachable markings can be constructed in a *reachability tree*; the root of the tree is the initial marking M_0. Every element of the tree is linked with its *successor* markings that result from firing an enabled transition. The set of reachable markings is defined as follows:

Definition [3.14] *Reachable markings*

(a) M[t> M" means that M enables t and that M" results from marking M if t fires (definition 3.7f)
(b) set of reachable markings
$[M_0>$ is the smallest set of markings such that $M_0 \in [M_0>$ *and* if $M_1 \in [M_0>$ and $\exists\ t \in T$ such that $M_1[t>M_2$ than $M_2 \in [M_0>$
(c) Every reachable marking in $[M_0>$ can result from M_0 by a certain firing sequence.
(d) For every marking M, the set [M> contains its successor markings; for example, $[M_0>$ contains the successor markings of M_0 (all reachable markings).

The set of reachable markings of figure 2.1 is:

$[M_0>$ = {s0 (s1, s2) (s1, s4) (s1, s7) (s1, s5) (s1, s6)
(s3, s2) (s3, s4) (s3, s7) (s3, s5) (s3, s6)}

Definition [3.15] *Safeness and liveness*

(a) A net is n-safe
$\Leftrightarrow \forall\ M \in [M_0>,\ \forall\ s \in S\colon\ M(s) \le n$
(b) A net is live
$\Leftrightarrow \forall\ M \in [M_0>,\ \forall\ t \in T\colon\ \exists\ M'' \in [M>$ which enables t.

Liveness (figure 3.7a) means that at every reachable marking every transition can be enabled after some firing sequence. Safeness means that for all reachable markings, every state contains at most n tokens. A net is simply called *safe* if it is 1-safe (figure 3.7b).

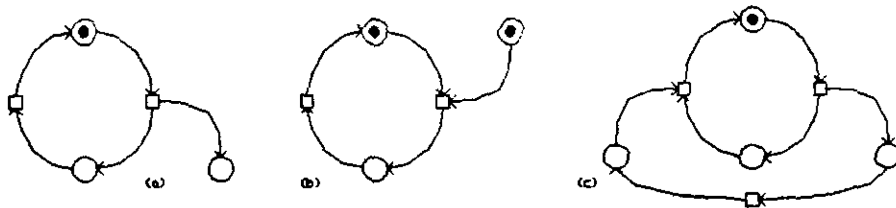

Figure 3.7
(a) A live, but not safe net
(b) A safe, but not live net
(c) A live and safe net

For a 1-safe P/T-net, the firing rule (definition 3.6 and 3.7) can be simplified.

Definition [3.16] *Firing rule for 1-safe P/T-nets*

(a) M enables t $\Leftrightarrow \forall s \in {}^\bullet t$: ${}^\bullet t \subseteq M$
(b) Firing: $M'' = M \setminus {}^\bullet t \cup t^\bullet$
(c) The marking M is a set of states that is marked.
(d) The set of marked states is called a *context* in Simplexys.
(e) All arc weights $w \in W$ are equal to 1, and not denoted.
(f) All state capacities $k \in K$ are infinite, and also not denoted.

Safe P/T-nets are an attractive sub-class of P/T-nets. As a consequence of safeness, all arcs weights must be equal to one, and the state capacity can be any number. Condition 3.6b for an enabled transition in a P/T-net can be omitted because it is known from 1-safeness that an output state is unmarked before firing.

Like C/E-nets, safe P/T-nets never have more than one token per state. The difference is that for C/E-systems multiple tokens are *prevented* by the firing rule, while for safe P/T-net it is *known* from the net topology that multiple tokens cannot occur. The contrast is clear when the conditions for an enabled transition are compared. (Condition 3.6d for C/E-nets and 3.16a for P/T-nets)

Simplexys protocols have only one token per state because multiple tokens *merge*. However, merging of tokens has no place in proper Petri nets, and therefore protocols must be *checked* for 1-safeness, which guarantees that merging of tokens will never occur (chapter 5).

Most of the definitions and results given here are derived from Reisig [1985]. In the remainder of this document, these definitions will be used; they are briefly recalled if necessary.

4 The protocol part of a Simplexys knowledge base

4.1 The Simplexys toolbox

The Simplexys real time expert systems toolbox allows the design of expert systems that are so *compact* that they can run on IBM PC XT or AT compatible computers and that are so *efficient* that they allow real time operation on such hardware [Blom, 1987; Blom, 1990]. Furthermore, they are *safe*, because a variety of checks concerning the logical correctness and consistency of the knowledge base are performed.

Section 4.2 describes the Simplexys knowledge base and the *protocol* that is part of a knowledge base. Such a protocol is a description of the time-sequencing knowledge part of the knowledge base. Section 4.3 will show that a Simplexys protocol closely resembles a Petri net.

Simplexys was developed for medical applications, especially in the domain of patient monitoring. Two applications are currently under development; one of these is an adaptive blood pressure controller [Lammers, 1990]. During and following cardiac surgery, the blood pressure of the patient is frequently artificially decreased through an infusion of the drug sodium nitroprusside. Manual control of the blood pressure by adjusting the flow rate of the infusion pump is often very time- and attention-demanding and interferes with the other tasks of the anesthetist. Automatic control can be substituted, but a problem is the wide variability of the inter- and intra-patient sensitivity to the drug; adaptive control of some form is thus necessary. Moreover, several other aspects, mostly having to do with the safety of the patient, need special consideration.

The second research project is an intelligent alarms system to monitor the integrity of an anesthesia machine [van der Aa, 1990]. Such a machine applies artificial respiration to a patient undergoing surgery by providing his lungs with a mixture of oxygen and anaesthesia gases. If some kind of failure develops somewhere in the machine, the expert system generates an error message which is as specific as possible, such as "leak at endotracheal tube". The expert system reanalyzes the data every 5 seconds.

The expert system is thus used to interpret the *measurements* of physiological variables, and it uses medical *knowledge* to generate a precise diagnosis of the causes of the failure (figure 4.1).

The expert systems methodology is very convenient in the design of process supervision systems, especially in the medical domain where much very specific heuristic knowledge must be incorporated into a successful system. In standard software, the medical knowledge would mostly be spread out throughout the program, which makes it difficult to upgrade and maintain those programs. In expert systems, the medical knowledge is separately specified in a knowledge base, which is easy to read, self documenting and easy to modify and update.

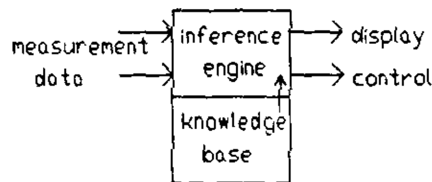

Figure 4.1
A Simplexys expert system

Knowledge is represented in a Simplexys rule base; the rule base is compiled by a *rule compiler* program into Pascal code (both as constant arrays and executable code), and subsequently linked with the *inference engine*, also a Pascal program, resulting in a runnable expert system.

The output of the rule compiler is an efficient internal format of the knowledge, which allows the inference engine to use look-up rather than search; this is one of the reasons why Simplexys expert systems are so fast. Moreover, before the knowledge is linked with the inference engine and compiled by the Pascal compiler, extensive *checking* is performed so that only a few checks on the correctness of the knowledge need to be made at run time (figure 4.2). Checking entails two different mechanisms; checking of the *protocol*, which describes the time-sequencing knowledge of the rule base, is the main subject of this document. *Semantic checking* [Lutgens, 1990] of the knowledge base involves checking the ways in which the individual chunks of knowledge interact, and will not be discussed here.

The executable code is called a runnable *expert system*. The rule compiler, the semantic checker, the protocol checker and the inference engine are Simplexys *expert system tools*. A tracer/debugger tool is also available.

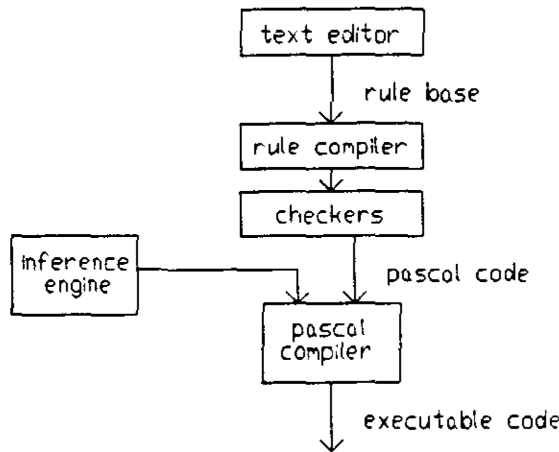

Figure 4.2
Constructing a Simplexys application

The Simplexys expert system toolbox generates fast expert systems, when compared to many other expert systems. At run time, no time is wasted in searching for rules or in matching strings, as is commonly done in other expert systems: knowledge is *compiled* and the inference engine knows where to find a chunk of knowledge if it needs it. Moreover, Simplexys generates expert systems that are safe; the semantic and protocol checks assist in the design of expert systems that can run unattended and without interaction with the user, which are important properties for real time expert systems.

4.2 The Simplexys knowledge base

The task of a *real time* expert system is to process (measured) data as soon as they are available, and to derive conclusions from the data before new data are available. Four types of knowledge exist in Simplexys:

- long term knowledge: *facts* (e.g. the patient's age)
- medium term knowledge: data *remembered* until a new value is assigned (e.g. 'the heart rate is normal')
- short term knowledge: data *forgotten* unless new data are made available (e.g. the actual blood pressure measurement)
- states: the *context* of the process, which remains constant until a transition occurs (e.g. 'the patient is connected to the ventilator').

The Simplexys inference engine will derive goals (final conclusions such as 'no problems') from the input data, using knowledge that is specified in the *rule base*. There are several types of rules; an example of an *evaluation rule* is:

Black: 'Coffee is black'
not Milk *and not* Sugar
then fa : Teaspoon

The first line of the rule gives a symbolic name to it, as well as a text string which can be used in explanations and to show results. The second line specifies the method or expression through which the value of rule Black can be acquired; possible outcomes are *true*, *false* or *possible* (unknown, undecidable). The conclusion of an evaluation rule, like Black above, is obtained through the evaluation of a logical expression that references other rules (Milk and Sugar) which will therefore need to be evaluated as well. The third line states that if rule Black is true, rule Teaspoon is to be made false.

Rule Black is an evaluation rule; its evaluation causes the evaluation of other rules, up to the *primitive rules*, which implement basic concepts. The value of a primitive rule is usually obtained through asking a question or performing a test on data, which can be acquired through Pascal code. Examples of primitive rules are:

Milk: 'There is milk in the coffee'
ask (the user must answer a question)

Sugar: 'There is sugar in the coffee'
test (start of pascal code)
  write ('How many sugar cubes are there in the coffee? ');
  readln (number);
  if number > 1 then *test* := *tr* else *test* := *fa*;
endtest

Besides primitive rules and evaluation rules, Simplexys implements *state rules* and *context switch statements* that describe a protocol. A state rule represents a state (place) in the corresponding Petri net. Examples of state rules are:

Day : 'It is daytime'
state
then goal: Temperature_at_least_15_degrees

Night : 'It is night'
state
initially tr
then goal: Temperature_at_least_5_degrees

The first line of the rule again states a symbolic name and a text string; the second line specifies the rule type. The value of a state rule is either *tr* (true) or *fa* (false); *po* (possible, unknown) is not allowed for state rules.

The third line of rule Night denotes that the initial value of the state is true. The last line of these state rules specifies the goals (final conclusions) that must be evaluated in every run in which that state rule is true.

A state rule obtains its value either through an initially statement, as in the third line of the above rule, or through an *on-statement*. The execution of an on-statement corresponds with the firing of a transition in a Petri net. The following set of on-statements implements a *protocol*:

on Sunset *from* Day *to* Night
on Sunrise *from* Night *to* Day
on Stop *from* Day *to* *

Day and Night are state rules, and the *trigger rules* Sunset, Sunrise and Stop can be rules of any type, for instance:

Sunset: 'sunset'
RedSky *and* SunWest

Sunrise: 'sunrise'
btest (time > = 6)

Stop: 'you want to stop'
ask

The general format of an on-statement is:

on trigger *from* from-list of states *to* to-list of states

Execution of an on-statement proceeds as follows: if all the states in the from-list are true, the corresponding trigger is evaluated and if it returns true, the on-statement is executed. The result of the execution of an on-statement is that the states in the to-list are made *active* (true) and the states in the from-list *inactive* (false).

Besides *initial states*, a protocol also contains *final states*, according to the notion that a medical protocol has both a beginning and an end. Final states are anonymous and marked with a '*', as in the last on-statement above.

The *context* is the set of active states rules. The goals of the active states, as well as the triggers that can invoke the execution of an on-statement, are evaluated in every *run*. After the execution of an *on-statement* a new context is activated, and as a consequence other goals and trigger rules will generally be evaluated from that moment on.

The state rules and on-statements in a Simplexys rule base collectively define a *protocol*; such a protocol closely resembles a Petri net. In the next section the correspondence of protocols and Petri nets is discussed.

4.3 The correspondence of protocols and Petri nets

In the previous section the interaction of the protocol and the remainder of the rule base was clarified. States and transitions (on-statements) play a different role in protocols: states represent *memory* and transitions *action*; in Petri nets, states and transitions are graphically denoted by circles and boxes, respectively.

The *context* in a Simplexys protocol is the set of active states, and it determines which goals and trigger rules the expert system must evaluate. Transitions are defined with *on-statements*, which describe in which way the results of the evaluation of trigger rules can change the active context. Figure 4.3 gives the state rules and the on-statements that implement a greenhouse climate control system, as well its Petri net representation:

s0 : 'Night'
state
initially tr

s1 : 'Heat the greenhouse to day-temperature'
state
then goal: goals concerning heating

s2 : 'Regulate climate'
state
then goal: goals concerning climate regulation

s3 : 'Supply of water'
state

s4 : 'Water supplied, dung added'
state

s5 : 'Water supplied, no more dung added'
state

s6 : 'Supply of water completed'
state

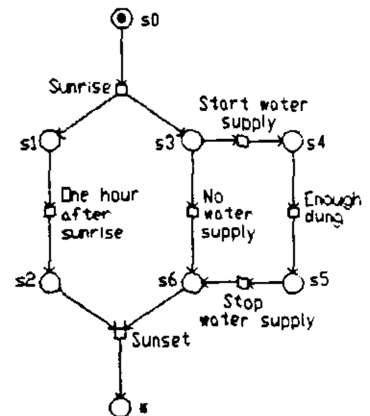

Figure 4.3
The Petri Net description of a greenhouse climate guarding system

on	Sunrise	*from*	s0	*to*	s1 s3
on	One_hour_after_sunrise	*from*	s1	*to*	s2
on	Sunset	*from*	s2 s6	*to*	*
on	No_water_supply	*from*	s3	*to*	s6
on	Start_water_supply	*from*	s3	*to*	s4
on	Enough_dung	*from*	s4	*to*	s5
on	Stop_water_supply	*from*	s5	*to*	s6

The *initial context* is s0, because that state rule is *initially tr* (true, active). Every on-statement corresponds with a *transition* in the Petri net; the execution of an on-statement is similar to firing a transition. The *trigger* of an on-statement is evaluated only if the states in its from-list are true, i.e. only if the transition is *enabled*. In figure 4.3, only the transition labeled Sunrise is enabled. If the trigger is true, the corresponding on-statement is executed. The result of the execution of an on-statement is that the states in its to-list are made active, and the states in the from-list inactive. In the corresponding Petri net the transition *fires*, and every output state gets a token, while every input state looses its token.

The *marking* in a Petri net corresponds with the *context* in a Simplexys protocol; it contains the active states. Petri nets provide the possibility of more than one active state at the same time. In the previous example, the time-sequencing knowledge about climate control is separated from knowledge about the supply of water; however, goals concerning climate control and water supply are evaluated simultaneously.

This Petri net contains one final state, denoted as *. Simplexys stops when no more states are active (except final states). The context in which only final states are active is called the *final context*. In the Petri net of figure 4.3, the final context is reached when transition Sunset fires.

Different *connections* between transitions and states are possible. For instance, in figure 4.3 the transition "*on* Sunrise *from* s0 *to* s1 s3" makes *both* s1 and s3 active, while the two transitions from s3 make *either* s4 or s6 active. This difference in *semantic* meaning is clear from the *syntax* of the on-statements as well as from the *graphical* Petri net representation.

Given the close resemblance between Petri nets and protocols, Petri net theory can be used for the analysis of Simplexys protocols. The possible occurrence of more than one token per state is the only topic in which protocols disagree with Petri nets. In Simplexys, the situation in which a state has more than one token cannot occur because multiple tokens merge. In C/E-nets, multiple tokens are prevented by the restriction on when a transition can fire. In P/T-nets, multiple tokens are allowed. The next chapter will show that a proper protocol represents a *safe* P/T-net, in which merging of tokens never occurs.

5 The Petri net class that represents correct protocols

5.1 Introduction

The previous chapters were an introduction to Petri nets and protocols and their function in Simplexys rule bases; the topic of the next chapters is protocol checking. There are several classes and sub-classes of Petri nets, and every individual net of a certain class should have certain properties. In this chapter, some desired *protocol properties* are formulated and translated into a *Petri net class*: Live Safe Place/Transition nets. The next chapters describe a method that checks whether a protocol represents such a Petri net. Live Safe Free Choice nets [Thiagarajan, 1984; Best, 1986] turned out to be a too restrictive net class to represent correct protocols (appendix A).

This chapter starts from the knowledge engineering point of view, and formulates a set of requirements which a proper protocol must satisfy (section 5.2). These requirements are translated into a Petri net class; the primary requirement that multiple tokens are not permitted results in the use of Safe P/T-nets rather than C/E-nets (section 5.3). Finally, some additional properties that a proper P/T-net must fulfill are formulated; the main property is that the so-called extended net, that is derived from the protocol, is live (section 5.4).

Where possible, properties and requirements in this chapter are described both in Petri net terminology as well as in Simplexys terminology. The protocol is implemented by *on-statements (section 4.3)*. An example of an on-statement is:

on t *from* s1 s2 s3 *to* s4 s5.

This on-statement describes the *transition* t of the *states* (s1 s2 s3) to (s4 s5); it is executed if (1) the states s1, s2 and s3 are all *active*, i.e. *marked* with a token, and if (2) the *trigger rule* t is true. After execution or *firing*, the states s1, s2 and s3 are made inactive and s4 and s5 are made active.

Basic net elements are *states* and *transitions*. A set of states that is true simultaneously is a *context*. The *from-list* or the set of *input states* of transition t equals (s1 s2 s3); the *to-list* or set of *output states* equals (s4 s5).

5.2 Requirements for correct protocols

The requirements below are the ones required by a knowledge engineering point of view. The terminology used here has been explained in sections 2.2 and 2.3.

Requirement 5.1 Initial states and final states

There is at least one initial state (in the rule base defined as *initially tr*), and at least one final state (denoted as '*'). The set of initial states is the *initial context*, the marking when Simplexys starts up; Simplexys stops when only final states are marked, i.e. when reaching the *final context.*

Requirement 5.2 Every transition has input and output states

Every transition has a non empty from-list and a non empty to-list; the syntax of Simplexys [Blom, 1990] and the rule compiler's checking already guarantee this.

Requirement 5.3 One token per state

There is at most one token per state: a state that is true will not become true again due to firing a transition, i.e. two or more tokens on a state, generally possible in P/T-nets, is meaningless for Simplexys.

Requirement 5.4 Every state is reachable

Every state can become true from the initial context after firing a sequence of transitions; it makes no sense to define states that can never become marked.

Requirement 5.5 Every transition can fire

Every transition can be enabled after some firing sequence; it makes no sense to define a transition that can never fire.

Requirement 5.6 The final context is always reachable

Simplexys stops when the final context is reached, a context in which only final states are marked. From every possible context, the final context must be reachable after at least one firing sequence, i.e. Simplexys can always terminate, independently of the transitions that have fired before.

Requirement 5.7 Always at least one transition enabled

A transition is *enabled* when all its input states are true; then the trigger rule is evaluated, and if the trigger rule results in true, the corresponding transition *fires*. Every reachable context must enable at least one transition; then there are no *deadlock* contexts, from which no further change of context is possible.

Requirement 5.8 Firing a transition does not depend upon its output states

In a Petri net, a transition is enabled if (1) the set of input states is true and if (2) its output states have enough capacity to store the transported tokens (definitions 3.6 and 3.7). Since only one token per state is permitted (requirement 5.3), an on-statement should not be executed if one of the states in the to-list is already true. This restriction is important for Petri nets and clear from the graphical representation; for example, it is clear from figure 2.5 that transition t2 cannot fire thrice without firing t3 in between. However, this restriction is not as clear from the syntax of an on-statement as from the graphical representation of a Petri net. The intuitively expected semantic meaning of an on-statement does not agree completely with the semantics of Petri nets. To prevent any confusion, Simplexys protocols should be represented by a Petri nets class in which restriction (2) is always fulfilled.

Requirement 5.9 There is concurrency

Generally, a context enables more than one transition; two transitions that are concurrently enabled form a *step* when after firing one of them, the other is still enabled (for instance in figure 4.3 at context (s1, s4)). There are net classes without concurrency, in which every context contains exactly one state. This is too restrictive for Simplexys protocols.

Requirement 5.10 There is choice between transitions

It two or more transitions are enabled concurrently, and if more than one transition can fire simultaneously, they form a step; a *choice* between transitions occurs when only one of them can fire. After firing one of the transitions in a choice, the other transitions are not enabled anymore (at state s3 in figure 4.3). Net classes without choice exist; for Simplexys protocols, this is too restrictive. Petri nets without choice are deterministic: it is known beforehand which sequence of contexts will become active.

Requirement 5.11 There are no conflicts

If two transitions are in choice, both cannot fire simultaneously. A choice results in a *conflict* when the triggers of the transitions that are in choice, are true at the same time. For instance, a conflict occurs if the triggers "start water supply" and "no water supply" are true at the same time. Generally it is not known beforehand whether or not two triggers will become true at the same time. However, if the triggers are logically or textually equivalent and the on-statements are concurrently enabled, a conflict is certain.

Requirement 5.12 Every net element is connected

Every state and transition in the net is connected with an initial state and with a final state; a state that is not connected with an initial state cannot become true; an active state that is not connected to a final-state can never become false.

5.3 Safe Place/Transition nets versus Condition/Event nets

Requirement 5.3 states that only one token per states is permitted; clearly C/E-nets fulfill this requirement (definitions 3.1 and 3.2). The firing method prevents multiple token states because a transition cannot fire unless its output states are false. However the same firing method disagrees with requirement 5.8. Safe P/T-nets satisfy both requirements: requirement 5.3 is met by safeness, and requirement 5.8 is satisfied by the firing method (definition 3.16).

Stated differently: for a C/E-net, *the firing method* prevents more than one token per state; for a safe P/T-net, *the net topology* prevents that a marked state becomes marked again.

Since P/T-nets allow both choice and concurrency, requirements 5.9 and 5.10 are also fulfilled. Furthermore, by checking for conflicts requirement 5.11 will be fulfilled, but conflict checking can only be partial.

5.4 Liveness of the protocol

The complete protocol in a Simplexys knowledge base can contain several independent sub-protocols. In such cases the Petri net contains some *subnets* that are not connected (figure 5.2a). In a correct protocol every individual subnet contains one or more initial states and one or more final states. The *extended net* results from the original protocol when the individual subnets are *connected* and the final context is *linked* with the initial context (figure 5.2b).

Requirements 5.4 to 5.7 for protocols are almost equal to the requirements for *liveness* for Petri nets (definition 3.15b). The difference is that a Simplexys knowledge base implements a (medical) protocol that has both a start and an end. As soon as no states but final states are active, Simplexys has reached the final context, and stops. In a live P/T-net every transition can always re-occur; thus a live Petri net is *cyclic*. A Simplexys protocol can be made cyclic if a special transition *next patient* is defined from the final context to the initial context. Stated differently, instead of stopping when the final context is reached, Simplexys would start up again from the beginning.

Figure 5.1 shows the general method through which the extended net is derived from the original protocol. Notice that the extension is only made in mind: neither the checking algorithm nor the Simplexys inference engine makes this extension in reality. The net is extended in order to find the correspondence between correct protocols and the powerful Petri net property liveness.

In order to extend the protocol, a new *start state* s0 and an new *end state* sw are added; the start state is connected, by a dummy transition t0, to the original initial states; the end state gets a token from every transition with a * in its to-list. Generally more than one token can arrive at sw (for instance two tokens in figure 5.2b), but these tokens merge. Strictly speaking state sw is not 1-safe, but that is no problem because it only exists in mind. An extra transition *next patient* is added as transition tc; it is fired when all states have lost their tokens except sw. After firing, state s0 gets a token again and the net has returned to the initial context.

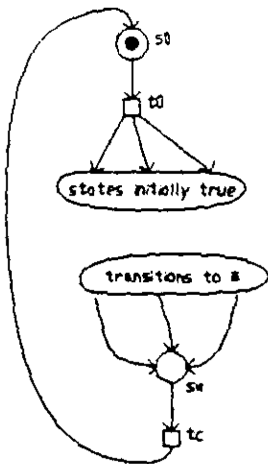

Figure 5.1
Extending a protocol

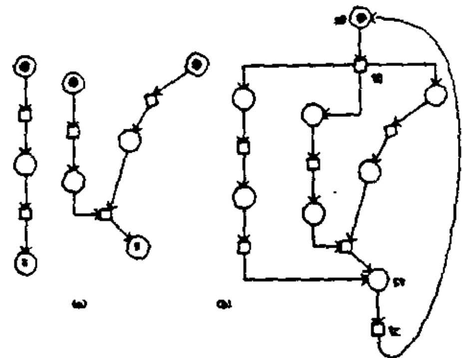

Figure 5.2
Connecting several subnets by extending the net

The extended net that is derived from a correct protocol is *strongly connected*; requirement 5.12 is fulfilled if the extended net is strongly connected (definition 3.13b). A net is strongly connected if there is a forward path from every net element to every other net element. For the original protocol, this means that there is a forward path from an initial state to any state, as well as from any state to a final state.

If the extended net is *live*, the protocol fulfills requirements 5.4 to 5.7. Liveness (definition 3.15b) means that in every reachable context every arbitrarily chosen transition can be enabled after a certain firing sequence. Liveness is a very strong property, too strong for (non-extended) protocols. The net in figure 5.2a represents a correct protocol, but it is not live. Figure 5.2b shows the extended net, which is derived from the same protocol, and which is live.

Indeed, protocol requirements 5.4 to 5.7 equal liveness of the extended net. Requirement 5.5 (every transition can fire) and requirement 5.7 (there is always at least one transition that is enabled) are fulfilled by the definition of liveness. Requirement 5.4 states that every state can become active. Since a correct extended net is strongly connected (requirement 5.12), every state is in the to-list or in the from-list of at least one transition; moreover, every transition can fire (requirement 5.5). A state that is in the to-list of a transition is marked after firing; a state that is in the from-list was marked before firing. Hence every state can be marked (requirement 5.4).

One requirement is left: it must always be possible to reach the final context (requirement 5.6). Liveness means that every transition can always fire from any reachable context. As a consequence, any transition that is enabled by the initial context can always re-occur. Thus the initial context must be reachable from every context, and since the initial context can only be reached from the final context, the final context is always reachable.

5.5 A correct protocol represents a Live Safe P/T-net

Section 5.2 specified a set of requirements for a correct Simplexys protocol. In sections 5.3 and 5.4, the correspondence with a Petri net class was established and additional properties were formulated. This results in the proposition that a correct protocol represents a Live Safe Conflict free Place/Transition net. The Petri net properties and the corresponding protocol requirements are summarized in this section.

Property 5.13 There is at least one initial state and one final state (req 5.1)
A net without initial and final states is not live.

Property 5.14 The extended net is strongly connected (req 5.12)

The extended net is strongly connected if there is a forward path from every state or transition to every other state or transition, i.e. the original protocol contains forward paths from an initial state to any state, as well as from any state to a final state.

Property 5.15 The protocol represents a 1-safe P/T-net (req 5.3 and 5.8)

A P/T-net generally allows more than one token per state; in Simplexys two or more tokens merge. The merging of tokens has no place in P/T-nets, however, and therefore the protocol must be checked for safeness. Moreover, a transition is enabled as soon as its from-list is true.

Property 5.16 The extended net is live (req 5.4 - 5.7)

In the extended net, the initial context can become active again after reaching the final context; if the extended net is live, every transition can (re)occur from every marking by firing a certain sequence of transitions. For the original protocol this means that always at least one transition is enabled, and that the final context is always reachable.

Property 5.17 The net is conflict-free (req 5.9 - 5.11)

P/T-nets allows both choice and concurrency. A conflict between transitions that are in choice occurs when the trigger rules both result in true. Conflicts cannot always be prevented since the conditions for firing a transition are not known to the checker. However, if the triggers of transitions that are in choice are logically equivalent, a conflict is certain.

The combination of these properties results in the use of Live Safe Place/Transition Nets (LS P/T-nets for short), where the property 'live' refers to the extended net, as defined in section 5.4. The next chapters develop a scheme to check a net for being live and safe.

6 Analysis of Petri nets

6.1 Introduction

The previous chapter demonstrated that a correct Simplexys protocol represents a Live Safe P/T-net. This chapter describes the method which checks whether a protocol is such a live and safe P/T-net; the next chapter briefly discusses some implementation strategies. Where Simplexys protocols differ from LS P/T-nets, appropriate warning messages are given (chapter 8). An important part in checking is the construction and analysis of the reachability tree. A Simplexys protocol represents a rather unrestricted type of Petri net. The knowledge engineer who constructs the protocol is generally not a Petri net expert; the popular net class Free Choice nets, which allows sophisticated analysis methods, cannot be used for Simplexys. Analysis of Free Choice nets is based upon deadlocks and traps [Best, 1987] (appendix A). Other checking strategies, such as linear algebraic techniques [Lautenbach, 1987] and synchronic distances [Goltz, 1986] are also not suitable for analysis in this expert system context.

Commercial computer tools are available for analysis [Feldbrugge, 1986; Jensen, 1986], tools that check for liveness, connectedness and other properties. In order to apply such a tool, it could have been integrated with the Simplexys environment, but this would have taken much effort. Moreover, messages that occur during the analysis must also be understandable for a knowledge engineer who is not a Petri net expert.

This chapter start with the definitions and theorems used by the checker (section 6.2). Section 6.3 describes the decomposition of the analysis into sub-checks. Checking connectedness, one of the sub-checks, is discussed in more detail in section 6.4. The topic of the remainder of this chapter is the reachability tree. Section 6.5 describes the impact of on-statements with equivalent trigger rules which can fire simultaneously, and section 6.6 states that liveness and safeness can be demonstrated by an analysis of the tree.

6.2 Terminology and theorems

This section contains the definitions and theorems which are used in the analysis. For P/T-nets, the word *place* is used for an S-element, and the word *marking* for a set of places that is marked simultaneously. This terminology does not agree with that of Simplexys, where *state* is used instead of place and *context* instead of marking. In this chapter the Simplexys terminology is used.

The difference between a sufficient and a necessary condition is important. Denoting A ⇒ B means that A implies B, i.e. that A is a *sufficient* condition for B, while B is a *necessary* condition for A. Furthermore, A ⇒ B means not B ⇒ not A. The text '*if* A then B' means A ⇒ B. Denoting A ⇔ B means that A implies B and that B implies A, i.e. that A is a *sufficient and necessary* condition for B, and reverse. The text '*iff* A then B' means A ⇔ B.

The difference between necessary and sufficient conditions is noted because some necessary, but not sufficient, net properties are analyzed. If *no* error develops during such a test, correctness has not yet proved. However, if an error develops, it is detected in an early stage, and the error can be reported with a more specific message.

A Simplexys *protocol* is implemented by a set of on-statements. Every on-statement represents a *transition* in the corresponding Petri net (figure 6.1).

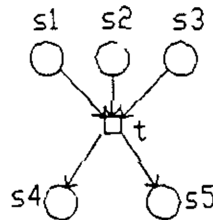

Figure 6.1 Graphical representation of "on t from s1 s2 s3 to s4 s5"

Trigger rule t is evaluated only if all of the *states* s1, s2 and s3 have a token. Stated differently: t is *enabled* if (s1 s2 s3) is (part of) the current *context*. When t is evaluated and results in true, the *input states* s1, s2 and s3 loose their tokens and the *output states* s4 and s5 each get one. The set of states (s1 s2 s3) is called the from-list and (s4 s5) the to-list of the on-statement.

From section 3.3 we recall the definitions of *preset* and *postset*. States and transitions are connected by directed arcs. The preset of a net element is the set of its upstream net elements; the postset is the set of its downstream net elements. For figure 6.1, the preset of t, denoted as ·t, equals (s1 s2 s3), the postset t· equals (s4 s5). Similarly the presets of s4 and s5 contain t, but can contain other transitions as well. Transition t is also part of the postset of the states s1, s2 and s3.

In summary, the following terms refer to the same set of states:

preset	·t	input states	from-list
postset	t·	output states	to-list

A net is *pure* if no on-statement contains an identical state in its from- and to-list. The nets shown in figure 6.2 contain a *self loop*, and are thus non-pure. A net is *simple* if no two or more different net elements are functionally equivalent, i.e. have equal pre- and postsets. For instance, a protocol that contains two on-statements with identical from- and to-lists is not simple. The definitions used in Petri net theory are:

Definition 6.1 *Pure and simple* (see also 3.12)

(a) Pure: $\forall x \in S \cup T : {}^{\cdot}x \cap x^{\cdot} = \emptyset$ (no self loops)
(b) Simple: $\forall x, y \in S \cup T : {}^{\cdot}x = {}^{\cdot}y$ and $x^{\cdot} = y^{\cdot} \rightarrow x = y$
(c) The notation $x \in S \cup T$ means: x can be any net element (state or transition).

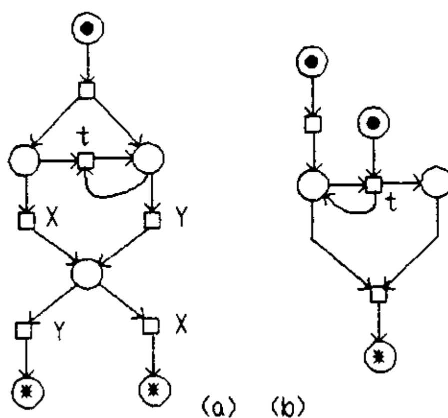

Figure 6.2
Live safe nets that contain a self loop

A net is *strongly connected* if every net element can be reached from every other net element by a path along forward directed arcs. *Weak connectedness* is a similar property, but now a mix of forward and backward arcs is allowed rather than only forward arcs.

Definition 6.2 *Weak and strong connectedness* (see also 3.13)

(a) A net is weakly connected *iff* all $x,y \in S \cup T$ are in the flow relation $(F \cup F^{-1})^*$.
(b) A net is strongly connected *iff* all $x,y \in S \cup T$ are in the flow relation F^*.
(c) The trace X^* denotes a sequence of elements of X.

A Petri net is *safe* if no state will ever have more than one token. Stated differently, if an on-statement is executed, it is known that all the output states are inactive. A Petri net is *live* if every transition can always (re)occur, independently of the sequence of transitions that fired before.

Definition 6.3 *Safeness and liveness* (see also 3.15)
(a) A net is safe
$\Leftrightarrow \forall\ M \in [M_0>,\ \forall\ s \in S:\ M(s) \leq 1$
(b) A net is live
$\Leftrightarrow \forall\ M \in [M_0>,\ \forall\ t \in T: \exists\ M'' \in [M>$ that enables t.

In this definition, M is a context or a *reachable marking*, and M_0 is the *initial context*. The set $[M_0>$ denotes all contexts that can be reached from M_0 by any firing sequence (definition 3.10). M(s) denotes the number of tokens that state s contains in context M. The property safeness means that no reachable context contains a multi-token state. Liveness means that in every reachable context M, for any transition t, there is a firing sequence resulting in a new reachable context M'' that enables t. This means that in every context any arbitrary transition can be enabled, after firing some sequence of transitions.

Both nets of figure 6.2 represent quite tricky protocols because the Petri nets contain a self loop, but from the Simplexys point of view both nets of figure 6.2 are live and safe, and thus correct. However, in order to gain liveness from the Petri net point of view, the nets must be extended with a transition *next patient* that can fire from the final context to the initial context. Stated differently, instead of terminating after reaching the final context, Simplexys would start up again from the beginning. Hence the protocols that are described by the Petri nets in figure 6.2 are correct, because their *extended* nets are both live and safe.

A correct protocol represents a *strongly connected* extended Petri net. From now on the properties liveness and connectedness will refer to the extended net.

Theorem [6.4] *Liveness and safeness imply strong connectedness*
If a P/T-net is live and safe $\Rightarrow$ it is strongly connected.

This theorem states that strong connectedness is a necessary condition for liveness and for safeness. Another necessary condition is:

Theorem [6.5] *Necessary condition for liveness and safeness*
If a P/T-net is live and safe $\Rightarrow$ for all $x \in S \cup T : {}^{\bullet}x \neq \emptyset \neq x^{\bullet}$

For a protocol, this theorems means (1) that every on-statement has non-empty to- and from-lists and (2) that every state is contained in at least one to-list and in at least one from-list.

6.3 Decomposition of the analysis

The goal of the analysis is to check whether the on-statements in a Simplexys rule base represent a Live Safe P/T-net. The checking algorithm is divided into tree stages: syntactic, topologic and dynamic checking. If errors are reported in some stage, further checking is abandoned.

1. Syntax checking

The protocol part of a Simplexys knowledge base is implemented by on-statements and state rules. To make the analysis faster, before the analysis the preset and the postset of every state and transition are computed from the on-statements, thus making them directly available in every step of the analysis.

In this stage a check for syntax errors also takes place: all pre- and postsets must be non-empty because that is a necessary condition for liveness and safeness (theorem 6.5). These algorithms are rather straightforward and not discussed in detail. Finally there is a check for one type of conflict: transitions with equal presets and equal trigger rules. Complete conflict checking is performed during dynamic checking.

2. Topologic checking

Checking whether the net is pure, simple and strongly connected is done in this stage. A non-pure net is likely to be neither live nor safe, but this is not certain: figure 6.2 gives two non-pure nets which are both live and safe. However, since non-pure Petri nets are quite tricky, warning messages are generated for every on-statement with common state rules in from- and to-list. In later stages of checking it will turn out whether or not such a net is live and safe.

A non-simple net element also generates a warning message only. Non-simple net elements affect liveness nor safeness, but the knowledge engineer might have made a mistake; and since the protocol can be simplified by combining equal net elements, warning messages are generated for net elements with equal pre- and postsets. These algorithms are derived directly from the definitions above, and are not discussed in further detail.

Finally a check for strong connectedness is made. Since strong connectedness is a necessary condition for liveness and safeness, a non-strongly-connected net will cause errors in the next stage, in which liveness and safeness are checked. By checking connectedness, these errors are reported in an earlier stage with more specific messages. The relevant algorithms are discussed in section 6.4.

3. Dynamic checking

To check for liveness, safeness and conflicts, the reachability tree is constructed. A reachable context is a set of active states; the reachability tree contains all reachable contexts. Checking for conflicts is described in more detail in section 6.5, and checking for liveness and safeness in section 6.6.

6.4 Analysis of strong connectedness

Strong connectedness is checked in the second stage of analysis (topologic checking). This check is redundant: protocols that are not strongly connected will cause deadlocks, transitions that cannot fire, or states that cannot get a token in the next (dynamic checking) stage. However, a message that some net part is not connected with an initial state or with a final state is more specific and thus more helpful than a message that a certain firing sequence results in a deadlock.

Strong connectedness means that there is a forward path from every state or transition to every other state or transition. Note that connectedness is defined for the extended net, in which the final states are connected with the initial states.

Checking the connectedness of a protocol is split up into two sub-checks. First it is checked whether a forward path exists from the initial state to every net element, second whether a forward path exists from every net element to the final state. The checking method is tuned to Simplexys protocols. These two sub-checks are necessary and sufficient for strong connectedness. There is a path from every arbitrary net element to the final state, and thus (in the extended net) to the initial state, and there is a path from the initial state to any arbitrary net element. Hence the extended net is strongly connected.

6.5 Simultaneous firing of transitions

A certain reachable context generally enables more than one transition. A transition is enabled if all the states in the from-list are active in that context.

Without any loss of generally, assume that two transitions are enabled by some reachable context. Two transitions are *independent* if their from-lists have no states in common. In figure 6.3a, transitions t1 and t2 are independent.

When trigger t1 is true, the transition will fire and similarly when trigger t2 is true, that transition will fire. These transitions can fire both, in any order, or even simultaneously, without interfering. During the construction of the reachability tree, t1 is fired first, and the resulting context (s2, s3, s4) is added to the tree. Second, t2 is fired from the original context (s1, s2, s3) and the resulting context (s1, s5) is also added to the tree (figure 6.3b). Because t1 and t2 are independent, the new context after firing t1 enables t2 and the context after firing t2 enables t1. Firing of t1 or t2 results in the same successor context (s4, s5) that is added to the tree. If at run time t1 and t2 are true at the same time, the intermediate contexts (s2, s3, s4) and (s1, s5) are not reached (but the checking algorithm cannot know that).

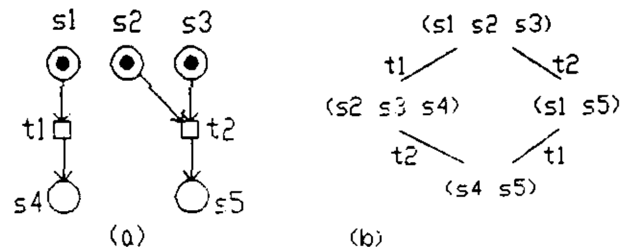

Figure 6.3
Independent transitions t1 and t2

Two transitions that are concurrently enabled are *dependent* if the intersection of their from-lists is non-empty. In figure 6.4a, both t1 and t2 are enabled. After firing t1, context (s3, s4) is added to the tree; this new context does not enable t2. Nothing is wrong with this: there is a *choice* to fire t1 or t2, but not both. Problems occur when triggers t1 and t2 are both true at run time; this is called a *conflict*. In Petri net theory, this is not allowed, and the resulting context is undefined. In Simplexys, the inference engine first fires t1 and then tries to fire t2 because that transitions was also enabled. However, it turns out that t2 is not enabled anymore and hence it is not fired.

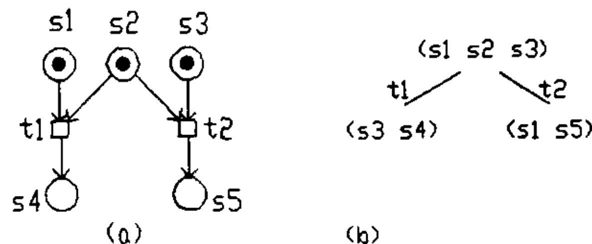

Figure 6.4
Dependent on statements t1 and t2

A net without any choice between transitions is deterministic: it is known beforehand which contexts will be active, and in which order. Because this is too restrictive for Simplexys, choices are supported, and generally choices do not cause conflicts. Checking for conflicts is possible during the construction of the reachability tree. Unfortunately the checker cannot always know whether the two trigger rules of the two transitions that are in choice will be true simultaneously at run time. However, in two cases the checker knows that a conflict is certain: if the two transitions refer to the same trigger rule and if the trigger rules are logically equivalent.

Trigger rules are logically equivalent if they refer to the same primitive rules in a logically equivalent way. For instance, *not* (a *and* b) is logically equivalent with *not* a *or not* b. Equivalence of trigger rules is checked by placing the *and* operator between the two trigger expressions, and evaluating this expression symbolically [Lutgens, 1990]. If the resulting expression equals true, the triggers are logically equivalent and a conflict is certain. If the expression results in false, a conflict never occurs. If the result is neither always false nor always true, certain circumstances could cause a conflict.

In summary, a conflict is detected when two or more on-statements satisfy the following conditions:

(a) The transitions are enabled at the same time *and*
(b) The intersection of the from-lists is not empty *and*
(c) The triggers of the on-statements are logically equivalent.

If two on-statements have equal from-lists so that (a) and (b) are satisfied, a conflict is certain if the triggers are logically equivalent (figure 6.5a). Such conflicts are detected in the first stage (syntax checking). It is known from liveness (which is checked in the third stage), that all transitions can fire at least once. Hence on-statements with equal triggers and equal from-lists cause conflicts. Figure 6.5b contains no conflict since condition (c) is not satisfied.

During the third stage (dynamic checking), conflicts are also checked. On-statements with equal from-lists are checked in the syntactic stage, on-statements with common states in their from-lists are checked for conflicts in the dynamic stage. The two transitions labeled y in figure 6.5c both have state s in their presets. Whether or not a conflict occurs depends upon the actual context: a conflict occurs only if s1 and s2 are both true.

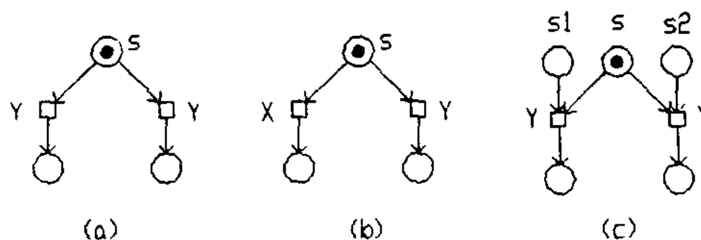

Figure 6.5
(a) Conflict with trigger x at state s
(b) No conflict at state s
(c) Conflict depends on s1 and s2

Two on-statements that refer to the same trigger rules and are simultaneously enabled but that are independent because the intersection of their from-lists is empty, do not cause conflicts. During the constructing of the reachability tree such transitions are fired simultaneously, and result in only one new context. The intermediate contexts that result if one of them is fired individually, is not generated. This method agrees with the way in which the Simplexys inference engine executes on-statements.

Because of the unknown relationships between triggers, there can be differences between the reachable contexts which are generated by the checker and the contexts that are reachable at run time. If two transitions fire simultaneously at run time because their triggers are both true, the intermediate contexts which the checker has generated are skipped. The checker fires transitions simultaneously only if the trigger rules are logically equivalent.

6.6 Analysis of liveness and safeness

The construction of the reachability tree has already been discussed briefly in the previous section. The reachability tree (definition 3.14) contains all possible state combinations or *contexts*; the root of the tree is the initial context. Generally the initial context enables more than one transition; firing one of these transitions results in a *successor context*, firing all enabled transitions from that context results in a set of successor contexts. Every reachable context is linked with its *successor* contexts, which results in a tree representation. In fact the "tree" is a *network* because a context might have more then one predecessor (figure 6.3b), but the term tree is established in Petri net theory.

When all transitions that are enabled by the initial context are subsequently fired, the initial context is *expanded* into a set of successors, which are added to the tree. Then the individual successors are expanded, in the same way as the initial context was expanded, and their successor contexts are added to the tree as well. This continues until every context in the tree has been expanded and no new contexts can be generated any more. When the tree is complete, it contains all possible state combinations.

A successor context is added to the tree only if it is not already in it. If two enabled transitions have equal triggers, these are fired simultaneously, as described in section 6.5.

Five kinds of errors can occur: three during the constructing of the tree and two more can be found when it is completed. The errors that can develop during the construction of the tree, are:

(1) A *deadlock* context is a reachable context that does not enable any transition. A deadlock context cannot be expanded into a set of successors. A net that comes to a deadlock is not live.

(2) A *non-safe* state is a state that gets a token due to the firing of a transition, while it already had a token before.

(3) A *conflict* occurs if two transitions fire simultaneously (because they are both enabled and have identical triggers) and have common input states. The occurrence of these conflicts has already been described in more detail in the previous section.

When the reachability tree is completed, the following two errors can occur:

(4) A context is *non-terminating* if no firing sequence from that context results in the final context. Once a non-terminating context is reached at run time, Simplexys can never stop. During the construction of the reachability tree, the contexts that can terminate are marked. This is described in more detail in section 7.5.

(5) Finally there is a check whether or not every transition has fired at least once.

Intuitively a Petri net that passes these five checks is correct, and indeed these checks are sufficient and necessary for liveness and safeness, the original checking goal. Check (3) does not play a role in liveness and safeness.

Four steps are taken to verify that the combination of checks (1), (4) and (5) is sufficient for *liveness* (definition 5.15b). First, it is clear from the preceding that the reachability tree contains all reachable contexts. Second, from the initial context every transition can fire at least once (5). Third, from every reachable context the final context is reachable (4). Fourth, the final context is connected with the initial context by a dummy transition *next patient* (liveness is defined for the extended net).

Liveness means that from every context a firing sequence exists so that any arbitrarily chosen transition can be enabled. Checking guarantees that for every reachable context a certain firing sequence results in the final context, and thus can result in the initial context. From the initial context every transition can fire. This is *sufficient* for liveness; when the net has passed checks (4) and (5), it is live. Note that check (1) does not play a role, since a deadlock context cannot terminate.

These checks are also *necessary* for liveness: if one of the checks (1), (4) or (5) results in an error, the net is not live. If a deadlock occurs (1), no transition is enabled; as a consequence the net is not live. If any transition can not fire at least once (5), the net is not live either. Finally, if there is a context from which the final context is not reachable, at least one of the transitions to the final context cannot be enabled.

Hence checks (1), (4) and (5) are both sufficient and necessary conditions for liveness. For *safeness*, check (2) is both sufficient and necessary since it covers definition 3.15a.

6.7 Methods that reduce the protocol's complexity

The reachability tree contains all possible state combinations. The larger the number of states, the larger the reachability tree. In the worst case, every new state that is added to the protocol can make the number of reachable contexts two times larger; the new reachability tree then contains all the original contexts twice: once when the new state is active and once when it is not active.

During dynamic checking, *reduction* (see below) might be necessary to prevent a blow up of the internal storage and to prevent checking from taking a very long period of time. Besides reduction of the Petri net, analysis of individual and independent *subnets* is possible. Reduction during syntactic or topologic checking is unnecessary because these checking stages are far less computation time and computer memory consuming than dynamic checking.

Reduction means that several states and transitions are deleted from the Petri net or are merged, without changing the relevant dynamic properties liveness and safeness. Two states can be *merged* without changing liveness or safeness if their presets as well as their postsets are equal. Two transitions can be merged if similarly conditions hold. Merging of net elements like this reduces the number of net elements, but does not reduce the number of reachable contexts.

The dynamic error messages that are reported contain the complete firing sequence that results in the context that causes the error. After merging states or transitions, the transitions of the firing sequence and the intermediate contexts are changed. Extra intelligence would then be necessary in order to report error messages that are relevant for the original protocol rather than for the reduced protocol. Merging of net elements is currently not implemented.

A second reduction method is *deletion* of net elements that are redundant to the checker. A state and transition pair (s, t) that is linked (s· = t) can be deleted if transition t has only one input state (s) and one output state (say s'), and state s has only one output transition (t) and one input transition (say t'). State s and transition t can be deleted if transition t' is directly connected to state s'; transition t and the intermediate state s can be skipped. Since both s and t have only simple connections, deleting s and t influences liveness nor safeness. Every state-transition pair that is deleted reduces the number of reachable contexts, because an intermediate state is skipped. In the firing sequence which is reported in case of an error, these intermediate transitions are also skipped, but the messages are still relevant for the original protocol. For protocols that contain many single linked state-transition pairs, this reduction method can decrease the number of reachable contexts considerably. This reduction method is not currently implemented either.

A more sophisticated method through which net elements can be merged, is the following. Two states s and s' can be merged if there is a single transition t between the two states (thus s· = t and t· = s') and moreover a single path of a set of transitions from s' back to s (thus s'· = t" and t"· = s" and so on until s^· = t^ and t^· = s). A new master state is defined in which all input and output transitions of s and s' are collected. Some additional conditions for merging are that s and s' have no common input and output transitions and have no different output transitions with equal triggers. Similarly two transitions can be merged. In a protocol that contains cyclic parts, this reduction method can decrease the number of reachable contexts dramatically. However error messages that refer to the reduced net are changed, and these are hardly relevant for the original protocol.

The preceding three reduction methods can be applied in turn repeatedly, until no reduction is possible anymore. Some protocols would reduce considerably, others hardly or not. Moreover, most of these reduction methods make the dynamic error messages less clear. The algorithms for reduction have been constructed and tested, but they have not been incorporated into the current version of the protocol checker.

Sequential analysis of individual *subnets* is a better and more appropriate method of reduction. A subnet is a part of the protocol that has no connection with the remainder of the protocol. Figure 5.2 contains two subnets. A subnet contains at least one initial state and at least one final state since the *extended net* must be strongly connected. In fact, the extended net is constructed in order to artificially *connect* the individual subnets and to make the net *cyclic*.

A subnet can operate independently of the remainder of the protocol, and thus dynamic checking can be performed for every subnet separately. The dynamic error messages are then related to one subnet only, and the firing sequence that is reported can easily be traced back by the knowledge engineer.

Since Simplexys offers the possibility to define more then one initial state, large protocols will generally contain several independent subnets. Sequentially checking the different subnets is usually much faster than checking the protocol as whole, because the total number of reachable contexts that is generated is often far less. This powerful reduction method has been implemented in the protocol checker.

7 Realization of the protocol checker

7.1 Introduction

In the previous chapter the checking for liveness and safeness was discussed at a high level; in this chapter the implementation of the checker's algorithms, especially the construction of the reachability tree, is discussed.

The Simplexys expert system is a *toolbox*: a set of programs to build real-time expert systems. The Petri net checker is one of these tools. Tools uses each other's data in a special way: the output of a tool is a set of files containing Pascal code, that is linked with the source code of the tool that needs that information. The file that is used by the Petri net checker contains the information about the protocol in terms of Pascal constant array definitions. The number of states and transitions is specified by constant definitions, so that the Petri net checker can define arrays that have the appropriate length.

During the first stage (syntax checking), the preset and the postset of every net element are derived from this description, and some syntactic checks are performed (section 7.2). Section 7.3 describes the implementation of the algorithm that checks for connectedness. Section 7.5 describes the construction of the reachability tree and section 7.6 describes the data structure that is used to store the reachability tree. Finally section 7.7 describes the way in which error messages are generated. In chapter 8 the error messages are explained in more detail.

7.2 The data structure

In the implementation of the checking algorithm, *sets* of states and transitions play an important role. The two set variables Swhole and Twhole contain all those states and transitions that are involved in a particular stage of checking. By changing Swhole and Twhole, a sub-protocol is checked; in that way dynamic checking is performed for every *subnet* separately. Initially Swhole and Twhole contain all net elements.

The set variable S0 is the initial context and contains all the states that are initially true. Similarly Tw contains the final transitions: all transitions that have a * in their to-list.

In the included file, the number of states and transitions is defined with the constants _N_S and _N_T. The Petri net is defined by three constant arrays to obtain the highest speed for the inference engine. For checking reasons, the preset and postset of every net element are derived from these arrays, and these are stored for later use.

As described in section 6.3 (1), during the first stage (syntax checking) a check is made for empty sets S0 and Tw and empty pre- and postsets. There is also a check for conflicts: on-statements that have the same from-lists and equal triggers. Other conflicts are detected during the construction of the reachability tree.

7.3 Checking of strong connectedness

As described in section 6.4, connectedness is checked in two steps. The first step starts at S0 and walks along downstream arcs (postsets) through the net; errors occur when not all states and transitions are passed through. The second step starts at Tw and walks along upstream arcs (presets) through the net; here too, errors occur when not all net elements are passed through. The two algorithms are similar, and only the first one is discussed in more detail.

Initially the set of connected states equals S0 and the set of connected transitions is empty. The first step is to add the postset of every state in S0 to the set of connected transitions; then, for every individual transition that is added, the states in the postset are added to the set of connected states. The second step is to add all transitions in the postset of the last added states and so on, until no more elements can be added. The scheme is given below:

```
Sconnected := S0
Tconnected := ∅
Sdelta := S0
repeat
   Salpha := ∅
   for all states i in Sdelta do
      for all transitions j in PostState(i) do
         Tconnected := Tconnected ∪ j
         for all states k in PostTrans(j) do
            if k not in Sconnected then
               Sconnected := Sconnected ∪ k
               Salpha := Salpha ∪ k
            fi
         od
      od
   od
   Sdelta := Salpha
until Sdelta = ∅
```

Now the sets Sconnected and Tconnected contain all the states and transitions that can be reached by downstream arcs from S0. The states and transitions that are not connected with S0 are easily found by comparing Sconnected and Tconnected with Swhole and Twhole.

The second algorithm starts at Tw and adds net elements that are in the presets, in a similar way. If all states and transitions are connected with Tw as well as with S0, the (extended) net is strongly connected; otherwise the net elements that are unconnected are reported.

7.4 Checking of liveness and safeness

Checking for liveness and safeness is the most complex type of checking. Many aspects are involved, such as administration of the reachability tree, simultaneous firing of on-statements with equal triggers, and error checking and reporting.

The worst case size of the reachability tree grows exponentially with the number of states, and thus for large protocols checking may become impossible due to the limited computer memory size. However, a large protocol is likely to contain several independent *subnets*. A subnet can be separately checked for liveness and safeness; it has no connection with the remainder of the protocol. For instance, the net of figure 5.2a contains two independent subnets: the left subnet contains one initial state, the right part two initial states.

A subnet contains at least one initial state (due to strong connectedness), and for every initial state a subnet can be generated. A subnet, that is generated, is stored in Swhole and Twhole which contain the states and transitions that are involved in dynamic checking.

The sets Swhole and Twhole are generated in a similar way as Sconnected and Tconnected above (section 7.3). The difference is that upstream as well as downstream net elements are added to Swhole and Twhole, rather than downstream net elements only. Stated differently, PostState (i) is changed in PostState (i) $\cup$ PreState (i); PostTrans (j) is changed in PostTrans (j) $\cup$ PreTrans (j).

Subnets that are generated for different initial states, can result in identical subnets. For instance, for figure 5.2 the subnet that is generated for the middle initial state is identical with the subnet that is generated for the right initial state. Sequentially, the *non-identical* subnets are checked for safeness and liveness. Checking for every subnet sequentially saves computation time and memory space, and error messages that are generated are more specific then if the Petri net is checked as whole.

The topic of section 7.5 is the generation of the reachability tree for a certain subnet, and the detection of deadlocks, non-safe states and conflicts. The tree itself is treated as a black box, which supports four functions: (1) store a newly generated context, (2) check whether a particular context is in the tree, (3) obtain an old context that is not yet expanded into its successor contexts and (4) trace back to the initial context and obtain the firing sequence that results in a particular context. It is important to notice that only one firing sequence resulting in a context can be stored. These four basic tree operations are implemented in four procedures which are described in more detail in section 7.6

7.5 Construction of the reachability tree

The root of the reachability tree is the initial context of the subnet that is to be checked. When the tree is completed, every element of the tree is linked with its *successor* contexts. In fact the "tree" is a *network* because a context might have more then one predecessor, but it is implemented as a *list*. The algorithm that constructs the reachability tree is:

(0) PutIntoList (S0 ∩ Swhole)
(1) *while* GetFromList (Sold) *do*
(2)   T := Enabled (Sold)
(3)   *for* all transitions j *in* T *do*
(4)   Snew := Fire (Sold, j, T)
(5)   *if not* InList (Snew) *then* PutIntoList (Snew) *fi*
(6)   *od*
(7) *od*
(8) check whether every transition has fired
(9) check whether all reachable contexts can terminate

(0) The initial states (S0) that are in the subnet that is currently being checked (Swhole) form the initial context, which is the root of the tree.

(1) The tree delivers a non-expanded context Sold. The first call to GetFromList delivers the initial context. As soon as all contexts in the tree have been expanded, GetFromList returns false, indicating a completed tree.

(2) T is the set of transitions that are enabled by Sold. If no transition is enabled, Sold is a deadlock context, and an error message is generated.

A transition is *enabled* if all the states in its from-list are true; the set of enabled transitions is determined by the context Sold and the structure of the Petri net. It is easy to find the enabled transitions by scanning through all transitions, and check if their preset is contained in Sold. However, this is not the most efficient way: it is faster to scan only through the transitions that are connected to Sold, and check if these are enabled:

```
scheme of  T := Enabled (Sold):

T := ∅
for all states i in Sold do
  for all transitions j in PostState(i) do
    if PreTrans(j) ⊆ Sold then
      T := T ∪ j
    fi
  od
od
if T = ∅ then
  ReportDeadlock (Sold)
fi
```

(4) *Firing* a transition j in T results in a new context Snew. If individual transitions in T have identical triggers, these are fired simultaneously:

```
scheme of  Snew := Fire (Sold, j, T)  without error detection:

Ssub := PreTrans(j)
Sadd := PostTrans(j)
for all transitions jj in T (except j) do
  if jj has the same trigger as j then
    Ssub := Ssub ∪ PreTrans(jj)
    Sadd := Sadd ∪ PostTrans(jj)
  fi
od
Snew := (Sold \ Ssub) ∪ Sadd
```

This scheme describes the firing of one transition j in the set of enabled transitions T. If there are no enabled transitions with the same trigger as j, only the first two and last lines are executed; the result of firing a transition is that all states in the preset are made false and all states in its postset are made true.

After firing j, the other transitions that are enabled and have triggers that are textually or logically equivalent, are fired. The preset and the postset of the transitions that fire, are put in the sets Ssub and Sadd respectively; eventually Ssub and Sadd will be the unions of the to-lists resp. the from-lists of all enabled on-statements that have equivalent triggers. The new context results from the old one by subtracting Ssub and adding Sadd, in that order. The order is important because one transition can make a state false, while another transition can make that state true.

If firing transition jj adds a state to Ssub that is already in Ssub, a *conflict* occurs: transition jj was enabled by Sold, and needs an active state for firing, that has already been made inactive by another transition (which was also enabled by Sold). The latter transition will not fire and will not update Ssub and Sadd (see the scheme below).

If firing transition jj adds a state to Sadd that is already in it, a *multi-token* or *non-safe state* has been detected: two transitions fire simultaneously, and both put a token on the same state. In such cases, the latter transition also fires and Ssub and Sadd are updated.

When Ssub and Sadd have been computed, the new context is derived from the old one by subtracting Ssub and adding Sadd. Then non-safe states can also be detected, as the scheme below shows. The set Sxx contains the states that were true before firing and that will stay true because they are not influenced by firing. The intersection of Sxx and Sadd are the states that were true before firing and are made true again by firing. Hence if the intersection is not empty, it contains non-safe states.

When the error detection strategies described above are included, the scheme for firing a transition is:

```
scheme of  Snew := Fire (Sold, j, T)   (with error detection):

Ssub := PreTrans(j)
Sadd := PostTrans(j)
for all transitions jj in T (except j) do
  if jj has the same trigger as j then
    if PreTrans(jj) ∩ Ssub ≠ ∅ then
      ReportConflict (Sold, jj)
    else (fire jj)
      Ssub := Ssub ∪ PreTrans(jj)
      Sns := PostTrans(jj) ∩ Sadd
      if Sns ≠ ∅ then
        ReportNonSafe (Sold, Sns, jj)
      Sadd := Sadd ∪ PostTrans(jj)
    fi
  fi
od
Sxx := Sold \ Ssub
Sns := Sxx ∩ Sadd
if Sns ≠ ∅ then
  ReportNonSafe (Sold, Sns, j)
fi
Snew := Sxx ∪ Sadd
```

(5) In step (4) one successor context Snew is generated by firing a transition (or a set of enabled transitions with the same trigger). Context Snew is one of the successors of Sold, and it is added to the reachability tree if it is not already in it.

(6) Back to step (4). Another transition that is enabled by Sold is fired, and its successor context is added to the tree in step (5). When all transitions that are enabled by Sold have been fired, Sold is expanded into a set of successor contexts that are stored into the reachability tree.

(7) Back to step (1). A new non-expanded context is taken from the tree, and it is expanded in steps (2) .. (6) into a set of successors. As soon as all contexts have been expanded, the reachability tree is complete and GetFromList returns false.

(8) During the construction of the reachability tree, every transition that fires is put into a set; when the tree is completed, this set must be equal to Twhole. If not, there are transitions that cannot fire at least once, and an error message is reported.

(9) When the reachability tree is completed, for every reachable context it is checked whether a firing sequence can result in the final context. During the construction of the tree, some administration is done to achieve this. Every context in the tree contains a *flag* that is set if that context can *terminate* because there is a firing sequence (a path in the tree) from that context to the final context. The original (first) scheme of this section is changed to implement this:

```
(0) PutIntoList (S0 ∩ Swhole)
    repeat
(1)     while GetFromList (Sold) do
(2)       T := Enabled (Sold)
(3)       for all transitions j in T do
(4)          Snew := Fire (Sold, j, T)
            if Snew = final context then (i)
               SetFlags (Sold)
            else
(5)              if not InList (Snew) then (ii)
                   PutIntoList (Snew)
                   SetFlags (Sold)
                 else
                   if Flag (Snew) then (iii)
                      SetFlags (Sold)
                   else (iv)
                      store Sold as not expanded
                   fi
                 fi
            fi

(6)       od
(7)     od
    until no flags set this pass
(8) check whether every transition has fired
(9) check whether all reachable contexts have an flag set
```

When a transition fires from context Sold in step (4), there are four possibilities for the successor context Snew.

(i) The successor context is the final context. The flags of Sold and its predecessors back to the initial context are set: for all these contexts it is known now that they can terminate.

(ii) The new context Snew is not yet in the tree. Snew is a newly generated reachable context, and it is added to the tree, but without its flag set.

(iii) The successor Snew is already in the tree, and moreover the flag of Snew is set. Then also the flags of Sold and its predecessors are set: as in the first case, for all these contexts it is known now that they can terminate.

(iv) The new context Snew is already in the tree, but its flag is not set: it is not decidable yet whether or not Snew can terminate, and thus it is undecidable for Sold as well. The context Snew has two predecessors: the first because it was already in the tree, and the second is Sold. It is likely that the flag of Snew will be set later, because one of its successors can terminate. Then also the flags of both predecessors and their paths back to the initial context must be set.

However, the tree is implemented in such a way that only one predecessor is stored (it would take too much memory space to store an unknown number of predecessors for every context). The fact that Snew is a predecessor of Sold cannot be remembered; instead Sold is marked as a not yet completely expanded context. When the reachability tree is completed, a second pass (actually a repeat-until loop) is necessary to check, for every non-expanded context, whether or not one of its successors has a flag. Note that during the second pass all successors of Sold are already in the tree. If one of the successors of Sold has a flag now, Sold and its predecessor's flags are set also; if not, it is still not decidable whether or not this context can terminate.

When the second pass is completed, three possibilities exist. First, no contexts without a flag are left; thus all contexts can terminate. Second, not all contexts have a flag, but during the last pass some flags were set; then another pass is provided in an attempt to add more flags. Third, not all contexts have a flag, and no flags were added during the last pass; the contexts that still have no flag, cannot terminate, and are reported as such.

7.6 The data structure of the reachability tree

In the previous section, the reachability tree was treated as a black box which offers four function: check whether a context is in the tree, add a context to the tree, get a non-expanded context from the tree, and produce the predecessors of a context.

The data structure of the tree can be implemented in different ways. The term *reachability tree* is established in Petri net theory, but in fact the reachability tree is a *network* because a context can have more than one predecessor. Storing the tree as a network would require a large amount of memory space: for every context an unknown number of addresses of predecessors must be stored (addresses of successors should not be stored, because successors can be re-generated if necessary). In order to obtain some example firing sequence that results in an error context (necessary for error messages), storing one predecessor is sufficient. As described in the previous section, for setting the flags for the contexts that can terminate, storing of one predecessor is also sufficient. The drawback of saving memory space is a longer computation time.

The reachability tree is thus stored as a singly *linked list*, in which the link points to a predecessor context. Besides memory space, *searching time* must be minimized. There are two types of searching in the checking process: for a non-expanded context and whether or not a context is in the tree. Searching is a real bottleneck since even for a relatively small protocol, hundreds or thousands of different reachable contexts can be generated.

Searching in a *linear list* is far from efficient since on average half the list must be scanned through to know whether a context is in the list. In a *tree* representation, as in figure 6.2, a non-expanded context is easily found, but finding out whether or not a context is in the tree takes as much time as in a linear list.

A better approach is a *binary tree* in which every level represents a state. For every reachable context there is a path along the levels; when a state is in the context, the path moves to the right at that level, else to the left. To find a context in the tree, as many steps as the number of states are necessary; the number of steps can easily be ten or more, but this method is far more efficient than the previous ones.

However the most efficient way to find out whether a context is in the tree, is a *direct organization*: a large store contains boxes that can contain one context each. The box number is called the *address*; the address where a context is stored depends completely on the pattern of the states that are true (if the state is in the context) or false (not in the context). In principle every possible combination of true and false states has a box where that context is stored (in fact a flag that indicates whether or not that context is in the box is sufficient). Some of the boxes represent a reachable context, others not. To search whether a context is in the tree, its address is computed, and its box is inspected. However, the implementation is somewhat more complicated.

Generally there will be thousands of possible state combinations, but only a few hundred may be reachable. When the protocol contains 20 states, one million (2^{20}) state combinations are in principle possible. Since such a large store is not available, several state combinations are *mapped* onto the same address. Mapping of a state combination to a box address is called *hashing*.

Due to hashing, more than one possible context is mapped onto the same box address. Although most contexts are not reachable, a context that is added to the tree can find its box occupied by another context. Then hashing is performed again, which results in another address; if that box is also occupied, hashing is performed again and so on. After 5 unsuccessful trials, the context in stored in an overflow list.

Searching for a context is performed in a similar way: hashing is performed and there are three cases. First, if the box contains the context, it has been found. Second, if the box is empty, that context is not in the tree. Third, if the box contains another context, hashing must be performed again and again as described above, until the context is found or the box is empty; finally a scan through the overflow list may be necessary.

Storing a new context and searching for a context is performed quickly now, but two more functions need to be performed. First, the tree must deliver a non-expanded context. If only a flag marks whether or not a context is expanded, a scan through part of the list will be necessary to find a non-expanded context. Especially when the tree is almost completed, it takes much time to find a non-expanded context by that method. A faster method is to administrate a list of occupied box addresses; the lower part of the list contains the expanded contexts, the upper part the non-expanded contexts. The address of a non-expanded context is found just above the border, and when the context is expanded the border moves one position upward. A newly generated reachable context is added to the tree by storing its address at the top of the list.

Finally, the tree must produce a predecessor of a context, and a sequence of transitions from the initial context to that context. The predecessor is used to add flags when a context can terminate, the firing sequence is used for error messages only. Every context in the tree contains the address of one of its predecessors. The firing sequence is obtained by tracing back to the initial context, and by determining at every step which transition caused that context switch.

7.7 Error messages

During dynamic checking, four types of errors are checked for: deadlocks, non-safe states, conflicts and non-terminating contexts. Four routines produce appropriate error messages; these are ReportDeadlock, ReportConflict, ReportNonSafe and ReportNoEnd. Every error message contains the complete firing sequence that leads to the context that causes the error. A complete description of the error messages is given in chapter 8.

To prevent the same error from being reported several times, a deadlock context is reported only if none of the states in that context was contained in an earlier reported deadlock context. The same method is used for non-terminating contexts. A similar method prevents multiple error messages to be generated for non-safe states and for conflicts.

8 The user interface of the protocol checker

8.1 Introduction

When a Simplexys application is created, first the rule base is composed with a text editor, which is then translated by the Simplexys rule compiler, which checks for *syntactic* correctness. Second, *semantic* analysis is performed by the Simplexys checker, which includes the protocol checking that is the subject of this document. If the knowledge base is semantically correct, the translated rule base is linked with the Simplexys inference engine, which results in a runnable real time expert systems.

Semantic analysis is performed to assist the knowledge engineer in the validation of the rule base, to report doubtful and/or wrong constructions, and to prevent run time errors. Since the difference between a doubtful construction and an error is not very strict, the checker does not prevent the building of the expert system if errors have been detected. For the protocol, the most serious error is a global system stop due to a *deadlock*: a context from which no transition is possible. The occurrence of a *non-terminating* context is also a serious error, because from such a context the final context cannot be reached, and hence Simplexys cannot stop.

In section 8.2, the resemblance between Petri nets and the on-statements of a knowledge base is studied. Firing of a transition in a Petri net agrees completely with the execution of an on-statement.

Checking of the Petri net is done in three stages. Section 8.3 explains the additional *syntax* checking, section 8.4 the *topologic* analysis and section 8.5 the analysis of the *dynamic* behavior. Section 8.6 illustrates the properties of a correct protocol, and section 8.7 the *run time errors* that might occur during the execution of an incorrect protocol. Finally section 8.8 briefly recalls and explains the messages which are reported in case of errors.

8.2 A protocol represents a Petri Net

Petri nets [Reisig, 1985] are a graphic representation of the dynamic behavior of a system. Petri net theory supports analysis of processes that are modelled by a Petri net. As described in chapter 4, the on-statements of a Simplexys knowledge base represent a Petri net: a state is drawn as a circle and the trigger of an on-statement as a square or a bar.

If a state is active (true), it is marked with a *token*; the set of active states is called the *context* or the *marking*. States that are initially true are called *initial states*; the set of initial states is the *initial marking* or the *initial context*.

States that are represented by a * are called *final states*; a context where only final states are active is called the *final context* (Simplexys stops when the final context is reached).

The syntax of an on-statement or a *transition* in the corresponding Petri net, is:

on trigger *from* from-list of states *to* to-list of states

The *trigger* can be a rule of any type, for instance an ask rule or an evaluation rule. The *from-list* and the *to-list* contain *state rules*. A transition is *enabled* if all the states in the from-list are active. Then the trigger rule is evaluated, and if it evaluates to true, the on-statement is executed: this makes the states in the from-list false and the states in the to-list true.

Different transitions can have the same trigger; in figure 8.1, the first and the last on-statement have the same *trigger*, but represent different *transitions*. The graphic representation of a set of on-statements with a Petri net is straightforward:

On-statements:

s1 and s3 are defined as *initially true*

on t1 *from* s1 *to* s2
on t2 *from* s3 *to* s6
on t6 *from* s2 s6 *to* s7
on t3 *from* s3 *to* s4
on t4 *from* s4 *to* s5
on t5 *from* s5 *to* s6
on t1 *from* s7 *to* *

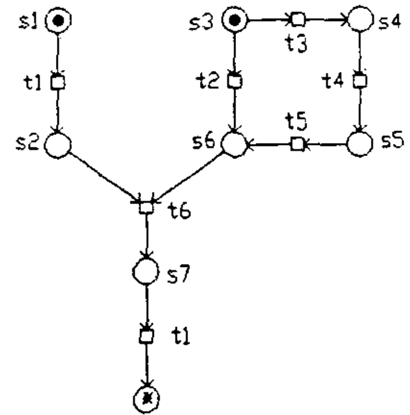

Figure 8.1
Graphic representation of on statements

Every transition has some *input* or *upstream* states and some *output* or *downstream* states; tokens in a Petri net flow in the direction of the arcs, starting at the initial states and disappearing at the final states.

A transition is *enabled* when every input state is marked with a token; in figure 8.1, the transitions with triggers labeled t1, t2 and t3 are enabled. For every enabled transition, the inference engine evaluates the related trigger; if the trigger is false nothing happens, otherwise the on-statement *fires* and a *context switch* occurs: all input states loose their tokens, all output states get one. A context is *reachable* if some *firing sequence* from the initial context results in that particular context.

So far the correspondence between Simplexys protocols and Petri nets. Note the difference between an *enabled* transition and a *firing* transition. Chapter 2 describes the firing of transitions in a Petri net in more detail.

8.3 Syntax checking

The rule compiler performs some syntax checking of the rule base, and thus also syntax checking of the protocol. The protocol is syntactically correct if:

[1] Syntax checking by the rule compiler

a. Every on-statement has non-empty from- and to-lists.
b. Every rule used in every from- and to-list is a state-rule.
c. At least one state is initially true.
d. No on-statement has a * in its from-list.

When it finds the knowledge base to be syntactically correct, the rule compiler produces several files that are used by the semantic checker. Semantic protocol checking is performed in three stages; this section explains the first stage of checking, which performs some additional syntactic checks. The next two sections are about the two other stages: topologic and dynamic checking. Syntax checking is mainly analysis of the *completeness* of the protocol. The syntax errors that can be reported, are:

[2] No final state

If the protocol does not contain any on-statement with a * in its to-list, Simplexys can never stop (figure 8.2a). The error message that is reported is:

** ERROR No ON statement has an empty TO list (for figure 8.2a)

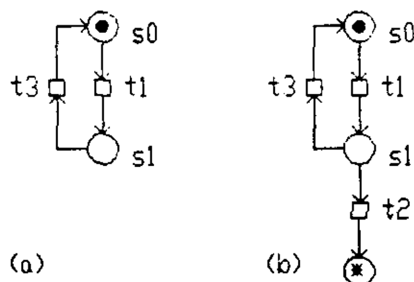

Figure 8.2
(a) A net without an end-state
(b) A correct net

[3] Conflict between transitions

Generally a context enables more than one transition. For instance, in figure 8.1 the transitions t1, t2, and t3 are concurrently enabled and Simplexys will evaluate the corresponding trigger rules. Concurrently enabled transitions are either *dependent* or *independent*. The transitions t1 and t2 can fire independently: if t1 fires, t2 is still enabled and the other way around. On the other side t2 and t3 are dependent: if t2 fires, t3 is not enabled anymore and the reverse (see also sections 2.3 and 3.4). There is a *choice* either to fire t2 or t3, but not both. Normally a protocol will contain several choices, which are necessary to describe non-deterministic behavior.

A *conflict* occurs when the triggers t2 and t3 both evaluate to true at the same time: then it depends upon the textual order of the on-statements in the rule base which of them will fire. Generally it is not known whether t2 and t3 can become true at the same time, However, if two transitions that have equal from-lists refer to the same trigger rule (figure 8.3a) or have logically equivalent triggers rules, a conflict is certain and a message is reported:

** ERROR Conflict at STATE s0 (for figure 8.3a)
ON x FROM s0 TO s1
ON x FROM s0 TO s2

During dynamic checking, conflicts can also occur (section 8.5 [16]). The net in figure 8.3b is correct: triggers x and y occur in more than one on-statement, but there is no conflict. The problems in figure 8.3a occur because the on-statements with the same trigger also have equal from-lists. If the knowledge engineer intended to give both s1 and s2 a token after firing x, figure 8.3c illustrates the correct net:

ON x FROM s0 TO s1 s2 (for figure 8.3c)

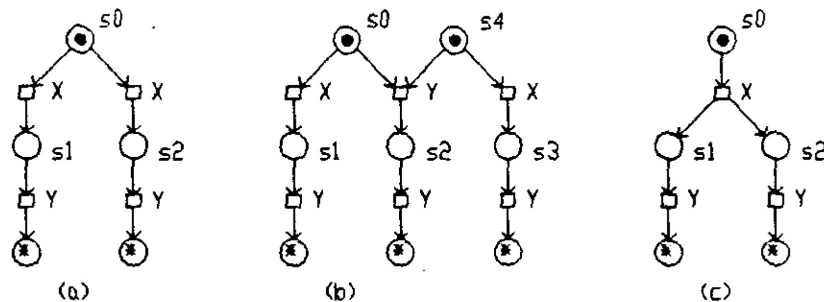

Figure 8.3
(a) Conflict occurring at s0 with trigger x
(b) No conflict
(c) Both s1 and s2 get a token when x fires

[4] States without upstream transitions

Every state (except initial states) must have at least one upstream transition. Stated differently: every state is in at least one to-list. If not, that state can never become true (figure 8.4a) and the following error message is reported:

** ERROR Unreachable STATE (not in any TO list) s2   (for figure 8.4a)

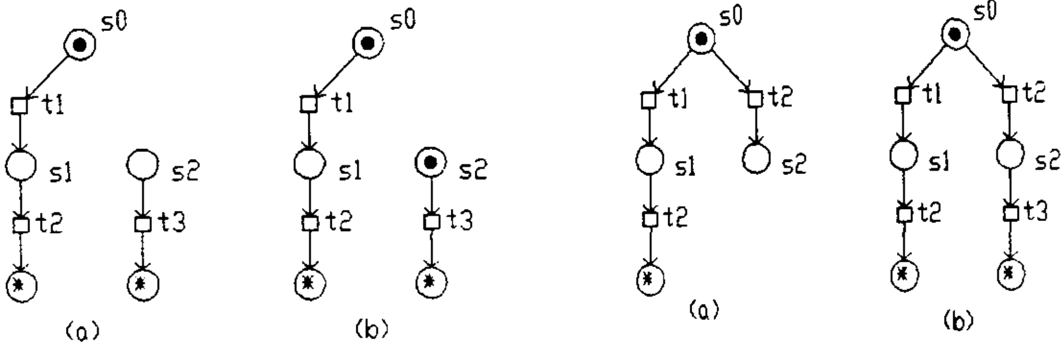

Figure 8.4
(a) s2 has no upstream transitions
(b) A correct net

Figure 8.5
(a) s2 has no downstream transitions
(b) A correct net

[5] States without downstream transitions

If a state has no downstream transition, once true it can never become false (figure 8.5a). The error message is:

** ERROR Dead-end STATE (not in any FROM list) s2   (for figure 8.5a)

If any error is found in one of the tests [2] to [5], further checking is aborted: for a net that is not syntactically complete, lots of error messages would be reported the next two stages, and the final conclusion about correctness is known already.

8.4 Topologic checking

After syntax checking has verified the completeness of the protocol, topologic checking verifies whether states and transitions are connected correctly.

[6] Self loops

Normally the from-list and the to-list of an on-statement are disjunct. If not, the from- and to-list contain common states. A self loop is a doubtful construction. When transition t2 in figure 8.6a fires, state s2 is made true and cannot become false. Figure 8.6b also contains a self loop: s2 must be true to enable t1, but to make s2 true, t1 must fire. Protocols that contain a self loop are likely to cause dynamic errors (a deadlock or a non-safe state) but this is not certain: see figure 6.2 for nets that contain a self loop but are correct. For self loops, warning messages are reported:

```
** WARNING  Self loop          (for figure 8.6a)
ON  t2  FROM  s2  TO  s2 s3

** WARNING  Self loop          (for figure 8.6b)
ON  t1  FROM  s1 s2  TO  s2
```

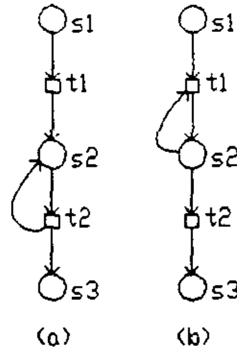

Figure 8.6
(a) Self loop at t2
(b) Self loop at t1

[7] Identical on-statements

Two on-statements are identical if their from-lists are equal and their to-list are equal as well. Since identical on-statements affect liveness nor safeness, this is not an error. Because identical transitions can be merged, as figure 8.7 shows, warning messages are reported:

** WARNING Identical ON statements (for figure 8.7a)
ON t1 FROM s1 TO s2 s3
ON t2 FROM s1 TO s2 s3

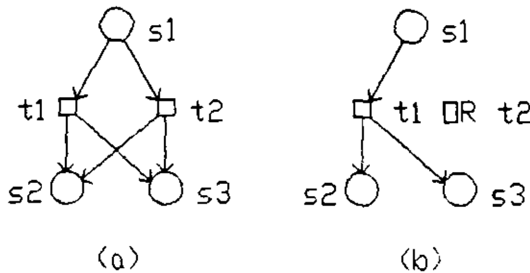

Figure 8.7
(a) Identical on-statements t1 and t2
(b) Merging of identical on-statements.

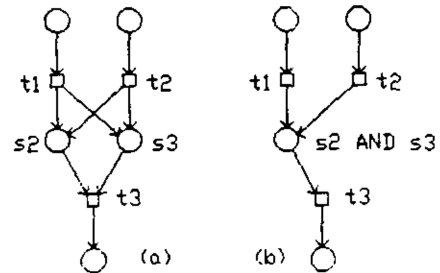

Figure 8.8
(a) Identical states s2 and s3
(b) Merging of identical states

[8] Identical states

States are identical if they have the same upstream and the same downstream transitions. As for identical transitions [7], only a warning message is given because identical states can be merged as figure 8.8 shows:

** WARNING Identical STATES s2 s3 (for figure 8.8a)

[9] States that are not connected with an initial state

There must be a path from an initial state to every other state. States that are not connected with an initial state can never get tokens. The net of figure 8.9a is corrected when either s1 or s2 is defined initially true, or when the two subnets are connected, as shown in figure 8.9b. Notice that the net of figure 8.9a does not produce error message [4].

** ERROR No connection with an initial STATE (for figure 8.9a)
ON t2 FROM s2 TO s1
ON t3 FROM s1 TO s2
ON t4 FROM s2 TO *

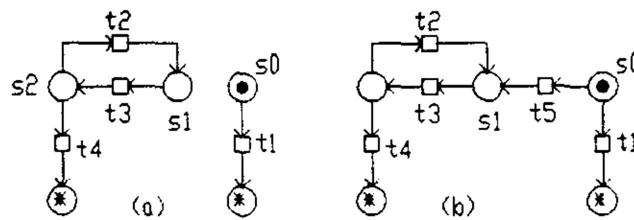

Figure 8.9
(a) There is no forward path from a start-state to s1 and s2
(b) All states are connected with a start-state

[10] States that are not connected with a final state

Every state must have a forward path to a final state (marked as *). States that are not connected with a final state cannot loose their tokens (figure 8.10a).

** ERROR No connection with a final STATE (for figure 8.10a)
ON t2 FROM s2 TO s1
ON t3 FROM s1 TO s2

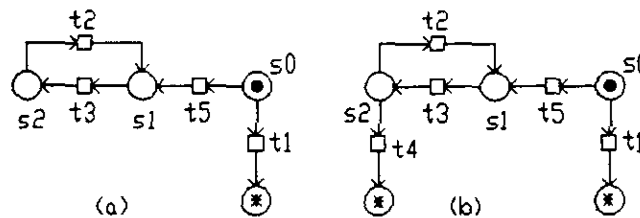

Figure 8.10
(a) States s1 and s2 are not connected with an end-state
(b) All states are connected with an end-state

If the states and the transitions are connected in a correct way, the net is topologically correct; the next stage will analyze the dynamic behavior.

8.5 Checking of the dynamic behavior

The final stage of checking is an analysis of the protocol's dynamic behavior. Checking is based upon the *reachability tree*, which contains all *reachable contexts*. During its construction, three types of errors can occur: a deadlock context, a non-safe state and a conflict context.

A *deadlock context* is a reachable context from which no change of state is possible; at run time Simplexys will stop at a deadlock context. A *non-safe* or *multi-token* state is a state that is true in a certain context, and made true again by the execution of an on-statement. A *conflict* occurs if two enabled transitions, that must fire simultaneously because of their equal triggers, cannot fire both because they have common states in their from-lists.

Three more errors are detected as soon as all reachable contexts have been generated: transitions that did not fire at least once, protocols that can never terminate, and contexts from which the final context is not reachable.

[11] Deadlock

A deadlock is a context that does not enable any transition. If a deadlock is detected, some firing sequence results in that context from which no further context switches are possible. In figure 8.11a, at the initial marking transition t1 or t2 can fire, so that either s1 or s2 gets a token. This token flows to s3 or s4, but not both states will get a token simultaneously, which is necessary to let t5 fire. The following error message is reported in that case:

```
** ERROR  Deadlock at CONTEXT  s3           (for figure 8.11a)
 1 ON  t1  FROM  s0  TO  s1
 2 ON  t3  FROM  s1  TO  s3
** ERROR  ON statement cannot fire
ON  t5  FROM  s3 s4  TO  s5
```

First the deadlock context is given, and the firing sequence that results in that context. Then a transition is given that is connected with the deadlock context, but cannot fire (this line is omitted if it is not clear which on-statement should be reported). The net shown in figure 8.11a can be made deadlock-free if s3, s4 and s5 are combined correctly, for instance as figure 8.11b shows.

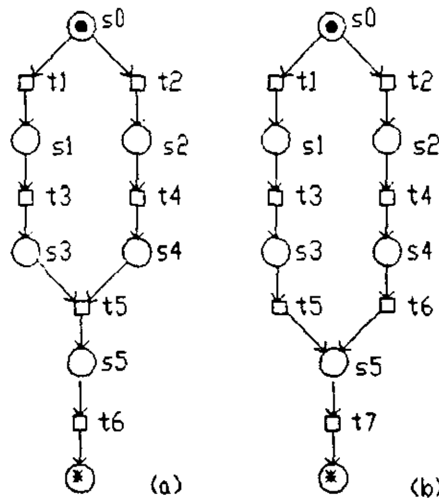

Figure 8.11
(a) A deadlock at t5
(b) A correct net

[12] Non-safe state

A state is non-safe if it is made true by a transition that fires, while it was already true before firing (and is not made false by another transition that fires simultaneously because it has the same trigger). In figure 8.12a, transition t1 fires, so that both s1 and s2 get a token; these tokens flow to s3 and s4. Then both t5 and t6 are enabled; firing one of them makes s5 true, firing the other transition makes s5 true again, which will produce the following message:

```
** ERROR  Multi-token STATE  s5          (for figure 8.12a)
  1 ON  t1  FROM  s0  TO  s1 s2
  2 ON  t3  FROM  s1  TO  s3
  3 ON  t4  FROM  s2  TO  s4
  4 ON  t5  FROM  s3  TO  s5
** ERROR  Extra token by ON statement
ON  t6  FROM  s4  TO  s5
```

First the state that gets two tokens is specified. Second, the firing sequence which gives that state the first token is given. The last line gives the on-statement that is enabled by that context and that gives the state the second token when it is executed. At run time, these two tokens will merge. Stated differently: a state that is made true, while it was already true, stays true. The net of figure 8.12a can be made safe if s3, s4 and s5 are combined in a correct way, for instance as figure 8.12b shows.

The Petri net shown in figure 8.12c is also correct (but tricky): state s3 is not a multi-token state because when y fires from context (s2, s3), first the states s2 and s3 are made false, and then states s3 and * are made true.

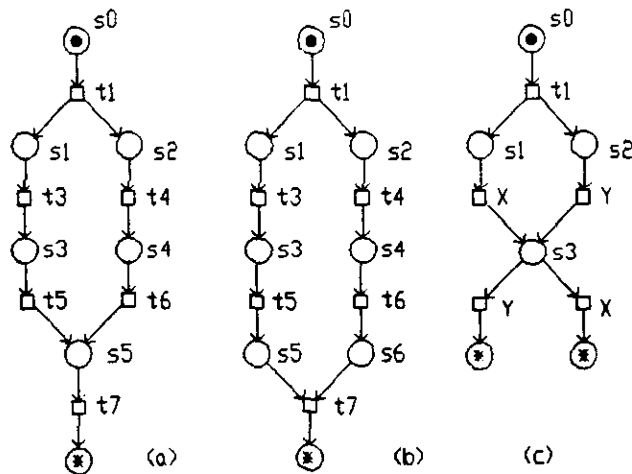

Figure 8.12
(a) State s5 gets two tokens
(b) A correct net
(c) A correct net (but tricky)

[13] Protocol can never terminate

A protocol cannot terminate if no firing sequence results in the final context (the context in which only * states are true); if such a protocol is executed, Simplexys will never halt. The net of figure 8.13 shows a protocol that does not come to a deadlock, but cannot terminate either.

** ERROR System cannot halt (for figure 8.13)

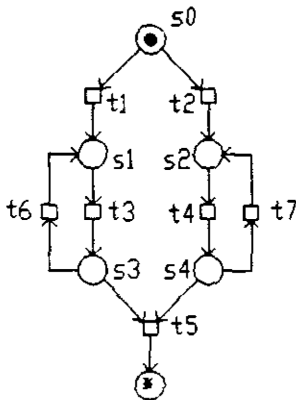

Figure 8.13
This net cannot stop

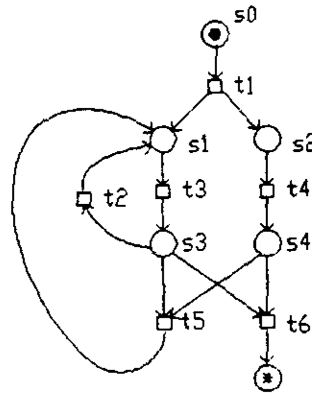

Figure 8.14
System cannot stop in context s2

[14] System cannot always terminate

The difference between this error and the previous one is that the net in figure 8.14 can terminate after the firing sequence t1, t3, t4, t6; thus error [13] is not reported. No deadlock occurs either, and all transitions can fire. However, the protocol cannot *always* terminate: after the firing sequence t1, t3, t4, t5 no sequence of transitions can results in the final context.

** ERROR System cannot halt in CONTEXT s1
ON t1 FROM s0 TO s1 s2
ON t3 FROM s1 TO s3
ON t4 FROM s2 TO s4
ON t5 FROM s3 s4 TO s1

This message specifies the non-terminating context, and gives the firing sequence that results in that context.

[15] Transitions that cannot fire at least once

Figure 8.15a gives a net that does not come to a deadlock, in which the final context is always reachable, but in which not all transitions can fire at least once. The error message that reports such a situation is:

** ERROR Transitions can never fire (for figure 8.15)
ON t4 FROM s1 s3 TO *

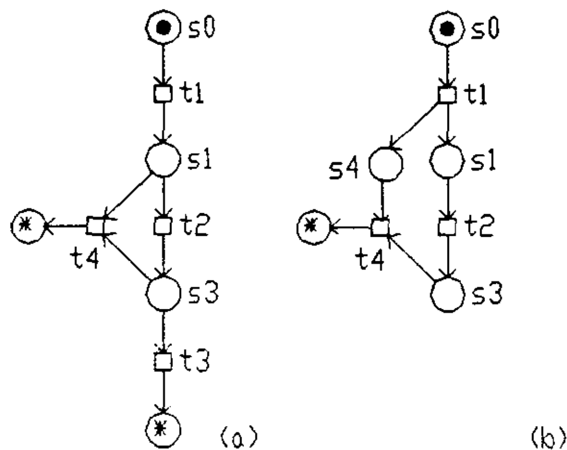

Figure 8.15
(a) Transition t4 cannot fire
(b) All transition can fire at least once

[16] Conflicts

Some conflicts have already been detected during the first stage (syntax checking), others are detected in this stage. The first type of conflicts (check [3]) occurs when two on-statements that have equal trigger rules also have the same from-lists. Note that from test [15] it is known that every transition can fire.

When two transitions have equal triggers and their from-lists are not equal but contain common states, a conflict *might* also occur when the on-statements are concurrently enabled, as shown in figure 8.16a. Context (s1, s2) enables both transitions with trigger x, and these transitions must necessarily fire simultaneously. However, when the first one has fired, the second one cannot fire because s2 has already been made false by the first firing. This error is *not* detected earlier in the first stage (syntax checking).

```
** ERROR  Conflict at CONTEXT  s1 s2          (for figure 8.16a)
 1  ON  y  FROM  s0  TO  s1 s2
** ERROR  Conflicting ON statements
ON  x  FROM  s1 s2  TO  *
ON  x  FROM  s2  TO  s3
```

First the context that enables the conflicting on-statements is specified, and the firing sequence that results in that context is given. Finally the conflicting on-statements are reported. The net of figure 8.16b contains three conflicting on-statements labeled x at context (s1, s2).

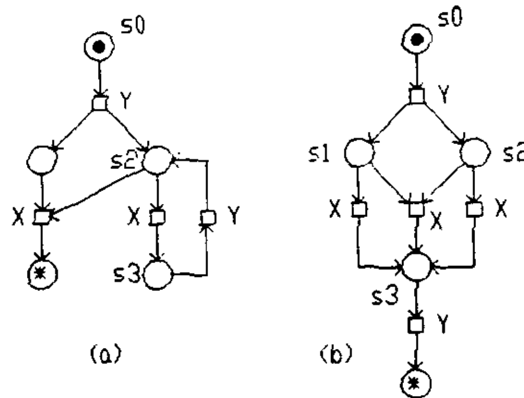

Figure 8.16
(a) A conflict between two transitions with trigger x
(b) A conflict between three transitions with trigger x

8.6 Properties of a correct protocol

If no errors are detected in the third stage of checking, the protocol is correct: it represent a conflict-free, live and safe Petri net. Such a protocol has some desirable properties. First, it will not come to a deadlock, a context from which no further context switch is possible. Second, independently of the sequence of on-statements that will be executed, the protocol can always terminate. Third, conflicts between two or more enabled on-statements will not occur and fourth, the execution of an on-statement will never make a state true that was already true before.

However, these properties are valid for the isolated protocol only; the protocol checker has no deep semantic knowledge about the remainder of the rule base. Whether or not a correct protocol can terminate at run time depends upon the semantics of the trigger rules. In other words: the *structure* of the Petri net makes it always possible to continue and to terminate after an arbitrary firing sequence, but whether this occurs at run time depends upon the actual values of the trigger rules.

The checker does not know the exact behavior of the trigger rules. Only if conflicts might occur, does the checker employ some extra effort to analyze the logical equivalence of two trigger rules.

8.7 Properties of an incorrect protocol

Because the checker cannot forbid building a runnable expert system, it is possible to construct an expert systems which contains a protocol that is not correct; then possibly run time errors occur. Just like for a correct protocol, whether run time errors occur again depends upon the semantics of the trigger rules. For instance, a deadlock context may never be reached at run time because the firing sequence leading to the deadlock context does not (or cannot) occur.

The protocol analyzer checks for 16 types of errors; these types of errors can result in one of the following four run time types of errors:

[17] Conflict

Conflicts have been explained above ([3] and [16]). When a conflict occurs at run time, the inference engine will fire one or more of the conflicting on-statements, but not all of them. It depends upon the textual ordering of the protocol which on-statements will be executed; the same conflict will always be solved in the same way.

For figure 8.16b, there are six possible orders, with two different results. If the middle transition fires first, the inference engine finds out that the left and the right transition are not enabled anymore, and will not fire these. When the inference engine first fires the left transition, then tries to fire the middle one (which fails), it can finally fire the right transition successfully. (In that case a multi-token state s3 results). Run time conflicts are not reported to the user. Protocols as shown in figures 8.3 and 8.16 can cause conflicts, which are reported during analysis, but for instance the protocol of figure 8.11b can cause a conflict at context s0 if t1 and t2 are true at the same time, which is not reported by the checker unless t1 and t2 are logically equivalent.

[18] Deadlock

At a deadlock context no further change of context is possible; the inference engine stops and reports the deadlock context. Run time deadlocks can be caused by nets as shown in figure 8.5a, 8.6b, 8.11a; all protocols that can come to a deadlock are reported during analysis.

[19] Multi-token state

A state is non-safe if it is made true by firing an on-statement while it was already true before firing. In a Petri net such a state gets two tokens. Since multi-token states have no significance for Simplexys, the inference engine will merge them. Nothing is reported to the user. The protocols shown in figure 8.6a, 8.12a and 8.16b can result in non-safe states at run time; all possible non-safe states are reported by the checker.

[20] System cannot terminate

There are protocols that do not come to a deadlock but cannot terminate anyway. This error is not recognized by the inference engine at all, and thus cannot be reported. A change of context is still possible, but this cannot result in the final context. Protocols that can cause this kind of error are given in figures 8.2a, 8.10a, 8.13 and 8.14.

8.8 List of warning and error messages

This section briefly recalls the protocol checker's error messages.

[2] ** ERROR No ON-statement has an empty TO list
The protocol must contain at least one on-statement with a * in its to-list

[3] ** ERROR Conflict at STATE [state that results in conflict]
[conflicting on-statements]
Do not use on-statements that have equal triggers and equal from-lists

[4] ** ERROR Unreachable STATE (not in any TO list) [state]
This state can never become active because it is not connected correctly. Put this state in the to-list of some on-statement, make it initially true, or delete it.

[5] ** ERROR Dead-end STATE (not in any FROM list) [state]
This state cannot become false. Put it in a from-list of some transition, replace it by a *, or delete it.

[6] ** WARNING Self loop
[on-statement that has common states in from-list and to-list]
It is tricky, but permitted to construct a protocol that contains a self loop.

[7] ** WARNING Identical ON statements
[on-statements that are identical]
These statements can be merged, but this is not necessary for correctness.

[8] ** WARNING Identical STATES [states that are identical]
These states can be merged, but this is not necessary for correctness.

[9] ** ERROR No connection with an initial STATE
[protocol part that is not connected with an initial state]
These states will never become true. Define one of the states as initially true, or connect this part of the protocol with another part that is connected with an initial state.

[10] ** ERROR No connection with a final STATE
[protocol part that is not connected with a final state]
These states will never become false. Replace one of the states by a *, or connect this part of the protocol with another part that is connected with a final state.

[11] ** ERROR Deadlock at CONTEXT [deadlock context]
[on-statements that result in that context]
** ERROR ON statement cannot fire
[on-statement that cannot fire]
From a deadlock context no further change of context is possible

[12] ** ERROR Multi-token STATE [state that gets two tokens]
[on-statements that give this state the first token]
** ERROR Extra token by ON statement
[on-statement that gives the second token]

[13] ** ERROR System cannot halt
Alter the protocol so that the context that contains only * states is reachable.

[14] ** ERROR System cannot halt in CONTEXT [context]
[on-statements that result in that context]
If this context is ever reached, Simplexys cannot halt.

[15] ** ERROR Transitions can never fire
[on-statements that cannot fire at least once]

[16] ** ERROR Conflict at CONTEXT [context resulting in conflict]
[on-statements that result in that context]
** ERROR Conflicting ON statements
[on-statements that are in conflict at that context]

Conclusions

The protocol that is part of a Simplexys knowledge base contains the time-sequential part of the knowledge. A protocol is considered correct if it represents a Live Safe P/T-net; if a protocol does not represent such a net, one or more warning messages are reported. Petri net theory has proved to be valuable in checking Simplexys protocols.

The Simplexys syntax and semantics do not allow a protocol to be cyclic; it must have a start and an end. For checking reasons, the protocol is extended so that it represents a (cyclic) connected Petri net; liveness is defined for the extended net. A correct protocol is live, which ensures that a transition to a new context is always possible and that the protocol can always terminate. Additionally, safeness is checked to prevent multiple tokens. If a protocol is correct, Simplexys executes it exactly like Petri net theory describes.

The checking algorithm is split up into three parts: syntactic, topologic and dynamic checking. The algorithms for syntactic and topologic checking are implemented straightforwardly from Petri net theory. These checks take little computation time, and warning messages are clear and uncomplicated.

The method of dynamic checking and the way in which errors are reported is tuned to knowledge engineers. Since a protocol represents a rather unrestricted type of Petri net, all possible reachable contexts must be generated. Although the construction of the reachability tree is rather time-consuming, it has been demonstrated that when an error is found, reporting the firing sequence resulting in that error is desirable. Moreover, additional conflict checking that is not supported by Petri net theory is possible if a reachability tree is constructed.

In order to make dynamic checking less time and memory consuming, it is sequentially performed for each individual subnet. Other methods that reduce the complexity of the protocol but do not affect the relevant dynamic characteristics, are possible as well. Since these are rather complicated methods which do not always gain results, these reduction methods have not yet been implemented, but may be relevant to further study.

The protocol checker is combined with the semantic analyzer that checks the remainder of the knowledge base; the protocol checker uses some of the algorithms of the semantic analyzer. Further integration should be studied. The semantic analyzer can now be improved because all possible contexts (generated by the protocol checker) can henceforth be known to it.

References

Aa, J.J.L.C.M. van der (1990).
Intelligent alarms in anesthesia: a real time expert system application. Ph. D. thesis. Eindhoven University of Technology, 1990.

Best, E. (1987) and P.S. Thiagarajan
Some classes of live and safe Petri nets. In: Concurrency and Nets: Advances in Petri nets. Proc. coll. on occasion of the 60th birthday of C.A. Petri, Schloss Birlinghoven, 12 Sept. 1986. Ed. by K. Voss, H.J. Genrich, G. Rozenberg. Berlin: Springer, 1987. P. 71-94

Best, E. (1986)
Structure theory of Petri nets: the free choice hiatus. In: Petri nets: central models and their properties. Advances in Petri nets 1986. Part 1. Proc. advanced course, Bad Honnef, 8-19 Sept. 1986. Ed. by W. Brauer, W. Reisig and G. Rozenberg. Berlin: Springer, 1987. Lecture notes in computer science, vol. 254. P. 168-205.

Blanchard, M. (1977)
Le GRAFCET pour une representation normalisee de cahier des charges d'un automatisme logique. Automatique et Informatique Industrielles, No. 61, pp. 27-32, 1977.

Blom, J.A. (1987)
SIMPLEXYS, a real-time expert systems tool. In: Expert systems: theory and applications. Proc. IASTED int. conf., Geneva, 16-18 June 1987. Ed. by M.H. Hamza. Anaheim, Cal.: Acta Press, 1987. P. 21-25.

Blom, J.A. (1990)
The SIMPLEXYS experiment: real time expert systems in patient monitoring. Ph. D. thesis. Eindhoven University of Technology, 1990.

Feldbrugge, F., Jensen, K. (1986)
Petri net tool overview 1986. In: Petri nets: applications and relationships to other models of concurrency. Advances in Petri nets 1986. Part 2. Proc. advanced course, Bad Honnef, 8-19 Sept. 1986. Ed. by W. Brauer, W. Reisig and G. Rozenberg. Berlin: Springer, 1987. Lecture notes in computer science, vol. 255. P. 20-61.

Genrich, H.J. (1986)
Predicate/Transition Nets. In: Petri nets: central models and their properties. Advances in Petri nets 1986. Part 1. Proc. advanced course. Bad Honnef, 8-19 Sept. 1986. Ed. by W. Brauer, W. Reisig and G. Rozenberg. Berlin: Springer, 1987. Lecture notes in computer science, vol. 254. P. 207-247.

Goltz, U. (1986)
Synchronic distance. In: Petri nets: applications and relationships to other models of concurrency. Advances in Petri nets 1986. Part 1. Proc. advanced course, Bad Honnef, 8-19 Sept. 1986. Ed. by W. Brauer, W. Reisig and G. Rozenberg. Berlin: Springer, 1987. Lecture notes in computer science, vol. 254. P. 338-358.

Jensen, K. (1986)
Computer Tools for Construction, Modification and Analysis of Petri Nets. In: Petri nets: applications and relationships to other models of concurrency. Advances in Petri nets 1986. Part 2. Proc. advanced course, Bad Honnef, 8-19 Sept. 1986. Ed. by W. Brauer, W. Reisig and G. Rozenberg. Berlin: Springer, 1987. Lecture notes in computer science, vol. 255. P. 4-19.

Lammers, J.O. (1990)
Knowledge based adaptive blood pressure control: a SIMPLEXYS expert system application. AIO thesis. Faculty of Electrical Engineering, Eindhoven University of Technology, 1990. EUT report 90-E-236.

Lautenbach, K. (1987)
Linear algebraic calculation of deadlocks and traps. In: Concurrency and Nets: Advances in Petri nets. Proc. coll. on occasion of the 60th birthday of C.A. Petri, Schloss Birlinghoven, 12 Sept. 1986. Ed. by K. Voss, H.J. Genrich, G. Rozenberg. Berlin: Springer, 1987. P. 315-336.

Lutgens, J.M.A. (1990)
Knowledge base correctness checking for SIMPLEXYS expert systems. M.Sc. thesis. Division of Medical Electrical Engineering, Faculty of Electrical Engineering, Eindhoven University of Technology, 1990. EUT Report 90-E-240

Petri, C.A, (1962)
Kommunikation mit Automaten. Diss. Bonn, 1962. Schriften des Institutes fuer Instrumentelle Mathematik der Universitaet Bonn, Nr. 2. English transl.: Communication with automata. Griffis Air Force Base, New York. Technical report. RADC-TR-65-377, vol. 1, Suppl. 1, 1966.

Petri, C.A. (1986)
Concurrency theory. In: Petri nets: central models and their properties. Advances in Petri nets 1986. Part 1. Proc. advanced course, Bad Honnef, 8-19 Sept. 1986. Ed. by W. Brauer, W. Reisig and G. Rozenberg. Berlin: Springer, 1987. Lecture notes in computer science, vol. 254. P. 4-24.

Reisig, W. (1985)
Petri nets: an introduction. Berlin: Springer, 1985. EATCS: monographs on theoretical computer science, vol. 4.

Reisig, W. (1986)
Place/Transition systems. In: Petri nets: central models and their properties. Advances in Petri nets 1986. Part 1. Proc. advanced course, Bad Honnef, 8-19 Sept. 1986. Ed. by W. Brauer, W. Reisig and G. Rozenberg. Berlin: Springer, 1987. Lecture notes in computer science, vol. 254. P. 117-141.

Rozenberg, G. (1986)
Behaviour of elementary net systems. In: Petri nets: central models and their properties. Advances in Petri nets 1986. Part 1. Proc. advanced course, Bad Honnef, 8-19 Sept. 1986. Ed. by W. Brauer, W. Reisig and G. Rozenberg. Berlin: Springer, 1987. Lecture notes in computer science, vol. 254. P. 60-94.

Thiagarajan, P.S. (1986)
Elementary net systems. In: Petri nets: central models and their properties. Advances in Petri nets 1986. Part 1. Proc. advanced course, Bad Honnef, 8-19 Sept. 1986. Ed. by W. Brauer, W. Reisig and G. Rozenberg. Berlin: Springer, 1987. Lecture notes in computer science, vol. 254. P. 26-59.

Thiagarajan, P.S. and K. Voss (1984).
A fresh look at free choice nets. Information and Control, Vol 61 (1984), p. 85-113.

Valette, R. (1986)
Nets in production systems. In: Petri nets: central models and their properties. Advances in Petri nets 1986. Part 1. Proc. advanced course, Bad Honnef, 8-19 Sept. 1986. Ed. by W. Brauer, W. Reisig and G. Rozenberg. Berlin: Springer, 1987. Lecture notes in computer science, vol. 255. P. 191-217.

Zwart, R.M.P. (1990)
Implementation and evaluation of a robust adaptive blood pressure controller (in Dutch). M. Sc. thesis, Division of Medical Electrical Engineering, Faculty of Electrical Engineering, Eindhoven University of Technology. 1990.

A Free Choice nets

A correct Simplexys protocol represents a Live Safe Place/Transition net; this is a rather unrestrictive net type. Before Live Safe P/T-nets were chosen to represent correct protocols, Free Choice nets have been studied. The difference between a Free Choice net and an ordinary P/T-net is an extra topological property that a Free Choice net must fulfill: states and transitions are connected in a restrictive way.

The literature [Best, 1987] states that such a restriction is appropriate: a protocol that represents a Free Choice net is preferable to a protocol that does not represent such a net. Furthermore, for Free Choice nets there are efficient methods through which the desired properties liveness and safeness can be checked [Best, 1987]. These algorithms are less time and memory consuming then those currently implemented.

For several reasons Free Choice nets are not used for Simplexys. First, in some cases the restriction of the way in which states and transitions should be combined is somewhat annoying. Also, the restrictions are not easy to explain to a knowledge engineer who is not a Petri net expert.

Second, error messages are less clear because the checking method finds neither the exact position of the error, nor the way in which it could be produced at run time. Third, the method of checking does not agree with the way in which the Simplexys inference engine executes the protocol: the checker would not take into account that transitions can fire simultaneously. As a consequence, checking for conflicts is not possible.

These drawbacks are the cause that Free Choice Nets have not been chosen to represent Simplexys protocols. Even the advantage that the analysis algorithms are more efficient, so that checking is faster, is not very important: checking of a real time expert system for medical applications is allowed to take as much time as necessary to prove its correctness as well as possible.

A basic property of a Petri net is that both concurrency and choice are possible (section 3.4). *Concurrency* means that two or more sub-processes can operate independently and simultaneously. *Choice* means that in some contexts there is a choice between two or more successor contexts. *Confusion* (definition 3.11) occurs when a choice interferes with concurrency.

Free Choice nets also support choice and concurrency, but the interaction between choice and concurrency is restricted, so that a choice is limited to one sub-process only. Free Choice Nets are confusion-free, because additional topological properties guarantee that choice and concurrency never interfere.

The assertion that is found in the literature [Thiagarajan, 1984] which states that a confusion-free net is preferable to an ordinary net, is difficult to understand. However, the deeper the impact of confusion is studied, the clearer the assertion that confusion should be avoided, becomes.

The following elementary subclasses of P/T-nets are both confusion-free, but neither supports both concurrency and choice.

Definition [A.1] *S-graph and T-graph*

a. A net is an S-graph
   $\Leftrightarrow \forall t \in T: |{\bullet}t|, |t{\bullet}| \leq 1$

b. A net is an T-graph
   $\Leftrightarrow \forall s \in S: |{\bullet}s|, |s{\bullet}| \leq 1$

In an S-graph, the from-list and the to-list of an on-statement both contain exactly one state. In a T-graph, each state is in the from-list of exactly one on-statement and in the to-list of exactly one other on-statement.

Free Choice nets are a restricted combination of S-graphs and T-graphs. There is no confusion because choice (from S-graphs) and concurrency (from T-graphs) are combined in a special way.

Definition [A.2] *Free Choice Nets and Extended Free Choice Nets*

a. A net is called a Free Choice net
   $\Leftrightarrow \forall\ s \in S$: if $|s{\bullet}| > 1 \rightarrow {\bullet}(s{\bullet}) = s$
   $\Leftrightarrow \forall\ t1, t2 \in T, t1 <> t2: {\bullet}t1 \cap {\bullet}t2 \neq \emptyset \rightarrow |{\bullet}t1| = |{\bullet}t2| = 1$
   $\Leftrightarrow \forall\ s1, s2 \in S: s1{\bullet} \cap s2{\bullet} \neq \emptyset \rightarrow \exists\ t \in T: s1{\bullet} = s2{\bullet} = t$
   $\Leftrightarrow \forall\ s \in S: |s{\bullet}| > 1 \Rightarrow \forall t \in s{\bullet}: {\bullet}t = s$

b. A net is called an Extended Free Choice net
   $\Leftrightarrow \forall\ s1, s2 \in S: s1{\bullet} \cap s2{\bullet} \neq \emptyset \rightarrow s1{\bullet} = s2{\bullet}$
   $\Leftrightarrow \forall\ t1, t2 \in T: {\bullet}t1 \cap {\bullet}t2 \neq \emptyset \rightarrow {\bullet}t1 = {\bullet}t2$

c. A Free Choice net is also an Extended Free Choice net.

Note that the "extended net" that is derived from a protocol, has no relation with the property "extended free choice". A protocol represents an Extended Free Choice net if two transitions that have common states in their from-lists have equal from-lists. Free Choice nets are shown for instance in figure 2.1 and figure 3.4; figure 3.5 shows a net that is not Free Choice.

ISSN 0167-9708
Coden: TEUEDE

(222) Jóźwiak, L.
THE FULL-DECOMPOSITION OF SEQUENTIAL MACHINES WITH THE SEPARATE REALIZATION OF THE NEXT-STATE AND OUTPUT FUNCTIONS.
EUT Report 89-E-222. 1989. ISBN 90-6144-222-2

(223) Jóźwiak, L.
THE BIT FULL-DECOMPOSITION OF SEQUENTIAL MACHINES.
EUT Report 89-E-223. 1989. ISBN 90-6144-223-0

(224) Book of abstracts of the first Benelux-Japan Workshop on Information and Communication Theory, Eindhoven, The Netherlands, 3-5 September 1989.
Ed. by Han Vinck.
EUT Report 89-E-224. 1989. ISBN 90-6144-224-9

(225) Hoeijmakers, M.J.
A POSSIBILITY TO INCORPORATE SATURATION IN THE SIMPLE, GLOBAL MODEL OF A SYNCHRONOUS MACHINE WITH RECTIFIER.
EUT Report 89-E-225. 1989. ISBN 90-6144-225-7

(226) Dahiya, R.P. and E.M. van Veldhuizen, W.R. Rutgers, L.H.Th. Rietjens
EXPERIMENTS ON INITIAL BEHAVIOUR OF CORONA GENERATED WITH ELECTRICAL PULSES SUPERIMPOSED ON DC BIAS.
EUT Report 89-E-226. 1989. ISBN 90-6144-226-5

(227) Bastings, R.H.A.
TOWARD THE DEVELOPMENT OF AN INTELLIGENT ALARM SYSTEM IN ANESTHESIA.
EUT Report 89-E-227. 1989. ISBN 90-6144-227-3

(228) Hekker, J.J.
COMPUTER ANIMATED GRAPHICS AS A TEACHING TOOL FOR THE ANESTHESIA MACHINE SIMULATOR.
EUT Report 89-E-228. 1989. ISBN 90-6144-228-1

(229) Oostrom, J.H.M. van
INTELLIGENT ALARMS IN ANESTHESIA: An implementation.
EUT Report 89-E-229. 1989. ISBN 90-6144-229-X

(230) Winter, M.R.M.
DESIGN OF A UNIVERSAL PROTOCOL SUBSYSTEM ARCHITECTURE: Specification of functions and services.
EUT Report 89-E-230. 1989. ISBN 90-6144-230-3

(231) Schemmann, M.F.C. and H.C. Heyker, J.J.M. Kwaspen, Th.G. van de Roer
MOUNTING AND DC TO 18 GHz CHARACTERISATION OF DOUBLE BARRIER RESONANT TUNNELING DEVICES.
EUT Report 89-E-231. 1989. ISBN 90-6144-231-1

(232) Sarma, A.D. and M.H.A.J. Herben
DATA ACQUISITION AND SIGNAL PROCESSING/ANALYSIS OF SCINTILLATION EVENTS FOR THE OLYMPUS PROPAGATION EXPERIMENT.
EUT Report 89-E-232. 1989. ISBN 90-6144-232-X

(233) Nederstigt, J.A.
DESIGN AND IMPLEMENTATION OF A SECOND PROTOTYPE OF THE INTELLIGENT ALARM SYSTEM IN ANESTHESIA.
EUT Report 90-E-233. 1990. ISBN 90-6144-233-8

(234) Philippens, E.H.J.
DESIGNING DEBUGGING TOOLS FOR SIMPLEXYS EXPERT SYSTEMS.
EUT Report 90-E-234. 1990. ISBN 90-6144-234-6

(235) Heffels, J.J.M.
A PATIENT SIMULATOR FOR ANESTHESIA TRAINING: A mechanical lung model and a physiological software model.
EUT Report 90-E-235. 1990. ISBN 90-6144-235-4

(236) Lammers, J.O.
KNOWLEDGE BASED ADAPTIVE BLOOD PRESSURE CONTROL: A Simplexys expert system application.
EUT Report 90-E-236. 1990. ISBN 90-6144-236-2

(237) Ren Qingchang
PREDICTION ERROR METHOD FOR IDENTIFICATION OF A HEAT EXCHANGER.
EUT Report 90-E-237. 1990. ISBN 90-6144-237-0

ISSN 0167-9708
Coden: TEUEDE

(238) Lammers, J.O.
THE USE OF PETRI NET THEORY FOR SIMPLEXYS EXPERT SYSTEMS PROTOCOL CHECKING.
EUT Report 90-E-238. 1990. ISBN 90-6144-238-9

(239) Wang, X.
PRELIMINARY INVESTIGATIONS ON TACTILE PERCEPTION OF GRAPHICAL PATTERNS.
EUT Report 90-E-239. 1990. ISBN 90-6144-239-7

(240) Lutgens, J.M.A.
KNOWLEDGE BASE CORRECTNESS CHECKING FOR SIMPLEXYS EXPERT SYSTEMS.
EUT Eeport 90-E-240. 1990. ISBN 90-6144-240-0

(241) Brinker, A.C. den
A MEMBRANE MODEL FOR SPATIOTEMPORAL COUPLING.
EUT Report 90-E-241. 1990. ISBN 90-6144-241-9