

Kurt Jensen

Coloured Petri Nets

Basic Concepts,
Analysis Methods
and **Practical Use**
Volume 3



Springer

Monographs in Theoretical Computer Science

An EATCS Series

Editors: W. Brauer G. Rozenberg A. Salomaa

Advisory Board: G. Ausiello M. Broy S. Even
J. Hartmanis N. Jones T. Leighton M. Nivat
C. Papadimitriou D. Scott

Springer

Berlin

Heidelberg

New York

Barcelona

Budapest

Hong Kong

London

Milan

Paris

Santa Clara

Singapore

Tokyo

Kurt Jensen

Coloured Petri Nets

Basic Concepts, Analysis Methods
and Practical Use
Volume 3

With 154 Figures



Springer

Author

Prof. Kurt Jensen
Aarhus University
Computer Science Department
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark

Series Editors

Prof. Dr. Wilfried Brauer
Institut für Informatik, Technische Universität München
Arcisstrasse 21, D-80333 München, Germany

Prof. Dr. Grzegorz Rozenberg
Department of Computer Science
University of Leiden, Niels Bohrweg 1, P.O. Box 9512
2300 RA Leiden, The Netherlands

Prof. Dr. Arto Salomaa
Data City
Turku Centre for Computer Science
FIN-20520 Turku, Finland

Cataloging-in-Publication Data applied for
Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Jensen, Kurt:
Coloured petri nets: basic concepts, analysis methods and practical
use / Kurt Jensen. – Berlin; Heidelberg; New York; Barcelona;
Budapest; Hong Kong; London; Milan; Paris; Santa Clara;
Singapore; Tokyo: Springer
(Monographs in theoretical computer science)
Vol. 3 (1997)

ISBN -13:978-3-642-64556-3 e-ISBN-13: 978-3-642-60794-3

DOI: 10.1007/978-3-642-60794-3

ISBN 978-3-642-64556-3 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1997
Softcover reprint of the hardcover 1st edition 1997

The use of registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and therefore free for general use.

Typesetting: Camera ready by author
Cover Design: MetaDesign, Berlin
SPIN: 10576689 45/3142-5 4 3 2 1 0 – Printed on acid-free paper

Preface

The contents of this volume are application oriented. The volume contains a detailed presentation of 19 applications of CP-nets, covering a broad range of application areas. Most of the projects have been carried out in an industrial setting. The volume presents the most important ideas and experiences from the projects, in a way which is useful also for readers who do not yet have personal experience with the construction and analysis of large CPN models. The volume demonstrates the feasibility of using CP-nets and the CPN tools for industrial projects.

The presentation of the projects is based upon material provided by the persons who have accomplished the individual projects. At the beginning of each chapter, we list their names and we say where the original material has been published. The original material often contains more elaborate information, e.g., about details of the modelled system and related work.

I have edited the material provided by the original authors. I have modified some of the CP-nets, e.g., to improve the layout and use more mnemonic names. In some cases, I have also changed a few net components, e.g., merged two transitions or introduced a Standard ML function for operations that are used in many arc expressions. These modifications make the CP-nets more appropriate as study material, but they do not change the essential behaviour of the CPN models.

The terminology in the original material has been modified to fit the terminology introduced in the first two volumes of this book. Redundancies with the material of the other volumes have been removed, e.g., the explanation of what a hierarchical CP-net is and how the CPN tools work. The typography has been modified to match that used for the other parts of the book. More detailed explanations have been added, e.g., of some of the CPN models and some of the analysis results. This has been possible since, Vols. 1 and 2 have given the readers a much more thorough knowledge of CP-nets than readers of ordinary research papers. Finally, it is discussed how some of the problems from the projects can be overcome or circumvented. Many of these problems have already been removed, e.g., by improvements of the CPN tools. Other problems can be avoided by a careful choice of modelling and analysis techniques.

The material has been modified in cooperation with the original authors and the final result has been approved by them. The conclusions and findings of the original papers have not been modified.

The CPN tools for occurrence graphs and performance analysis are rather new compared to the CPN editor and simulator. Nevertheless, they have been successfully used by several of the industrial projects reported in this volume.

For place and transition invariants there is not yet adequate tool support, and hence they are much more infrequently used in practical projects.

How to read Volume 3

In this volume we assume that the reader is familiar with the basics of CP-nets, in particular Sects. 1.1–1.4 and 3.1–3.2 of Vol. 1.

The individual chapters are to a very large extent independent, and hence they can be studied in any order. Readers who only want to study a few projects are invited to choose the application areas in which they are most interested. They may also choose chapters that cover the use of those analysis methods in which they have special interest. A short summary of each project is provided on the first page of each chapter. Readers who want to study all (or nearly all) the projects are recommended to read the chapters in the order in which they occur.

At Aarhus University, Vol. 3 is being used as part of the material for a graduate course. The course is organised as a set of colloquiums where each student is responsible for the presentation and discussion of one or more projects. In parallel to these presentations the students undertake a project in which they use CP-nets for modelling and analysis of a system. The students are expected to use one third of their study time on this course, for four months.

Acknowledgements

This volume would not have been possible without the efforts of those people who have accomplished the projects. All the hard work has been done during the projects and the production of the original material. Hence it is the original authors, and the other participants in the projects, who should get all the credit for the results. I sincerely thank all of them. I also thank the publishers for the permission to reuse the material. A number of students and colleagues have read and commented on earlier versions of this volume. Moreover, I am grateful to Andrew Ross for linguistic assistance during the preparation of the manuscript. The CPN project has been supported by the Danish National Science Research Council.

How to contact me

Despite all help some errors remain. That seems to be inevitable, no matter how many people read the manuscript. If you wish to report errors or discuss other matters you may contact me via electronic mail: kjensen@daimi.aau.dk. You may also take a look at my WWW pages: <http://www.daimi.aau.dk/~kjensen/>. They contain a lot of material about CP-nets and the CPN tools, including a list of errata for this book.

Table of Contents

1 Security System	1
1.1 Introduction to Security System	2
1.2 CPN Model of Security System	3
1.3 Simulation of Security System	10
1.4 Occurrence Graph Analysis of Security System	12
1.5 Implementation of Security System	16
1.6 Conclusions for Security System Project	18
2 UPC Algorithms in ATM Networks	21
2.1 Introduction to UPC Algorithms	22
2.2 CPN Model of UPC Algorithms	22
2.3 CPN Model of Traffic Sources	29
2.4 Simulation of UPC Algorithms	34
2.5 Conclusions for UPC Algorithms Project	36
3 Audio/Video System	39
3.1 Introduction to Audio/Video System	40
3.2 CPN Model of Audio/Video System	42
3.3 Simulation of Audio/Video System	45
3.4 Occurrence Graph Analysis of Audio/Video System	47
3.5 Conclusions for Audio/Video Project	50
4 Transaction Processing and Interconnect Fabric	51
4.1 Introduction to Transaction Processing	52
4.2 CPN Model of Transaction Processing	57
4.3 Introduction to Interconnect Fabric	64
4.4 CPN Model of Interconnect Fabric	66
4.5 Conclusions for Transactions and Interconnect Project	71
5 Mutual Exclusion Algorithm	73
5.1 Introduction to Mutual Exclusion Algorithm	74
5.2 CPN Model of Mutual Exclusion Algorithm	75
5.3 Occurrence Graph Analysis of Mutual Exclusion Algorithm	78
5.4 Conclusions for Mutual Exclusion Algorithm Project	84

6 ISDN Supplementary Services	85
6.1 Introduction to ISDN Supplementary Services	86
6.2 CPN Model of ISDN Supplementary Services	90
6.3 Validation of ISDN Supplementary Services	96
6.4 Conclusions for ISDN Supplementary Services Project	97
7 Intelligent Network	99
7.1 Introduction to Intelligent Network	100
7.2 CPN Model of Intelligent Network	103
7.3 Conclusions for Intelligent Network Project	114
8 Communications Gateway	117
8.1 Introduction to Communications Gateway	118
8.2 CPN Model of Communications Gateway	119
8.3 Conclusions for Communications Gateway Project	127
9 BRI Protocol in ISDN Networks	131
9.1 Introduction to BRI Protocol	132
9.2 CPN Model of BRI Protocol	136
9.3 Conclusions for BRI Protocol Project	147
10 VLSI Chip	149
10.1 Introduction to VLSI Chip	150
10.2 CPN Model of VLSI Chip	151
10.3 Conclusions for VLSI Chip Project	159
11 Arbiter Cascade	161
11.1 Introduction to Arbiter Cascade	162
11.2 CPN Model of Arbiter Cascade	163
11.3 Conclusions for Arbiter Cascade Project	168
12 Document Storage System	171
12.1 Introduction to Document Storage System	172
12.2 CPN Model of Document Storage System	173
12.3 Simulation of Document Storage System	176
12.4 Conclusions for Document Storage Project	178
13 Distributed Program Execution	179
13.1 Introduction to Distributed Program Execution	180
13.2 CPN Model of Distributed Program Execution	181
13.3 Verification of Distributed Program Execution	185
13.4 Conclusions for Distributed Program Execution Project	188

14 Electronic Funds Transfer System	189
14.1 Introduction to SADT	190
14.2 Introduction to Electronic Funds Transfer System	193
14.3 CPN Model of Electronic Funds Transfer System	196
14.4 Conclusions for Electronic Funds Transfer Project	201
15 Bank Courier Network	203
15.1 Introduction to Bank Courier Network	204
15.2 CPN Model of Bank Courier Network	205
15.3 Conclusions for Bank Courier Network Project	212
16 Network Management System	213
16.1 Introduction to Network Management System	214
16.2 CPN Model of Network Management System	217
16.3 Validation of Network Management System	221
16.4 Conclusions for Network Management Project	223
17 Naval Vessel	225
17.1 Introduction to Naval Vessel	226
17.2 CPN Model of Naval Vessel	227
17.3 Simulation of Naval Vessel	233
17.4 Conclusions for Naval Vessel Project	234
18 Chemical Production System	237
18.1 Introduction to Chemical Production System	238
18.2 CPN Model of Chemical Production System	240
18.3 Validation of Chemical Production System	243
18.4 Conclusions for Chemical Production Project	245
19 Nuclear Waste Management Programme	247
19.1 Introduction to Nuclear Waste Management Programme	248
19.2 CPN Model of Nuclear Waste Management Programme	253
19.3 Simulation of Nuclear Waste Management Programme	258
19.4 Conclusions for Nuclear Waste Management Project	260
References	261

Chapter 1

Security System

This chapter describes a project accomplished by *Jens L. Rasmussen, Mejar Singh, and Søren Christensen, Aarhus University, Denmark, in cooperation with Torben Andersen, Klaus L. Nielsen, and Søren V. Hansen, Dalcotech A/S, Nørresundby, Denmark, and John Mølgaard, Delta Software Engineering, Hørsholm, Denmark*. The chapter is based upon the material presented in [46]. The project was conducted in 1995.

We present an industrial use of CP-nets and the CPN tools to design a new security system, i.e., a system in which a building is surveilled and different kinds of irregularities reported to a control centre via the public phone network. During the project a graphical animation utility was developed. This made it possible to perform user-friendly simulations of the CP-nets. The animation package is general, and it is now one of the libraries which are offered to all users of the CPN tools. The project lasted for approximately one and a half years, consuming more than six man-years – approximately two for CPN activities.

Simulations and occurrence graphs were used to debug the CP-nets and to investigate their dynamic behaviour. In this way, a series of errors in the model were found and corrected. The final CPN model was used as the specification of the security system and hence as the basis for the implementation. The validation of the CPN model reduced the number of errors in the final implementation, hence increasing the software quality. By using CP-nets the company obtained a system which is more reliable, more flexible, and easier to maintain. The project was very successful. Dalcotech has reached the overall conclusion that CPN is well-suited for the types of systems they are developing and that it will be their main design method in the future.

Section 1.1 contains an introduction to the security system. Section 1.2 describes the project organisation and the CPN models of the security system. Section 1.3 discusses how the CPN models were simulated and presents the graphical animation utility. Section 1.4 describes how occurrence graphs were used to validate the design. Section 1.5 describes our experiments with automatic generation of the final code by extracting some of the Standard ML code generated and used by the CPN simulator. We also describe how the final implementation was done using conventional programming. Finally, Sect. 1.6 presents a number of findings and conclusions for the project.

1.1 Introduction to Security System

Dalcotech A/S is a small company. Altogether it has less than ten employees. The main product is security systems, which are sold in several European countries. The project described in this chapter used CP-nets to design and validate a new security system, Prisma C-96, which is an intruder alarm system. Figure 1.1 shows an example of a small installation. The **central unit** controls the security system and it is connected to a **control centre** via the public telephone network. The **PIDs** (Prisma Interface Devices) handle physical inputs, e.g., different kinds of detectors (shown as small circles) together with physical outputs, e.g., horn and flashlight. Usually there are several PIDs in an installation, and they are connected to the central unit via a network. The security system is operated from one or more **control panels** located inside the building. Each control panel has a keypad and an alphanumeric display. Access to the building (and the control panels inside) is granted by presenting a valid user code, a physical key, or a magnetic card (or combinations of these) at an **entry unit**, which usually is located on the outside wall of the supervised building, next to the entrance.

The security system divides the building into **areas** which may overlap each other. Each area may be in two different states. When an area is **set**, the activation of a detector usually leads to an alarm – reported locally by means of horn and flashlight and externally by sending a message to the control centre. When an area is **unset**, the activation of a detector usually does not cause an alarm. However, certain kinds of input (e.g., glass-break detectors or sabotage indications) may cause alarms even for unset areas. The states of the areas are con-

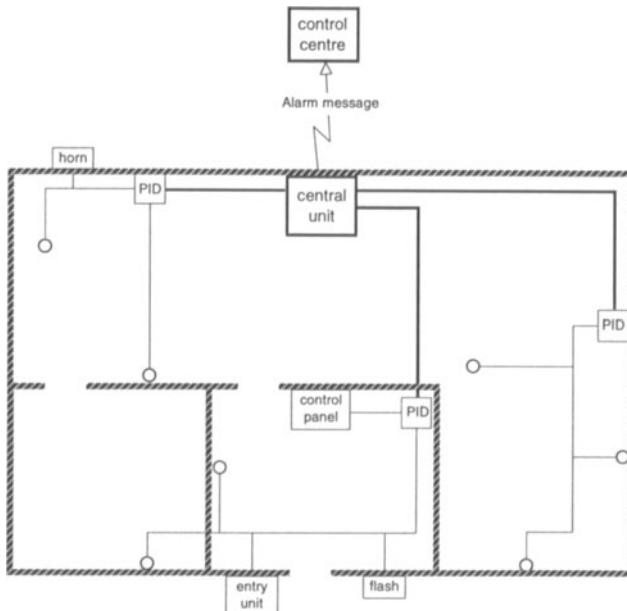


Fig. 1.1. Example of a small security system

trolled via the control panels and entry units. The most important function of the security system is to generate an alarm whenever a detector is activated in a set area – or under other illegal circumstances.

Installations can be quite large. The system handles up to 100 areas, 1000 inputs, 1000 outputs and 1000 different user codes. The reliability of the system is essential, which is one of the reasons for applying CP-nets in the design. The aim is to increase the quality of the software and make the new system more flexible than its predecessor.

1.2 CPN Model of Security System

In this section we describe the project organisation and the CPN models that were constructed.

The project group had eight members. All the hard work was done in a work group with three engineers from Dalcotech and two computer science students from Aarhus University. When the project started the engineers had no prior exposure to Petri nets while the students had a detailed knowledge of CP-nets and their tools. Additionally, three external consultants had regular meetings with the work group (one of the consultants is the author of this book). During these meetings the CPN models were discussed and proposals for improvements were made. Moreover, the overall project plan was discussed and further developed.

The consultants were responsible for the initial training of the engineers, which lasted for six full days and acquainted the engineers with the CPN language and the CPN tools. The training was very practical. Most concepts were introduced and investigated by means of small examples, where the participants used the CPN tools directly at the computer. For the first couple of days a number of small standard examples were used, but later on most examples were based on security systems. After the course, the engineers were able to do some amount of work on their own. They continued developing CP-nets in connection with a few small projects and in this way they became more acquainted with the language and the tools.

Meanwhile, the design of the new security system was started. At this stage, we discussed how CP-nets were to be applied in the project. What was the purpose of the CPN model? Which level of detail should the model have? Which parts of the security system should be modelled? The project group agreed that the main task was to design the software of the essential part of the security system, i.e., the central unit. Communication protocols, hardware failures, and the individual low-level behaviour of each control panel and entry unit were not to be modelled. Moreover, the CPN model should be sufficiently detailed. For instance, it should describe that some of the actions in the system are based on time-events. As an example, a user has to insert his key in the block key unit within 60 seconds after entering his user code at the code entry unit. At this stage, the work group did not know how to model this. It was just recognised that time-events play an important role in the system.

During the start-up period the engineers realised that CP-nets were well suited for making detailed system designs. This was reflected by the use of CP-nets in connection with two other projects at Dalcotech. With respect to the security system model, the start-up phase did not yield many tangible results, but provided an increased understanding of how CP-nets were to be used. The start-up phase lasted 2–3 months, partly because it takes a while to become familiar with the use of CP-nets, and partly because in this phase the work group also reflected upon the system's functionality. A requirements specification was initiated by the engineers. This was a textual description of the future system and was to serve as a guideline for developing the CPN model.

One of the consultants suggested that it might be possible to obtain the code to be used in the final implementation by automatically extracting some of the Standard ML code generated and used by the CPN simulator. In this way an error-prone and time-consuming manual implementation could be avoided. The feasibility of this idea was examined in considerable detail throughout the entire project, and we shall return to it in Sect. 1.5. However, at this stage it was uncertain whether it would be possible to perform automatic code generation, or whether the CP-net had to serve as a detailed design supporting a traditional implementation.

With a possible C++ implementation in mind, the work group started modelling the system using a client/server approach. It was believed that this would ease the final implementation and be flexible, allowing easy expansion of the system with few changes to old parts. In this CP-net, a single place models the communication between the different client and server processes – each token representing a message queue. Data belonging to a process can only be modified in the subnet of that process, avoiding situations in which two processes try simultaneously to update the same data. In this way, a high degree of data control and modularity was obtained. On the other hand, all processes need to communicate through the place with the message queues to obtain access to data in other processes. This gives a CP-net in which the net structure provides little overview of the communication patterns, because all messages pass through a single place.

During the start-up phase, one of the consultants constructed an alternative CPN model. The aim of this model was to capture the essence of the security system, as described in the requirements specification without considering the way of implementation. Hence we refer to this model as the implementation independent model. In this model the net structure provided a much better overview of the communication, since the messages were no longer channelled through a single place.

One approach had to be chosen and used for the rest of the project. The choice fell on the implementation independent approach. The loss of clarity and the focus on implementation issues were unacceptable drawbacks of the client/server approach. The project group decided that it was a question of modelling the security system concepts themselves instead of modelling an implementation of a security system. Hence it was agreed that a more comprehensible and general model would be desirable. Moreover, the engineers were becoming more acquainted with CP-nets and they began to get an idea of how an imple-

mentation independent model could be implemented. As a result of this decision, a new model was constructed. Small parts of the previous models could be re-used, but the new model was developed almost from scratch, with major revisions of all data structures and functions previously declared.

Having already made two (incomplete) models of the system, it was much easier to start on the new model. The work group was very careful with the design of the fundamental structure of the CP-net. Quite some time was spent on identifying candidates for the system's key processes. Most of the colour sets were based on information in the textual requirements specification – which was extended and made more detailed in parallel with the development of the CPN model. As an example, the requirements specification tells us that each user has a certain number of attributes, including user name, user code, and access rights. This was readily translated into a colour set declaration, where all the user data is represented by a list of records with a field for each user attribute mentioned in the requirements specification.

The work group used a top-down approach and started by determining the basic structure of the CPN model. In Fig. 1.2, the most abstract view of the CP-net is shown. The main system components are represented by the three substitution transitions. The *Environment* models the peripheral devices, i.e., the detectors, control panels, entry units, horns, flashlights, etc. The *Central unit* constitutes the largest and most important part of the model. It represents the software that manages the global data and realises the functionality of the security system. The *Configuration* consists of a single page. It is used to initialise the global data, by reading the necessary information from input files.

The socket places in the left part of Fig. 1.2 represent the communication between the *Central unit* and the peripheral devices in the *Environment*. They

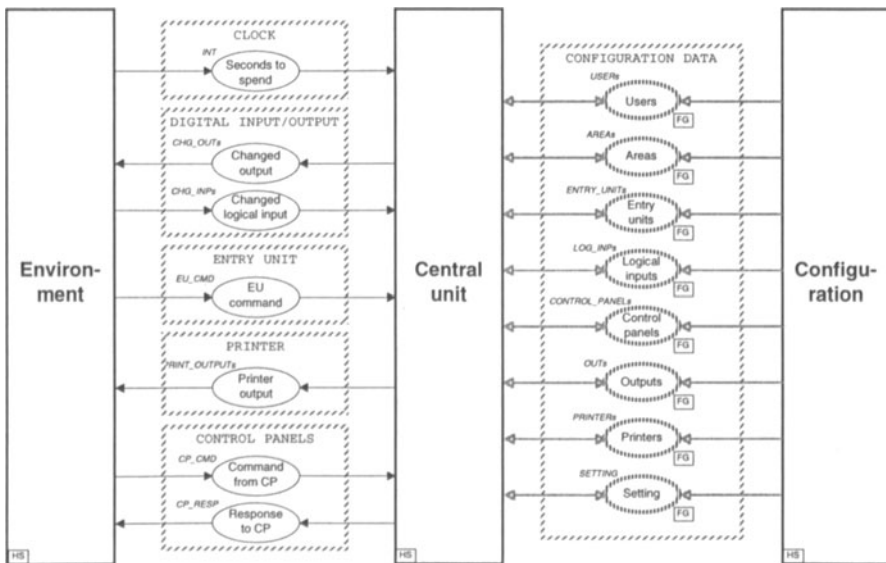


Fig. 1.2. Most abstract CPN view of security system

constitute the I/O interface to the environment. As an example, we can see that digital input/output are modelled by two places: *Changed logical input* holds a list of those detectors for which the PIDs have recorded a change. Analogously, *Changed output* holds a list of outputs from the *Central unit* to the PIDs. This may, e.g., cause the horn or the flashlight to be activated.

The fusion places in the right part of Fig. 1.2 represent global data initialised by the *Configuration*. Their colour sets are lists of records, where each record has 10–20 entries describing both static and dynamic data. Instead of using fusion places, we could have represented the global data by means of socket places. However, this would have introduced a lot of extra arcs at the subpages of the *Central unit* – because each page must contain the places for those global data which are used by any of its subpages, even when the page itself does not use the corresponding data. These extra arcs would have blurred the net structure without providing any extra information. Hence, we represent the global data by means of fusion places instead of using the port/socket mechanism. The arcs surrounding the fusion places in Fig. 1.2 are auxiliary arcs. This means that they have no formal meaning. They are added to show that the data is initiated by the *Configuration* and read/modified by the *Central unit*.

Splitting the model into subnets for the *Environment*, *Central unit*, and *Configuration* turned out to be a good design decision, as it isolates the software of the central unit from the peripheral hardware units and the configuration. Later on, it also made it easy to use three different versions of the *Environment* and *Configuration*. One version was used during simulations, another during occurrence graph analysis, and a third during automatic code generation.

Figure 1.3 shows the subnet of the *Central unit*. The work group identified five key processes: *Input event handler* receives changed inputs from PIDs and determines whether alarms are to be generated. *Time handler* controls time-dependent data and handles time-out events. *Control panel handler* takes care of the user interaction via the control panels by sending menus to the control panels and executing the actions requested by the user. *Log handler* updates the logs and produces printer output, e.g., when an alarm occurs. *Entry unit handler* takes care of the entry units. Each of the five processes are activated when there are tokens on the input places. The tokens represent environment changes, e.g., an activated detector or a command entered by the user.

The *Central unit* starts by executing the two upper transitions. They *Initialize outputs* and *Initialize control panels*. Then the five key processes can be executed. However, the single token at place *CPU idle* ensures that only one of the key processes can occur at a time. One of the processes is chosen and finished before another can proceed. This kind of co-processing is done to prevent the processes from operating on shared data, simultaneously. It also reflects the fact that only one processor exists in the hardware to be used for the implementation of the security system.

Figure 1.4 shows the subnet of the *Input event handler*. The sequential process flow runs from top to bottom and is indicated by thick arcs. The fusion places are a subset of those shown in Fig. 1.2. They represent the global data used by the *Input event handler*. The data flow is shown by using black and white arrow

heads (where the former indicate the main direction of the data flow). As an example, it can be seen that transition *Find alarm type* (in the central part of Fig. 1.4) updates the global data for *Areas* (and removes the old data). Some of the arc expressions use the *%*-operator, which is a shorthand for an if-then-else construction. When the left-hand expression evaluates to true, the value is the value of the right-hand expression. Otherwise the value is the empty multi-set. The guards and arc expressions contain a considerable number of quite complex Standard ML functions. We will not explain these functions in any detail – their names and arguments provide some hints about what they are doing.

Now let us investigate Fig. 1.4 in more detail. On the upper, leftmost place *Changed logical inputs* are received from the *Environment*. This may, e.g., represent an opened input circuit due to an activated infrared detector. An input can be either closed, opened, or sabotaged. The first transition *Receive changed inputs* updates the state of the changed inputs (i.e., the marking of the *Logical inputs* fusion place). Then *Get input data* retrieves the state of the *Area* in which the input is situated. *Find alarm type* determines the alarm type to be generated. For example, a glass-break detector will generate an intrusion alarm. Additionally, *Find alarm type* delays input changes from areas which are temporarily unset via entry units. If an unset command is not received from a control panel within 45 seconds, the delayed inputs will activate alarms (via the *Time handler*). The next transition *Filters events already registered*. This will, e.g., avoid repeated reporting of a broken window. In order to reduce the number of false alarms inputs may be combined in sets, where all members must be opened before an alarm is triggered. This is handled by *Combined input check*. Then, an alarm

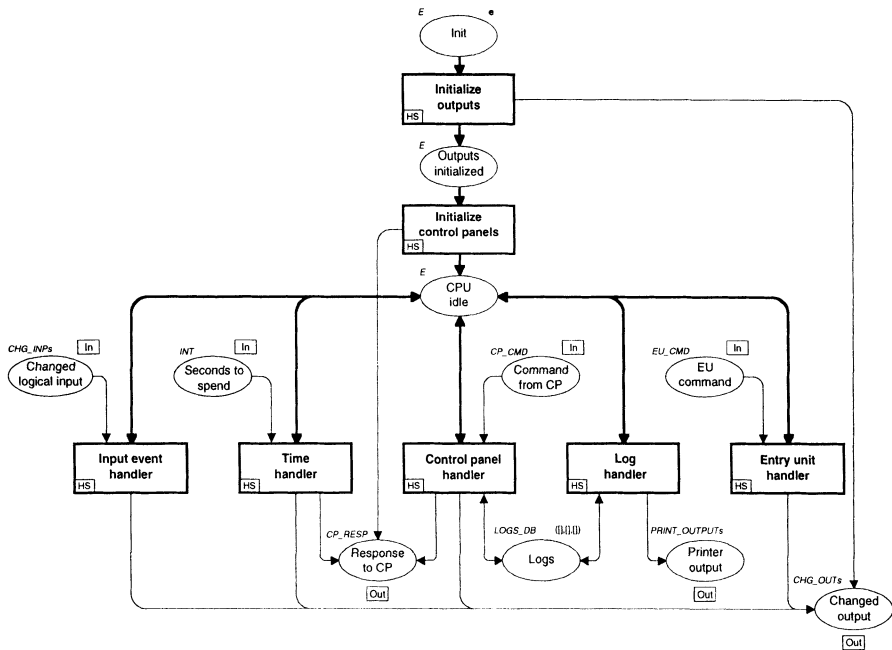


Fig. 1.3. CPN page for Central unit

condition may be registered by the subnet of *Register alarm* (which is a substitution transition).

A detailed inspection of the arc inscriptions shows that the *Input event handler* executes the six transitions in the central column, starting from the top and continuing downwards. Under certain conditions a transition may decide that there is nothing more to be done. Then it puts a token on *Generate outputs* (instead of producing a token to trigger the next transition). This is, e.g., the case when *Find alarm type* determines an alarm type which is a “no alarm”. When this happens the subnet of *Update outputs* (which is a substitution transition) reevaluates the output expressions to determine whether there are any *Changed outputs* due to the newly detected input changes. Each output has an expression that determines whether the output shall be on or off. As an example, the output expression for the horn could be *AlarmCond(1) or AlarmCond(2)* indicating that the horn is activated iff there is an alarm condition either in area one or two.

Figure 1.5 shows how the *Time handler* manages the time-dependent data. The six fusion places (in the upper left part) contain tokens with a colour in which an integer represents the number of seconds before some time period ex-

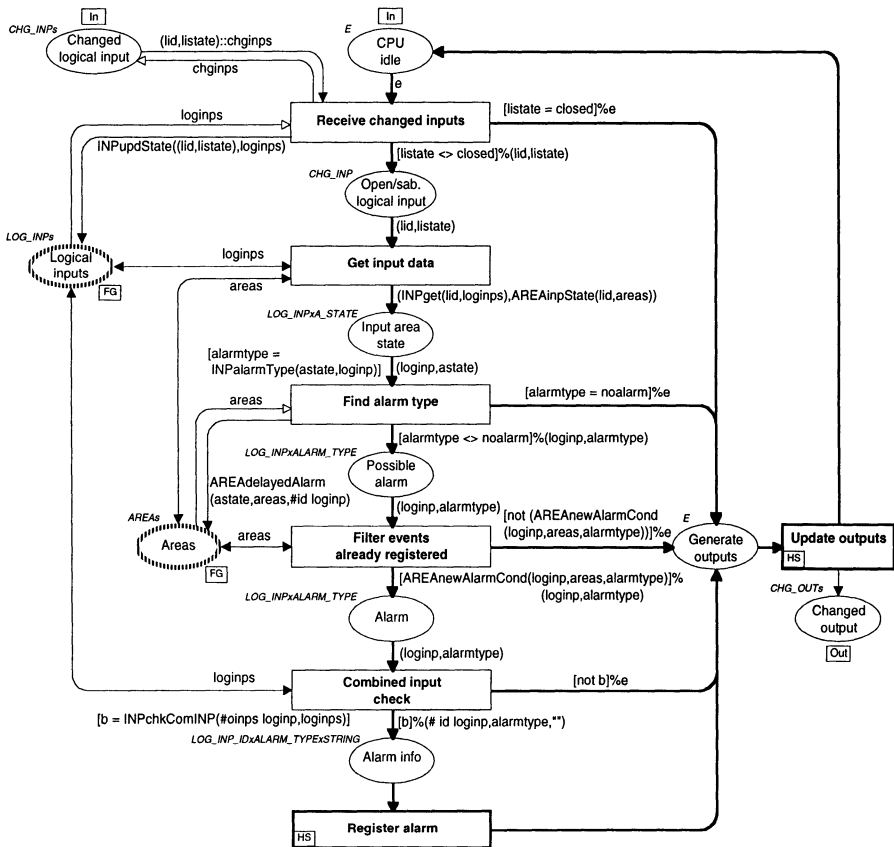


Fig. 1.4. CPN page for *Input event handler*

pires. As an example, each *Area* has an integer field denoting the remaining seconds the area stays temporarily unset. The *Pass one second* transition decreases all these time values by one. The transition can only occur when *Seconds to spend* has a token. The token represents clock-ticks generated by a clock in the *Environment*. When one of the time values reaches zero, appropriate action is taken by one of the five transitions in the lower left part.

Note that the CPN model of the security system is *untimed* – although it describes time-dependent issues, such as time-outs. Timed CP-nets were not considered, since we did not intend to make performance evaluations. Instead, the CP-net should be able to explicitly cope with “real” clock-ticks received from the *Environment*.

Above, we have described the most abstract level of two of the key processes in the *Central unit*. The complexity of the remaining three processes is similar. Altogether, the model consists of 38 pages with 95 transitions and 325 places. Some parts of the handlers for the control panels and entry units are identical and hence they use the same subpages. The CPN model contains approximately 4 000 lines of CPN ML code – primarily for declarations of functions and colour sets. Most of the subpages and all the colour set declarations can be found in an appendix of [45].

The modelling and the simulation (to be described in the next section) were done by the entire work group. The engineers did approximately half of the work while the students did the rest. Most of the work was performed at two different locations, separated by more than 100 kilometres. For this purpose hierarchical CP-nets turned out to be very useful. The engineers could work on one set of pages while the students were working on another. Then the modifications were merged by loading pages from one model into the other. After a few

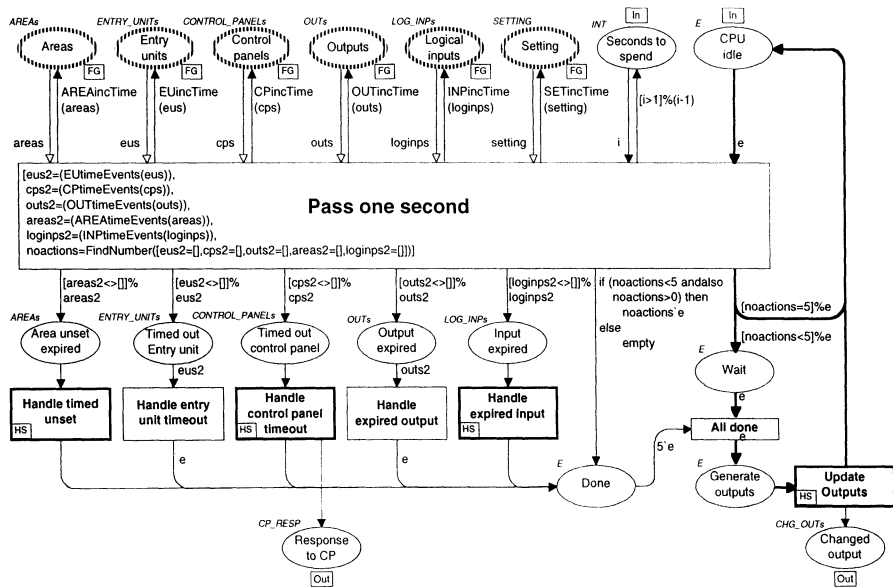


Fig. 1.5. CPN page for *Time handler*

months the engineers were able to work on their own. This meant that they constructed and simulated their own CPN pages.

As the design of the system evolved throughout the modelling process, the work group frequently had to reject or radically change parts of the model. The most serious changes (apart from rejecting the client/server approach) were made after employees from Dalcotech learned more about the requirements that a security system needs to fulfil in order to be approved by the German standard for security systems. Furthermore, monthly meetings with the consultants induced a number of changes to the model. In this way, the modelling process was conducted in a prototyping fashion.

In parallel with constructing the CP-net, a written requirements specification was worked out. For those of us without knowledge of security systems, this was an important aid in the modelling process, as we could see a detailed description of what we were supposed to model. On the other hand, the design of the detailed CP-net made it possible to elaborate the requirements specification. The correlation between the requirements specification and the CPN model was useful and also resulted in two system descriptions instead of one. This was useful later on, as both were used to retrieve information about the system.

1.3 Simulation of Security System

Simulation is an important instrument for debugging and validating CP-nets. It gives the developers an improved understanding of the system behaviour.

In order to enhance the user interface during simulations, we constructed a graphical animation utility, as a library written in Standard ML. This library

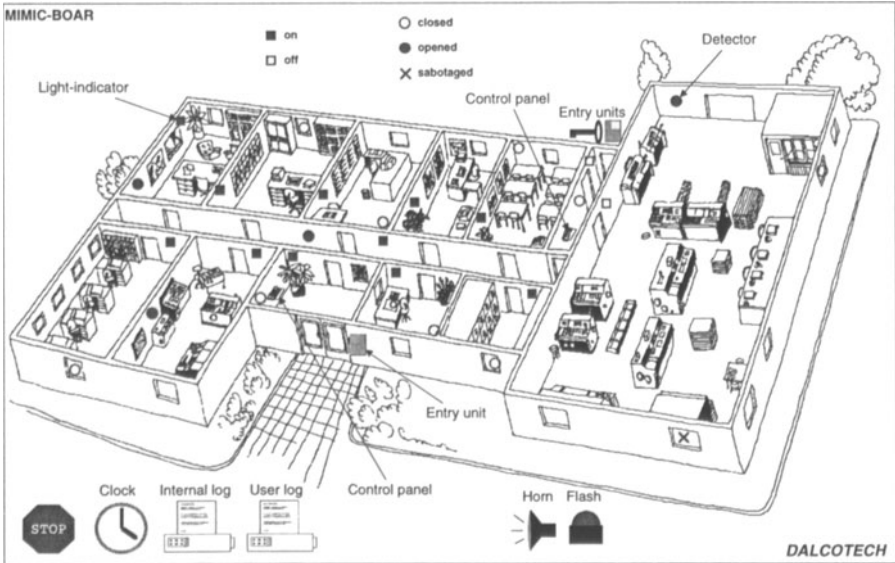


Fig. 1.6. Animation page for user-friendly simulations

supports two-way communication between a user and the CPN simulator. As the simulation of a CPN model progresses the simulator animates the results, by showing/hiding graphical objects and by moving them around – in a similar way as actors can appear and move on a theatre scene. During the animation the user can interact with the CPN model by using the mouse to click at some of the graphical objects. In this way a user-friendly simulation is obtained.

During the simulations of the security system, the user observes a window similar to the one shown in Fig. 1.6. It shows the building being surveilled by the security system. The user can inspect the state of each detector, which is represented by a small circle, and can change this state by clicking on the detector and then on one of the three possible states (in the upper part of the window). The user can also inspect the state of each area by looking at the small square boxes positioned in each room. They represent light indicators (*on* means that it is safe to enter the room, while *off* means that the room is under surveillance).

At the bottom of the window the user can see that the horn and flashlight for the moment are active. When they are deactivated the two icons are hidden and two other, slightly different, icons are shown (on the same positions). The stop icon (in the lower left corner) allows the user to stop the simulation. The clock icon allows him to advance the time some specified number of seconds (without observing all the intermediate changes). Finally, the two log icons allow him to inspect the contents of the system logs (which are shown in separate windows).

The user can also click one of the small icons representing control panels (positioned in the two rooms with an outer door). This opens the window shown in Fig. 1.7. The display (in the middle upper part) contains the appropriate text; the light indicators (to the very left) are as on a real control panel. Moreover, the keys are active and can be pressed by the mouse, e.g., to enter a user code, which then is caught by the *Environment* subnet and passed to the subnet for the *Central unit*. The entire control panel dialogue, where the user browses in menus (received from the central unit) and selects commands (to be sent to the central unit) has been implemented using our animation utility. The entry units are handled in a similar way, but their window is of course different, since it reflects the look and dialogue of the entry units. The animation utility and the simulator are synchronised. When an input has been requested, the simulator waits until an object has been clicked by the user.

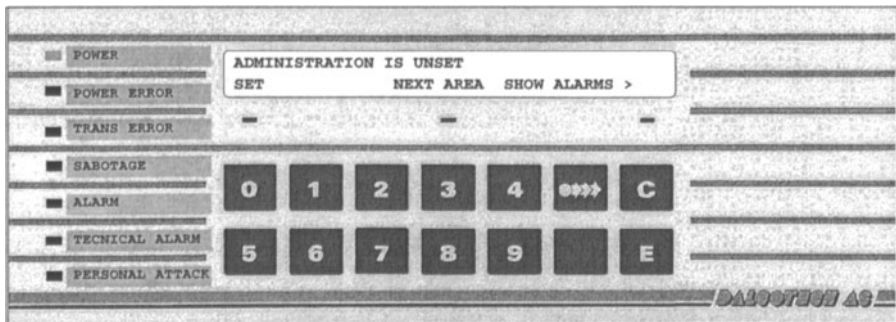


Fig. 1.7. Animation page with control panel

The library for the animation utility was constructed as part of the security system project and as part of the students' Master's thesis work. It is now a standard library offered to all users of the CPN tools. The library consists of a number of Standard ML functions, which are called from code segments of the individual transitions. The graphic objects are built by the user within the CPN editor (or they are bit-maps entered via a scanner). This allows modellers to customise the graphic layout for their own specific needs. For more information see the user's manual [47].

Due to the animation utility, the modellers did not have to inspect the tokens of the CP-net. Instead they could follow the progress of the simulation in a much more direct and natural way. This made it easy to check whether the security system had the intended behaviour. Only when errors occurred did the modellers have to dig down in the CPN model to locate and fix the error. The animation utility was also useful when the design was presented to people without any knowledge of CP-nets, or to the external consultants (who all had a detailed knowledge of CP-nets).

The work group used an iterative approach, alternating between modelling and simulation. The first prototype was gradually refined, and eventually it constituted the final model. Whenever the model was changed, simulation was used to investigate and validate the changes. By using the animation utility it was usually easy to check whether the present design met the modellers' expectations. Typical simulation runs contained 300–400 steps.

The quite extensive simulations revealed a considerable number of errors, which were located and corrected long before the implementation started. The simulations also gave the members of the work group a deep understanding of the design – identifying good and weak spots. Note that this insight was obtained during the design phase and not afterwards. This fact is believed to have had a strong positive impact on the quality of the final design.

1.4 Occurrence Graph Analysis of Security System

As mentioned above, the CPN model is used as the specification of the final program. Therefore, finding an error in the model, by means of simulation or analysis, may remove an error from the final program. Simulations of the CP-net revealed many flaws and provided feedback for improving the model during the entire design phase. At the end of the design phase, the two students applied occurrence graph analysis in order to locate as many errors as possible before the implementation. Due to lack of resources and a very tight time schedule, the O-graph analysis was made by the students alone. However, we do believe that the analysis could have been performed by the engineers – after some modest amount of training.

To obtain occurrence graphs with a manageable size, it was necessary to simplify the CPN model. Some of the colour sets were records with 10–20 fields which either change dynamically during simulation or are determined by the configuration of the system. Even when large colour sets (such as integers) are

replaced by smaller ones, the number of different reachable markings is enormous. Hence, we did not expect to be able to prove the correctness of the CP-net for all possible configurations and all possible inputs. Instead, we wanted to use small configurations and limited sets of inputs – to look for errors in the model and to get an increased understanding of the system. In this way, the occurrence graph analysis became more like an “extended simulation” investigating all possible occurrence sequences of small configurations for limited sets of inputs. This supplements our “normal simulations” which investigate one occurrence sequence of a large configuration allowing all inputs.

In order to be able to generate full O-graphs, we applied a minimal configuration with only one area, one input, one control panel, etc. Furthermore, some of the data fields were modified. For instance, time-out periods were all set to expire after one second. It should be obvious that these modifications constitute a quite strong simplification of the security system, limiting the generality of our analysis results. For example, we were not able to examine conflicts between two control panels attempting to operate on the same area. As described later in this section, we also made occurrence graphs for larger configurations, but for these we were only able to obtain partial O-graphs.

We also made a new version of the *Environment* subnet. For the simulations we used an environment that contained, among other things, the interface to the animation utility. This utility is totally unnecessary and useless for the O-graph analysis. Hence, the simulation *Environment* was replaced by a much simpler O-graph *Environment* consisting of the single page shown in Fig. 1.8.

Each of the five transitions generates a different kind of input. The upper transition generates clock-ticks, and from the arc expression of the rightmost arc, we can see that the only possible value is one. The *Time sync* fusion place synchronises the transition with the *Central unit*. It makes it impossible to generate a new clock-tick until the previous one has been processed. The second transition generates *Changed logical inputs*. The two places to the left of the transition determine the range of the possible inputs. With the initial marking shown it is only possible to change input number *one* to either *closed* or *opened*. The third transition generates different kinds of commands from entry units. The *EU sync* fusion place synchronises the transition with the *Central unit*, while two of the other input places determine the range of the possible commands. With the initial marking shown it is only possible to generate a *code enter* or an *euunset* command from entry unit number *one*. The last two transitions generate commands from control panels. The upper transition generates user codes while the lower one generates menu selections. The guard guarantees that only interesting selections are made, i.e., those that are necessary in order to test the important functions of the control panel.

In addition to changing the *Configuration* and *Environment*, we had to make several other modifications to the CPN model. For example, we excluded the log facility, and we limited the number of *Changed outputs* so that there was never more than one. We also made the *CPU idle* place accessible at the *Environment* page – forcing the environment transitions to occur only when the CPU was idle.

In spite of all these modifications, we could not achieve a full O-graph for the minimal configuration and the environment displayed in Fig. 1.8. Instead, we made O-graphs for selected parts of the system. For example, we disabled the entry unit commands by removing some tokens from the initial marking of Fig. 1.8. In this way, we generated a considerable number of different O-graphs. The largest had 150 000 nodes and 250 000 arcs. For practical reasons, we usually worked with smaller O-graphs which had up to 50 000 nodes.

Above we have described how it was necessary to simplify the CPN model in order to be able to obtain full O-graphs. It is obvious that the simplifications make it impossible to claim that we have *proved* the correctness of the security system. Nevertheless, the O-graph analysis turned out to be successful. It greatly improved the work groups' confidence in the system – and our knowledge of its dynamic behaviour. As we shall see below, the occurrence graph analysis also located a significant number of errors in the CPN model.

The O-graphs were constructed by means of the OG tool, which now is an integral part of the CPN tools. When an O-graph has been calculated, a set of standard queries makes it possible to investigate reachability, boundedness, home, liveness, and fairness properties – corresponding to the proof rules of Sect. 1.4 of Vol. 2. For example, a single function call returns a list of all tran-

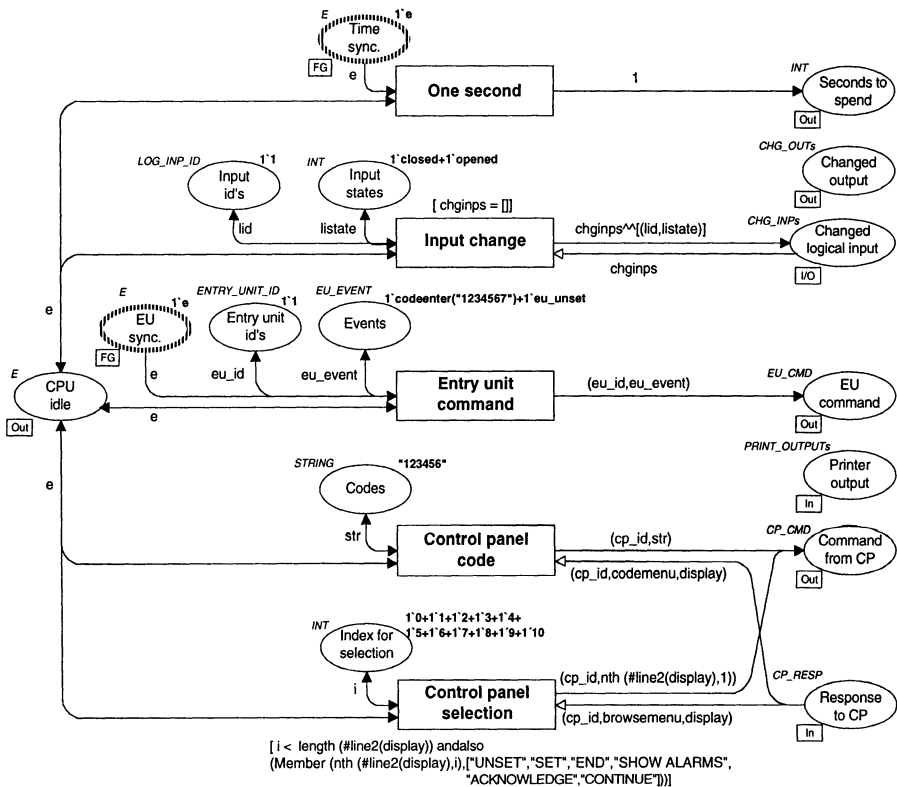


Fig. 1.8. CPN page for *Environment* used during occurrence graph analysis

sition instances which are live, while another function returns the maximal number of tokens which a specified place instance may have. In general, the standard queries increased our confidence in the model. For instance, we established bounds on places and determined which transitions were live.

With environments like the one in Fig. 1.8, we found that only one terminal strongly connected component existed. Thus, the individual nodes of the terminal component are all home markings. This agrees with our expectations. When alarms are generated, they can be acknowledged and thereby removed; when an area is unset, it can be set again; and so forth.

Some properties could not be verified by means of standard queries. For this purpose, the OG tool provides search functions for traversing nodes, arcs, or strongly connected components of the O-graph, using user-defined functions to retrieve the desired information. As an example, we made a query which searched all nodes of the O-graph and checked that each of them had only a single token representing one of the five key processes.

The most important property of the security system is to report an alarm to the control centre whenever a detector is triggered under illegal conditions. We applied the search functions of the OG tool to make queries that found all nodes where a token had arrived at the place *Open/sabotaged logical input* in the upper part of Fig. 1.4. Such a token represents a situation for which an alarm should be generated. For all these nodes, we found the first successor nodes in which the input change had been treated by the *Input event handler*, and for these nodes we verified that the system was in an alarm condition and that the horn had been turned on.

As described above, we investigated standard as well as system-specific dynamic properties of the security system model. Moreover, the analysis revealed 14 non-trivial errors, which had not been found by the quite extensive simulations performed. Approximately half of these errors are of kinds which make them highly unlikely to be found by simulations or testing. They only occur under very special circumstances, e.g., involving a time-out for one particular action while another particular action is being executed.

Many of the errors were found by means of exceptions raised by Standard ML functions, e.g., when by mistake the function received an empty list as argument. When an exception is encountered during the construction of an O-graph, the generation halts. It is possible to move the marking of each individual OG node to the simulator. In this way, the marking that causes problems can be examined. Furthermore, the OG tool makes it easy to find occurrence sequences leading to the marking and in this way the error can be located.

For example, we discovered that the O-graph generation halted when the control panel timed out (which normally happens after 30 seconds of idle time). The exception appeared when a user had been acknowledging alarms, a time-out had occurred, and the user had chosen to acknowledge alarms again. This turned out to be caused by an index error in the control panel data structure. It is highly unlikely that the sequence of actions leading to this error would have been encountered in ordinary simulations/program tests. Hence, the error is likely to

have existed in the final program – had it not been found by means of the occurrence graph analysis.

The OG tool makes it possible to display selected parts of the marking of nodes and the binding element of arcs. By examining these markings and binding elements, we also found a number of errors. For instance, we discovered that the list of area records situated on the *Areas* place (in Fig. 1.2) sometimes were expanded (so that some records were duplicated). The problem had not been encountered during the simulations.

As mentioned above, we also constructed a number of partial O-graphs for larger configurations (e.g., with two control panels) and a wider range of inputs generated by the OG *Environment*. For these O-graphs we limited the number of times the *Central unit* was allowed to become active (often allowing only a single activation). Thereby, the O-graphs became acyclic, and most of the standard dynamic properties of the O-graphs became uninteresting. For instance, no transitions were live. We also found errors in the model by this approach. As an example, we discovered that inconsistencies could occur when simultaneously two users were acknowledging alarms at different control panels. The problem was that a user could attempt to acknowledge an alarm which had already been acknowledged by another user. Also this kind of error is difficult to find by means of simulations/tests.

All in all, O-graph analysis turned out to be an efficient and fruitful way of debugging the security system. We have demonstrated that O-graphs can be used even when the system is complex and only part of the CPN model can be examined. One lesson to be learned is that it is important to be as economical as possible by removing unnecessary parts of the occurrence graph. To illustrate this, let us once more consider the O-graph environment in Fig. 1.8. Late in the occurrence graph analysis, we recognised that this environment is too general, since it allows several inputs to be generated and queued for processing – while it would have been sufficient to generate one input at a time. A quick experiment showed that this simple modification, for a typical O-graph, reduced the number of nodes from more than 10 000 to less than 2 000, i.e., by a factor 5.

A much more detailed description of the occurrence graph analysis and the located errors can be found in part 3 of [45].

1.5 Implementation of Security System

As mentioned in Sect. 1.2, one of the consultants suggested that it might be possible to obtain the code for the final implementation by extracting some of the Standard ML code used by the CPN simulator. In this way an error-prone and time-consuming manual implementation phase would be avoided. The feasibility of this idea was examined in considerable detail throughout the entire project and it actually turned out that such an approach was possible.

The basic idea was to reuse some of the ML functions which the simulator generates for its own use. One of these functions tests whether a transition has any enabled binding elements, while another function calculates the effect of an

occurring binding element, i.e., how the marking is updated. The ML functions were automatically extracted from the simulator code, positioned in a simple run-time environment (without any kind of graphics, etc.) and transferred from the Macintosh used for the CPN modelling to an IBM PC. Here the code was compiled under Moscow ML and CAML light, which provide an ML implementation that runs significantly more slowly but requires much less run-time memory than those ML systems usually applied by the CPN tools. Finally, the object code was burned into two PROMs and mounted in a prototype of the hardware to be used for the central unit. The stand-alone executable of the SML code used approximately 450 KB of memory, and the data also took up 450 KB on the 80386 PC-card used in the central units of the security system.

The experiments were encouraging. They showed that the idea was feasible and the PROMs seemed to work as expected. However, we also recognised that the automatic implementation was too slow, at least for large configurations. Hence, it would be necessary to improve the speed. This is not at all impossible, since improvements can be obtained in several different ways. First of all, we used the simulator code without any modifications. Instead we could have generated code which was more optimal, e.g., by removing those parts that deal with breakpoints and other unnecessary issues. Secondly, the speed depends on the order in which the individual transitions are tested and executed. We did some work to optimise this, but more could have been gained. Finally, we could have used a faster processor, and it is indeed possible to purchase a processor which is ten times as fast as the one Dalcotech is using, without paying a significantly higher price. However, here we were stopped by a number of bureaucratic problems related to the certification of the security system.

At the end it was decided that it would be too risky to make the first implementation of the security system via automatic code generation. A tight schedule, lack of resources, and uncertainty about the success of this new approach were the main reasons for the decision. By using CP-nets, the project had already involved a large amount of new technology. Thus there were no resources for additional experiments. For marketing reasons, there was a strong pressure to finish the implementation as soon as possible. Hence, it was decided to make the first implementation in a conventional way. However, there are still ongoing experiments with automatic code generation, and it is believed that later versions of the system may be realised in this way.

To implement the central unit, the ML functions and colour sets of the CPN model were translated into C++ functions and types. This was a reasonably straightforward and trivial task. To make the translation as easy and consistent as possible, the engineers developed a company standard describing how this should be done. The standard prescribes that the structure of the CP-net be used as a template for the C++ program. This means that there is a piece of code for each CPN page, and that this code is divided into units that implement the individual transitions. Page and transition names are shown in comments or (when appropriate) in the name of the function that implements a transition. Hence it is easy to relate the individual parts of the C++ program to the parts of the CPN model. In the CPN model there is a separate file for each global fusion place containing

the ML functions used to access the data represented by the place. Each of these files is translated into a file with a C++ class containing a similar set of functions. The new file gets a name which is identical to the old one (with some specified extension).

Although the implementation is done in C++, it turns out that the engineers still think and talk about the security system in terms of CP-nets. As an example, they have made a small debugger that shows the marking of some of the most important CPN places – now realised via C++ data structures.

During the implementation it has become necessary to make small modifications to the design. In this case the engineers update the CP-net before they implement the changes. Hence, the CPN model is kept up-to-date with the actual implementation. The model serves as the specification of the security system and it will be used when new employees are introduced to the system.

The implementation of the security system took approximately eight months. However, this also includes the time used to design and implement the “low-level” parts of the systems, e.g., the communication between the PIDs and the central unit. To specify this part, the engineers also used CP-nets, and for this purpose they constructed a new, quite complex CPN model.

1.6 Conclusions for Security System Project

Figure 1.9 provides an overview of the project activities up to the start of the final implementation. The inscription in the lower left corner of each box presents the main agents of the activity (engineers, students, and/or consultants). Analogously, the inscription in the lower right corner – and the size of the box – indi-

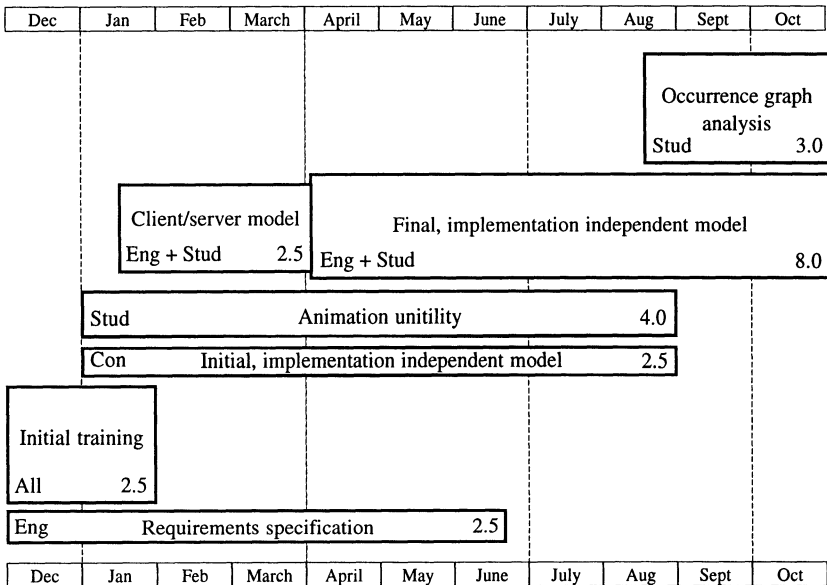


Fig. 1.9. Overview of the activities in the security system project

cates the number of man-months used for the activity. Together the activities shown involve a total of 25 man-months.

The project was supported by a grant from ESSI, which is an EU-funded programme to support the introduction of new software development methods in small companies. The project was very successful. The rest of this section quotes from [1], which constitutes Dalcotech's final report to ESSI. It is written by one of the engineers.

The CPN method was successfully used for software design in the base-line project. It proved to work excellent for gradually developing a design from a written requirements specification to a software design ready to implement:

- Powerful in describing control and data flow.
- Supports both abstract designs and detailed/specific designs.
- The designer determines the mix of graphics and text.
- Hierarchical model.
- Deficiencies easy to spot – early.
- Design errors spotted through simulations (test).
- Easy for several persons to discuss a CPN design of a system (CPN is a good common language!).
- We spent 6–7 days of intensive training followed by 2–3 weeks of work before we were ready to apply the method productively.
- Not well suited for user interface design.

It is an advantage to start using the CPN model already in the requirement specification phase. The method provides a very good help in structuring thoughts and ideas at this time. The CPN method facilitates reuse, as it is very easy to take parts of one design and use them in the design of a different system. We will look further into:

- The occurrence graph method.
- Automatic code generation from CPN.

We have reached the overall conclusion that CPN is well suited for the types of systems we develop and that it will be our main design method in our future development.

Chapter 2

UPC Algorithms in ATM Networks

This chapter describes a project accomplished by *Henning Clausen and Peter Ryberg Jensen, Aarhus University, Denmark*. The chapter is based upon the material presented in [17]. The project was conducted in 1993.

We have studied different Usage Parameter Control (UPC) algorithms for high speed Asynchronous Transfer Mode (ATM) networks. The purpose of the UPC algorithms is to prevent congestion. This is done by monitoring incoming traffic and marking the excess cells in such a way that they may be dropped, should congestion arise. During the standardisation of ATM networks, one of the issues to be decided is the choice of an appropriate UPC algorithm. To contribute to this discussion, we used timed CP-nets and the CPN tools to investigate four different UPC algorithms, proposed in the protocol literature. The choice of algorithm may depend upon the traffic type. However, as the ATM network is a multi-purpose network one algorithm has to be found, which is optimal under as many different situations as possible. Hence, we investigated the appropriateness of all four algorithms for six different traffic types. The traffic was generated by a separate CPN model, which simulates Markov Chain Processes. This model is independent of the ATM network and the UPC algorithms, and hence it can be used to generate test data for other kinds of networks.

Previous, theoretical work has proposed the Leaky Bucket algorithm and the Exponentially Weighted Moving Average algorithm to be the best UPC algorithms. Surprisingly enough, our investigations do not confirm this proposal. Instead our results indicate that the Triggered Jumping Window algorithm is the best for a majority of traffic types.

Section 2.1 contains an introduction to ATM networks and the purpose of UPC algorithms. Section 2.2 presents the CPN model of the four UPC algorithms. Section 2.3 presents the CPN model of the traffic sources. Section 2.4 discusses how simulation was used to investigate and compare the UPC algorithms. Finally, Sect. 2.5 presents a number of findings and conclusions for the project.

2.1 Introduction to UPC Algorithms

By the mid-1980s it became clear that the ISDN standard was unable to meet the emerging demand for broadband services and that it did not fully take advantage of the new high-speed transmission, switching, and signal processing technologies. Hence, the international standardisation organisations started work on **Broadband ISDN** (B-ISDN), which was supposed to incorporate both low-speed and high-speed applications and be able to handle both bursty and continuous traffic. As examples, it should be able to handle telephone connections requiring 64 Kbps (kilobits per second) as well as multi-media applications demanding several hundred Mbps (megabits per second). To meet these requirements a flexible and fast underlying transfer mode is needed, and for this purpose **Asynchronous Transfer Mode** (ATM) was chosen.

With optical transmission media the performance bottleneck is no longer the transmission speed in the physical media. Instead it is the processing time in the network switches and the time it takes a signal to propagate from the sender to the receiver. Also the probability of transmission errors is very low when optical fibers are used. The need for complex (hence slow) transmission protocols that are capable of detecting and correcting transmission errors is reduced. ATM is defined as a simple transmission protocol with a minimum of processing done in each network node. Congestion control is moved from the individual nodes of the ATM network to the points where the users connect to the network. At these connection points the user traffic must be monitored to guarantee that the user does not exceed the reserved bandwidth. This is done to avoid problems for other users in the ATM network, e.g., lost data due to buffer overflow and longer transmission times. The monitoring is done by a **Usage Parameter Control** (UPC) algorithm, which detects excess cells and tags them, indicating that they may be dropped, should congestion arise. A UPC algorithm is **appropriate**, i.e., well suited for its purpose, if it:

- detects and tags the correct amount of excess traffic, without interfering with non-excess traffic,
- reacts quickly when an excess situation starts/stops,
- is so fast that it does not introduce significant cell delays.

2.2 CPN Model of UPC Algorithms

Our CPN models are timed. This means that they have a **global clock** which represents the model time. Moreover, we allow each token to carry a **time stamp** which describes the *earliest* model time at which the token can be used, i.e., removed by a binding element.

In a timed CP-net a binding element is said to be **colour enabled** when it satisfies the requirements of the usual enabling rule (Defs. 2.8 and 3.6 of Vol. 1). However, to be **enabled**, the binding element must also be **ready**. This means that all the time stamps of the tokens to be removed must be less than or equal to the current model time. To model that an activity/operation takes Δt

time units, we let the corresponding transition t create time stamps for its output tokens that are Δt time units larger than the clock value at which t occurs. This implies that the tokens produced by t are unavailable for Δt time units.

The execution of a timed CP-net is time driven, and it works in a similar way to that of the event queues found in many programming languages for discrete event simulation. The system remains at a given model time as long as there are colour enabled binding elements that are ready for execution. When no more binding elements can be executed, at the current model time, the system advances the clock to the next model time at which binding elements can be executed. Each marking exists in a closed interval of model time (which may be a point, i.e., a single moment). The occurrence of a binding element is instantaneous. A much more detailed introduction to timed CP-nets and a formal definition can be found in Chap. 5 of Vol. 2. The use of timed nets allow us to investigate the performance of a model, i.e., the speed by which it operates.

The CPN model of the UPC algorithms has the page hierarchy shown in Fig. 2.1. The *ATM* page provides the most abstract view of the system. The *Network* page describes the ATM network. The *User* page describes the users of the network. The two subpages *Generate Traffic* to the ATM network and *Receive Messages* from the UPC algorithm. The *UPC* page and its subpages model the *Buffer Overflow Control* and the four *UPC Algorithms*. The *Initialisation* page creates the initial marking of the CPN model by means of information read from a text file.

ATM traffic is transferred in fixed-sized cells of the form shown below. The four fields with thick border lines stands for Virtual Path Identification, Virtual Channel Identification, Cell Loss Priority, and Information.



The other three fields are not interesting for our work, and hence they are not included in the CPN model. They stand for Generic Flow Control, Payload Type, and Header Error Control.

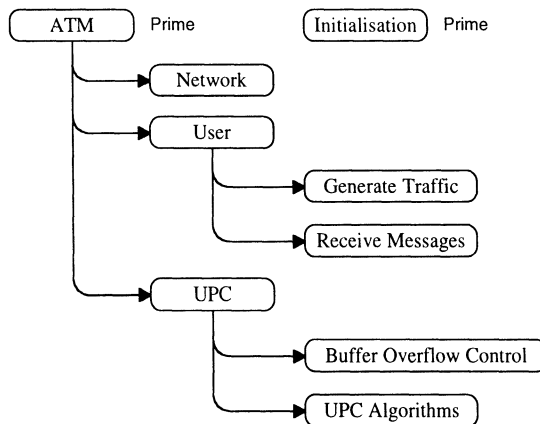


Fig. 2.1. Page hierarchy for UPC algorithms

The above structure of ATM cells is modelled by the following colour set declarations:

```

color VPI = Int;
color VCI = Int;
color CLP = Bool;
color Connection = record vpi: VPI * vci: VCI;
color Data = Int;
color Message = with UPCCellLoss;
color Signal = record con: Connection * mes: Message;
color Information = union data: Data + signal: Signal;
color Cell = record con: Connection * clp: CLP * inf: Information timed;

```

We use integers to represent the VPI and VCI fields, while we use a boolean for the CLP field. The latter is used to tag those cells that may be dropped, should congestion arise. A *Connection* is a record containing a VPI and a VCI. We are not interested in the actual *Data* to be transferred via the ATM cells. Hence, we abstract away the data and replace it by an integer, which is used to recognise the cell during our subsequent analysis of delays, tagging, etc. In our model there is only one kind of *Message* to be generated by the network. It informs the user that UPC cells have been lost at the specified *Connection*. Finally, we see that the *Information* in a *Cell* either is *Data* (to be transferred over the network) or a *Signal* (to the user).

The *UPC* page is shown in Fig. 2.2. It receives cells from the *User* page and passes these to the *Network* – after performing *Buffer Overflow Control* and *UPC Algorithm* enforcement.

Now let us describe the four UPC algorithms. Three of the algorithms build on window mechanisms. A window is a time period, during which the algorithm

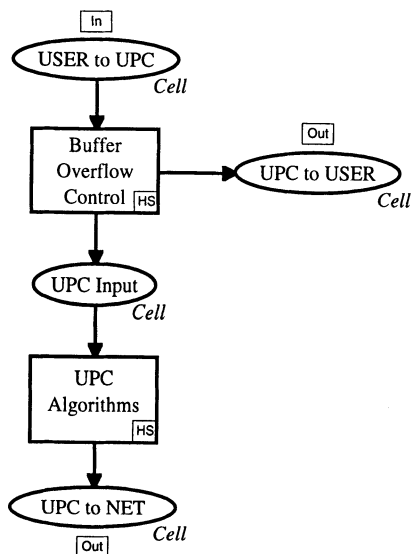
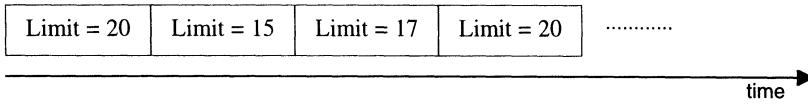


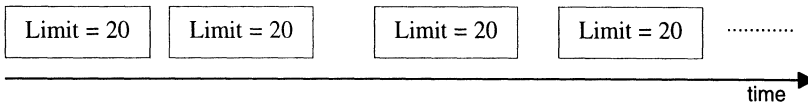
Fig. 2.2. CPN page for *UPC*

counts the number of cell arrivals. By comparing the count to a given limit, the algorithm determines whether an excess situation has happened.

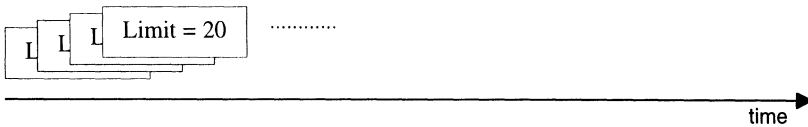
The **EWMA** algorithm uses a **jumping window**. This means that a new window is started as soon as the old has finished. At the end of each window a new limit is computed as an **exponentially weighted moving average** of all previous arrival counts:



The **TJW** algorithm uses a **triggered jumping window**. This means that when a window has finished, a new one is not started until the moment when the next cell arrives. The same limit is used for all windows:



The **SW** algorithm uses a **sliding window** which moves continuously along the time axis. Cells are added to the window when they arrive, but removed again after a time period determined by the width of the window. The same limit is used for all windows:



The **LB** algorithm behaves like a **leaky bucket** where the content runs out of the holes in the bottom in a steady stream. When the bucket is full new cells are tagged as excess. Otherwise they just increase the level in the bucket. More details about the four UPC algorithms can be found in [19] and [48].

The *UPC Algorithms* page is shown in Fig. 2.3. It has five places. The two upper places are input and output ports (assigned to sockets in Fig. 2.2). Each token on these places represents a *Cell* and carries a time stamp indicating the cell arrival time at the UPC/network.

When a cell arrives at the UPC it has to be checked for excess, i.e., violation of the agreed bandwidth. This is done by transition *UPC Check*. The *Algorithm* place specifies the UPC algorithm to be used, while two other places specify different *Parameters* used by the algorithms and the times for *Updates* of the window/bucket.

During a simulation the *Algorithm* place contains a single, fixed token. This means that we use one UPC algorithm at a time. By modelling all algorithms on the same CPN page, it becomes easier to see the differences and similarities of the four algorithms. Moreover, we can switch between them, simply by changing the token on *Algorithm*. The colour set of this place is determined by the following declarations:

```

color EWMA = product Real * Real;
color TJW = Real;
color SW = Real;
color LB = Int;
color Algorithm = union ewma:EWMA + tjw:TJW + sw:SW + lb:LB;

```

EWMA, TJW and SW have a real which determines the width of the window. The second real in EWMA is part of the formula to calculate the new limit. The integer in LB determines the size of the bucket (the size of the holes are determined from the bandwidth).

To perform the *UPC Check* it is necessary to know the status of the connection to which the arrived cell belongs. This information is stored in place *Parameters*, which has a token for each active connection:

```

color Parameters = record bandwidth: Int * count: Int * limit: Int *
                    period: Real * history: Real;
color ConnxParams = product Connection * Parameters timed;

```

The *Parameters* specify the *bandwidth*, the current cell *count*, the *limit*, the time *period* between updates, and the *history* of previous arrivals. The latter is used by EWMA in its calculation of new limits.

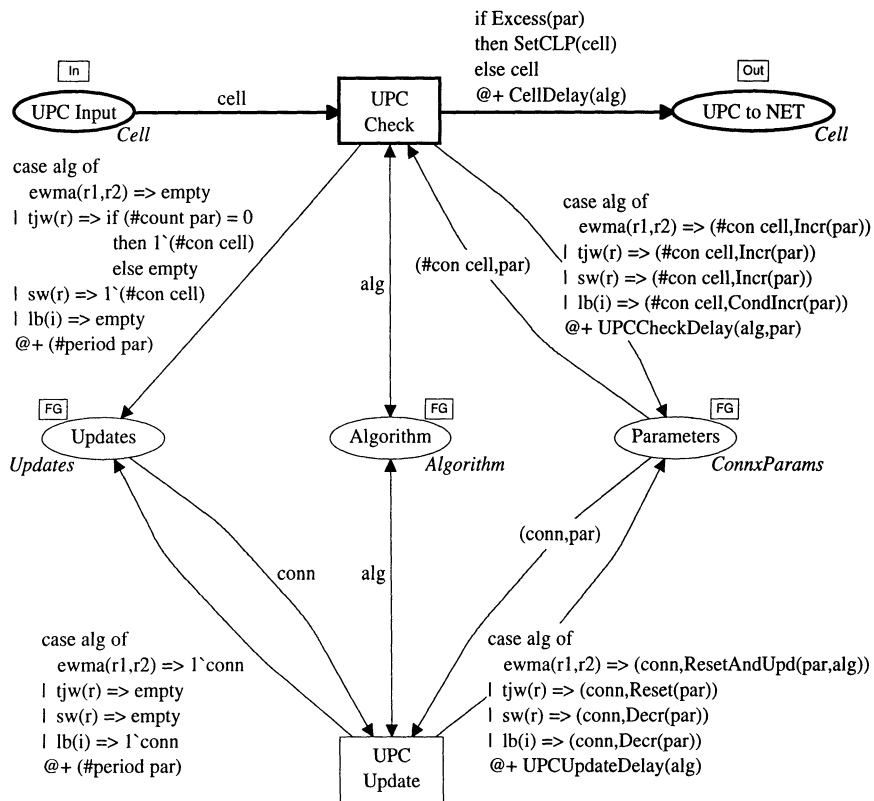


Fig. 2.3. CPN page for *UPC Algorithms*

Transition *UPC Check* examines whether the incoming traffic complies with the reserved bandwidth. The *Excess* function (in the upper right-hand arc expression) compares the count and the limit. If the count is larger than or equal to the limit, the *CLP* bit is set. Otherwise, the cell passes without modifications. In both cases the cell experiences a small *CellDelay*, while the necessary calculations are done. The length of the delay depends on the algorithm, and has been determined by considering the sequence of actions necessary to perform the corresponding calculations. Our delays reflect the performance of ordinary hardware technology. For instance, an addition has been set to take 10 nanoseconds. The effect of these delay assumptions turns out to be minor and therefore we will not go into details with this subject. An elaborate discussion of the delay values can be found in [18].

Transition *UPC Check* returns the *Algorithm* token without changing it. However, for the *Parameters* token the count is incremented by 1. This is done by the functions *Incr* and *CondIncr* (the latter only adds 1, when count < limit). The *UPC CheckDelay* describes the time used to process the cell. It is equal to the *CellDelay* plus the time used to update the *Parameters*.

Two of the window algorithms need to record the arrival time of some cells. This is done at place *Updates*, which has the colour set:

color Updates = Connection timed;

The time stamps tell us when updates of the window/bucket is to be performed. For the TJW algorithm we put a token on *Updates* each time the count is increased from 0 to 1. This indicates that a new window has been created. The time stamp of the token expires after a *period* which is equal to the width of the window. For the SW algorithm we put a token on *Updates* for all cell arrivals. The time stamp expires when the cell is too old to be part of the current window.

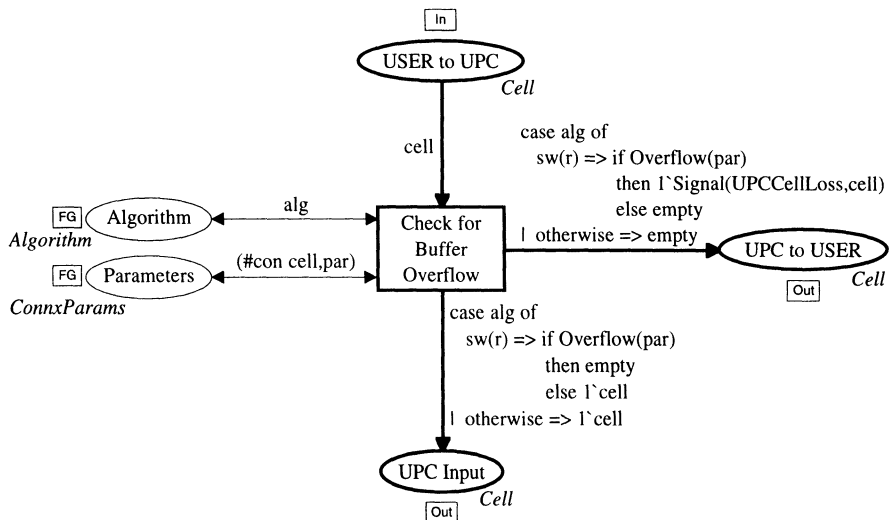


Fig. 2.4. CPN page for Buffer Overflow Control

Occurrence of transition *UPCUpdate* (in the lower part of Fig. 2.3) is triggered by the tokens on *Updates*. The transition reads information from *Algorithm* and *Parameters* and modifies the latter. For a TJW algorithm the count is *Reset* to zero. For SW and LB the count is decremented by one. For EWMA the situation is more complicated, since we also have to calculate the new limit and update the history. All this is done by function *ResetAndUpdate* which is declared as follows:

```

fun ResetAndUpd(par: Parameters, ewma(period, gamma)) =
  {bandwidth = #bandwidth par,
   limit = NewLimit(par, period, gamma),
   count = 0,
   period = #period par,
   history = NewHistory(par, period, gamma)};

```

For EWMA and LB, updates are done with fixed intervals, totally independent of cell arrivals. For these algorithms *UPCCheck* never adds tokens to place *Updates*. Instead there is always exactly one token (per connection), and this token is put back (with a new time stamp) when *UPCUpdate* occurs.

The *Buffer Overflow Control* page is shown in Fig. 2.4. The places for *Algorithm* and *Parameters* are the same as in Fig. 2.3. This is achieved by means of two global fusion sets. From the two case-statements, we see that overflow is relevant only for the SW algorithm. Overflow situations are detected by means of an ML function *Overflow* which simply compares a cell count to a buffer limit. The user is notified by a *UPCCellLoss* signal represented by a token at place *UPCtoUSER*.

Page *Initialisation* is shown in Fig. 2.5. It contains a single transition which always occurs in the first step of each simulation and then never again. The code segment reads a text file by means of a set of standard input functions. Based on the information in the file, a number of tokens are created at the output places, which all belong to global fusion sets. In this way, tokens are distributed all over the CPN model. The binding of the five variables is determined by the code segment. They are all multi-set variables. This means that they are bound to

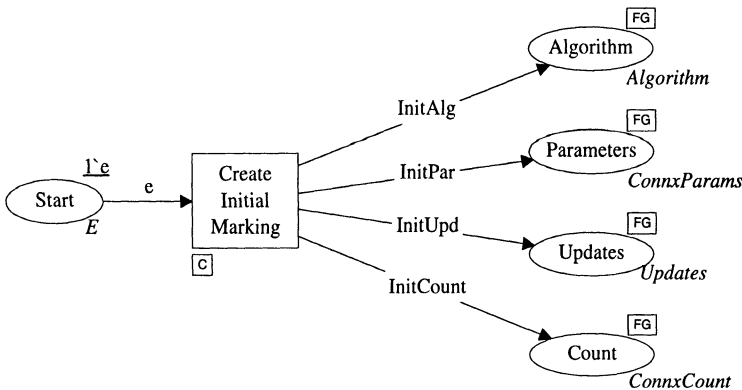


Fig. 2.5. CPN page for *Initialisation* of the CPN model

multi-sets of token colours. By means of the mechanism described above, it is easy to switch between different initial markings. The user simply modifies the text file, while nothing has to be changed in the CPN model.

Page *Generate Traffic* works in a similar way as *Initialise*. It reads the time stamps to be used for cell arrivals from a text file generated by the CPN model described in Sect. 2.3. By using two separate CPN models we save simulation time, since it is only necessary to generate each sequence of cell arrivals once, instead of generating it during the analysis of each of the four UPC algorithms. This also means that we use totally identical sequences of cells to analyse all four UPC algorithms.

Some of our simulations deal with more than 200 000 cells. To ease the subsequent analysis of delays, tagging, etc., we store the results of each simulation in a text file. This is done by a code segment on the *Network* page. For each cell we record the cell number, the arrival time, and the value of the CLP field. The data on the output file is analysed by means of a number of simple ML functions which generate input to a spreadsheet and charting program.

2.3 CPN Model of Traffic Sources

We use a discrete-state continuous-time Markov Chain Process (MCP) to generate our traffic. The bit rate is divided into a finite number of discrete levels:

$$0, 1, 2, \dots, m-1, m.$$

The rate at level i is $i * a$, where a is a constant. With negative exponentially distributed intervals, the level is changed by one – upwards or downwards. At level i , the next change is determined by two stochastic variables *up* and *down*:

$$\text{up} \sim e^{-((m-i) * u)}$$

$$\text{down} \sim e^{-(i * d)}.$$

u and d are two constants, while e is the base of the natural logarithm. Every time the MCP enters a new level, an outcome of *up* and *down* is obtained. If $\text{down} < \text{up}$, the next change is downwards, with waiting time *down*. Otherwise, it is upwards, with waiting time *up*. The mean of a negative exponential function e^{-p} is p^{-1} . Hence, high levels have a large chance of being decremented, while low levels have a large chance of being incremented. The MCP is illustrated in Fig. 2.6. Each node represents a level with the inscribed bit rate. The arcs represents the changes from one level to another.

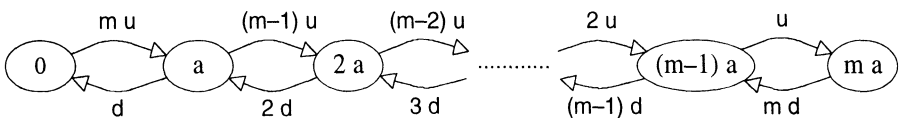


Fig. 2.6. State/transition diagram for Markov Chain Process

The MCP has an equilibrium which depends on the parameters m , a , u , and d . The equilibrium level has a binomial distribution with parameters m and $u/(u+d)$. By choosing an outcome of this distribution as the start level, we immediately reach the equilibrium with the following mean and variance:

$$\text{Mean}(i * a) = m * a * \frac{u}{u+d}$$

$$\text{Var}(i * a) = m * a^2 * \frac{u * d}{(u+d)^2}$$

From this we see how to control the MCP. If $u/(u+d)$ is close to 0, the mean bandwidth is also close to 0. If $u/(u+d)$ is close to 1, the mean bandwidth is close to the maximum, $m * a$. By introducing more levels (i.e., by increasing m , without changing $m * a$), we decrease the variance.

Our CPN model of the traffic sources is simple. It only has three pages. The first page is used for *Initialisation* of the model by reading parameters from a text file. It has a form which is similar to Fig. 2.5. The second page models the *Markov Chain Processes*, while the third page records the outcome of the *Traffic Source* by producing a text file which is used as input for the CPN model in Sect. 2.2.

Page *Markov Chain Processes* is shown in Fig. 2.7. It uses the following colour set declarations:

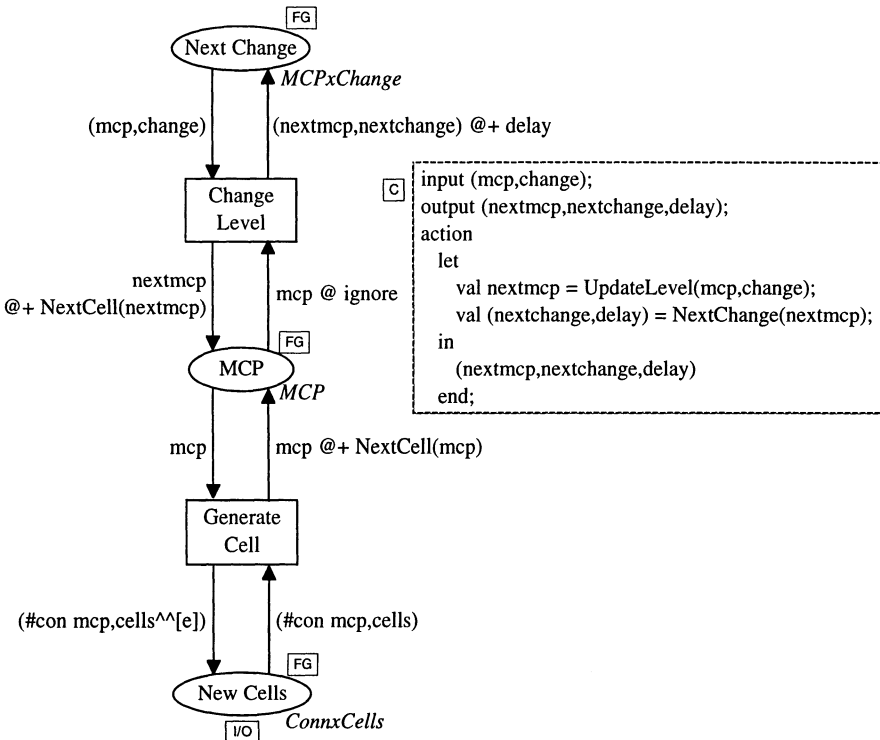


Fig. 2.7. CPN page for *Markov Chain Processes*

```

color TrafficType = with CBR | Dataflow | Databurst |
                    Videophony | Voice | Video;
color MCP = record level: Int * con: Connection * traffic: TrafficType *
                 m: Int * a: Int * u: Real * d: Real timed;
color Change = with Up | Down;
color MCPxChange = product MCP * Change;
color Cells = list E;
color ConnxCells = product Connection * Cells;

```

For each *Connection*, the input traffic is generated by one or more MCPs. Hence, we have a number of MCPs, which each is represented by a token on place *MCP*. This token specifies the current *level*, the *Connection*, the *TrafficType*, and the parameters *m*, *a*, *u*, and *d*. Place *NextChange* also has a token for each MCP. This token specifies the next change of level (via the second component of the colour) and the time for this change (via the time stamp). Finally, place *NewCells* has a token for each *Connection*. The second element in the token colour is a list in which each element represents an arriving cell.

Each occurrence of transition *GenerateCell* corresponds to a cell arrival. It adds an element to the correct list at place *NewCells*, i.e., the list that corresponds to the connection specified in *mcp*. The *MCP* token is returned with a new time stamp – specifying the time of the next cell arrival (from the corresponding MCP). The time stamp is determined by means of an ML function *NextCell* declared as follows:

```

val CellSize = 424;
fun Time(bandwidth: Int) = (real(CellSize)/(real(bandwidth) * 1024.0));
fun NextCell(mcp: MCP) =
  let
    val bandwidth = (#level mcp) * (#a mcp);
  in
    case #traffic mcp of
      CBR => Time(bandwidth)
    | Dataflow => RandomNexp(1.0/Time(bandwidth))
    | Databurst => Time(bandwidth)
    | Videophony => Time(bandwidth)
    | Voice => RandomNexp(1.0/Time(bandwidth))
    | Video => Time(bandwidth)
  end;

```

When transition *ChangeLevel* occurs (for a given MCP) the level changes, as specified by the token at *NextChange*. The *MCP* token is updated to reflect the new level, and the succeeding level change is stored in *NextChange*. The new *MCP* token and the new *NextChange* token are calculated in the code segment by means of two ML functions *UpdateLevel* and *NextChange*, declared as shown below. Note that transition *ChangeLevel* ignores the time stamps of the tokens at place *MCP*. This means that the tokens are always ready.


```

fun UpdateLevel(mcp: MCP, Down: Change) =
  {level = (#level mcp) - 1,
   con = #con mcp, traffic = #traffic mcp,
   m = #m mcp, a = #a mcp, u = #u mcp, d = #d mcp}
| UpdateLevel(mcp: MCP, Up: Change) =
  {level = (#level mcp) + 1,
   con = #con mcp, traffic = #traffic mcp,
   m = #m mcp, a = #a mcp, u = #u mcp, d = #d mcp};

local
  fun CalcNextChange(up: Real, down: Real) =
    if up = 0.0 then (Down, RandomNexp(down))
    else if down = 0.0 then (Up, RandomNexp(up))
    else let
      val nextup = RandomNexp(up);
      val nextdown = RandomNexp(down);
    in
      if nextdown < nextup then (Down, nextdown)
      else (Up, nextup)
    end;
in
  exception NextChangeError;
  fun NextChange(mcp: MCP) =
    let
      val down = real(#level mcp) * (#d mcp);
      val up = real((#m mcp) - (#level mcp)) * (#u mcp);
    in
      case #traffic mcp of
        CBR => raise NextChangeError
      | Dataflow => raise NextChangeError
      | otherwise => CalcNextChange(up, down)
    end
end;
end;

```

Page *Traffic Source* is shown in Fig. 2.8. The lower transition inspects the *New Cells* produced by the *Markov Chain Processes*, and records them on a text file. We may have several MCPs for a connection, and hence it is possible that two cells are generated very close to each other. On a physical network each cell has an extension and hence there is a *Limit* on how close the cells can be. This is achieved by place *Wait*. The limit is set to 323 nanoseconds (which corresponds to a 1.28 Gbps network).

We have used six different Traffic Types – providing a good variation over the kinds of traffic that can appear on a real ATM network. The parameters are shown in Fig. 2.9. The values for *a* are in kilobits per second. The last two rows indicate whether there is more than one level, and whether the cells are generated by a negative exponential function (as explained at the beginning of this section) or by a continuous source (with a constant time interval between cells). For

video we use two MCPs. A detailed explanation of the traffic types and their parameters can be found in [18].

When the CPN model was finished, we investigated each traffic type by performing an automatic simulation with 50 000–3 500 000 transitions occurring. On a Macintosh Quadra 950 each simulation took 1–100 hours. With the new CPN simulator described in Chap. 4, the simulations would have been many times faster.

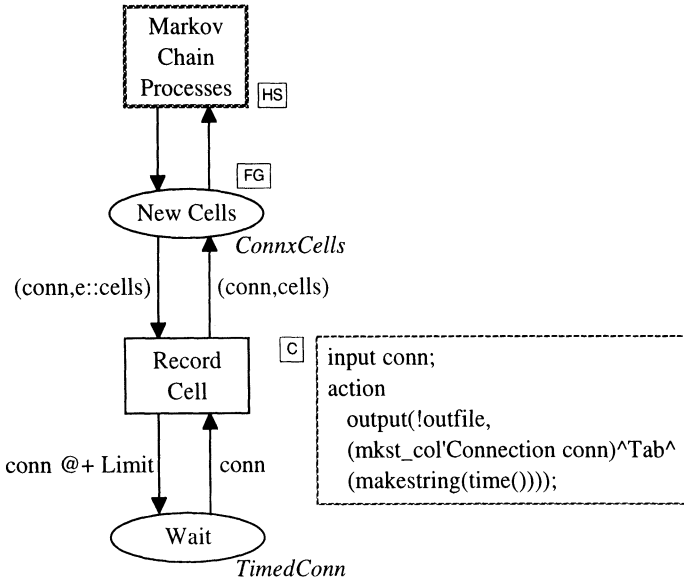


Fig. 2.8. CPN page for Traffic Source

Parameters	CBR	Data-flow	Data-burst	Video-phony	Voice	Video	
						Low	High
m	–	–	1	10	50	10	2
a	10 000	2 000	10 000	490	64	4 000	17 000
u	–	–	1.0	3.18	0.5	3.18	0.25
d	–	–	10.0	0.815	0.5	0.815	1.0
Level Changes	No	No	Yes	Yes	Yes	Yes	Yes
Cell Spacing	Cont	Nexp	Cont	Cont	Nexp	Cont	Cont

Fig. 2.9. Parameters for six different traffic types

2.4 Simulation of UPC Algorithms

In this section we describe how automatic simulations were used to investigate the appropriateness of the UPC algorithms. Here, we only present an overview of our results. A much more thorough description and discussion can be found in [17], [18], and [19].

We investigated six different kinds of traffic, covering a broad variety of the traffic types that are expected to be found in ATM networks. As we have four UPC algorithms we carried out a total of 24 automatic simulations. Each simulation covered three seconds of real time. They each contained 50 000–2 000 000 occurring transitions and took 1–8 hours on a Macintosh IIfx. With the new CPN simulator described in Chap. 4 (and a more modern machine), each simulation would take only a few minutes.

Figures 2.10 and 2.11 illustrate how good the different UPC algorithms are to tag the correct amount of cells for a videophony source. We see the total input traffic and the amount of tagged output traffic from each UPC algorithm. The horizontal lines in the upper part of the figures indicate the reserved bandwidth. The ideal algorithm only tags what is above the horizontal line. Only a small part of the total simulation is shown.

It can be seen that EWMA tags too much, while SW tags far too much. The other two UPC algorithms provide results that are close to the correct answer. The main difference is that LB is more **flexible** than TJW. For short periods, LB allows small amounts of excess traffic, without tagging. Whether this is a good or bad property depends on the application of the network. Similar results

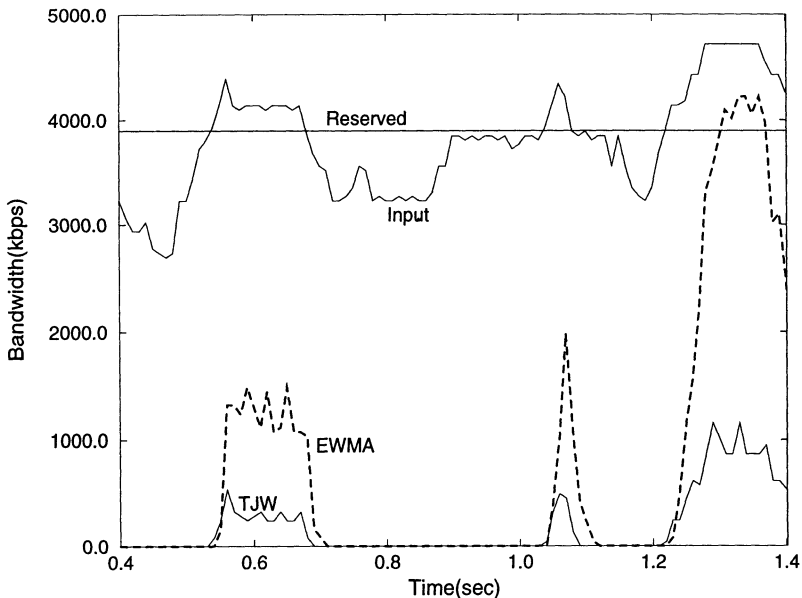


Fig. 2.10. Tagging of excess cells for EWMA and TJW

were found for the other five traffic types, and this gives us the first row in Fig. 2.12.

Next, we investigated how fast the different UPC algorithms react, i.e., detect the start/stop of an excess situation. From Figs. 2.10 and 2.11, it can be seen that SW and TJW start tagging cells immediately when an excess situation arise, while EWMA and in particular LB are slower to start. When the excess situation stops, all algorithms continue tagging for some time. EWMA stops later than the other algorithms. Similar results were found for the other five kinds of traffic types, and hence we get the second row of Fig. 2.12. The rating of LB can be discussed, since immediate start/stop of tagging prevents flexibility.

Finally, we investigated how much the different UPC algorithms delay the ATM cells. The delays were calculated by means of the time stamps recorded in the input and output files. The results of this investigation are shown in the third row of Fig. 2.12. For all bandwidths SW delays the ATM cells. EWMA and LB introduce delays that depend on the bandwidth. The maximum delay of EWMA is far too big to be acceptable, while those of LB are more reasonable. TJW is the only UPC algorithm that does not introduce any delays.

Based on the results described above (summarised in Fig. 2.12), we concluded that EWMA and SW are inappropriate as UPC algorithms. Hence, we concentrated our further work on TJW and LB, which we investigated in more detail – by varying the window width and bucket size, respectively. For each of the two algorithms we investigated four parameter values, yielding eight simulations for each of the six traffic types. Results from this investigation can be found in [17] and [18].

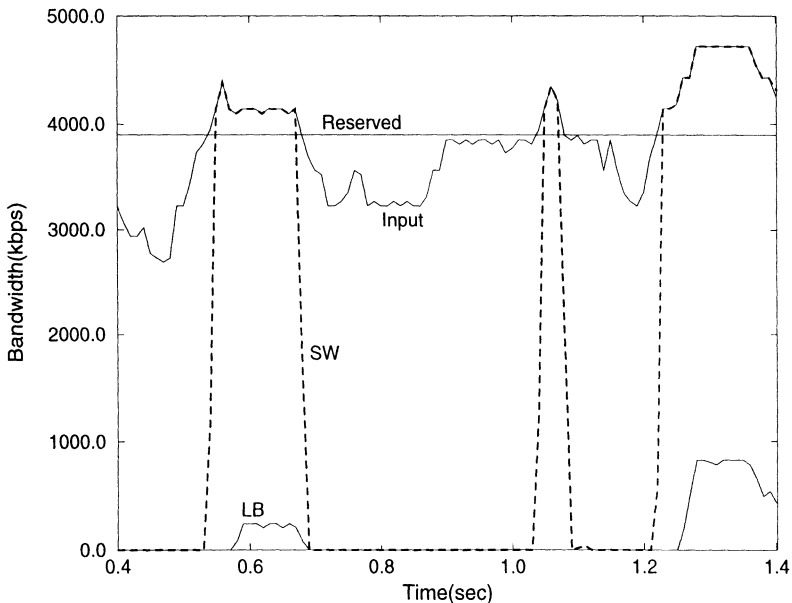


Fig. 2.11. Tagging of excess cells for SW and LB

	EWMA	TJW	SW	LB
Tagging of excess cells	** Tags too much	**** Tags correct amount of cells (inflexible)	* Tags far too much	**** Tags correct amount of cells (flexible)
Speed of reaction	* Starts too late Last to stop	*** Starts OK Stops too late	*** Starts OK Stops too late	** Starts too late Stops too late
Delay of cells	* Large delays, bandwidth dependent	**** No delays	** Small delays, bandwidth independent	*** Small delays, bandwidth dependent
Summary	*	***	*	**

Excellent = ****, Good = ***, Average = **, Bad = *

Fig. 2.12. Comparison of the four UPC algorithms

2.5 Conclusions for UPC Algorithms Project

In [48] it is argued that LB and EWMA are the most promising algorithms, because they are the most flexible. The arguments are based on an analytic solution of an equation system of the UPC algorithms, using a bursty source with low bit rate. Our results are based on a variety of bit rates and both long and short duration excess situations, covering many combinations of mean, peak, peak duration, and variance. We also investigated the delay of the ATM cells and the speed of reaction – issues not considered in [48].

From our simulations, we conclude that SW and in particular EWMA are inappropriate as UPC algorithms. EWMA tags too much in situations with large excess and it introduces cell delays of unacceptable length. LB introduces small, but bandwidth dependent, cell delays and it starts and stops tagging too late. TJW tags the correct amount of excess cells and reacts quickly to an excess situation. It stops tagging too late, but does not introduce cell delays. Hence, TJW is the best of the available UPC algorithms – although not perfect.

The use of a graphic modelling language gave us a better overview and more modelling power than an ordinary textual language, without losing the possibility of making simulations. It was easy to model the different parts of the ATM network at the required level of abstraction. The expressive power of the ML language made it easy to describe the details of the UPC algorithms. The hierarchy

constructs made it straightforward to combine the individual parts into a full model. It was also straightforward to model the Markov Chain Processes used to generate traffic. The possibility of using text files for input and output turned out to be very useful. It allowed us to transfer data from the traffic model to the UPC model, and from the UPC model to appropriate analysis tools.

We found that the time concept of CP-nets has a sound basis and is well integrated with the non-timed Petri-net concepts. It was easy to model the different delays by timed CP-nets. When we made our experiments there was no tool support for timed occurrence graphs. Otherwise, we would have liked to use these in our investigation.

It is easy to incorporate a new UPC algorithm in our model. In [18] it is outlined how this can be done for the Virtual Scheduling algorithm. The purpose of our model was to compare the UPC algorithms. If the purpose had been to specify the algorithms, we would have used a separate page for each algorithm, instead of the case-statements on page *UPC Algorithms*. That would have given a more readable but less compact model.

Chapter 3

Audio/Video System

This chapter describes a project accomplished by *Niels T. Sørensen, Bang & Olufsen A/S, Struer, Denmark, in cooperation with Søren Christensen and Jens B. Jørgensen, Aarhus University, Denmark*. The chapter is based upon the material presented in [14]. The project was conducted in 1995–96.

The purpose of the project was to investigate whether CP-nets and the CPN tools would be useful to specify, validate, and verify the protocols that Bang & Olufsen (B&O) use to connect their equipment. B&O is a renowned manufacturer of high-quality audio and video products. The BeoLink system distributes sound and vision throughout a home via a network. In this way, e.g., while doing the cooking in the kitchen, a person can remotely select and listen to a track from a CD, loaded at the CD player in the living room.

In the first part of the project, an engineer from B&O used the CPN editor to build a CPN model of a central part of the BeoLink protocol. While the model was being developed, it was validated by means of simulation. When the CPN model was finished, it was verified by means of occurrence graph analysis. No new errors were found, but a number of known problems were demonstrated, e.g., that a certain kind of installation error causes the system to malfunction. In the second part of the project, the B&O engineer developed a slightly modified version of the protocol. Again, simulation and occurrence graphs were used to demonstrate correctness. As part of this, it was shown that the two versions of the protocol are compatible, in the sense that devices using the new version can coexist, without problems, with devices using the old version.

The entire project lasted for nine months. Nearly one man-year was used, half of this by an engineer from B&O and the other half by members of the CPN group at Aarhus University. Based on the experiences from the project, CPN has been included in the set of methods that B&O uses for specification, validation, and verification of protocols.

Section 3.1 contains an introduction to the BeoLink system and the project organisation. Section 3.2 presents the CPN model of the selected part of the BeoLink system. Section 3.3 discusses how the CPN model was validated by means of simulation. Section 3.4 describes verification by means of occurrence graphs. Finally, Sect. 3.5 presents a number of findings and conclusions for the project.

3.1 Introduction to Audio/Video System

The Danish company Bang & Olufsen A/S (B&O) has a long tradition of producing sophisticated audio and video products. B&O employs more than two hundred developers, of which fifty work full-time with software. An increasing part of the software development deals with the protocols used to connect the different devices.

A relatively new invention of B&O is the BeoLink concept. A home equipped with a BeoLink system has a central room, typically the living room, where audio/video sources such as a radio, a CD player, a cassette recorder, a TV set, and a video recorder are located. The idea is that from the other rooms, such as the kitchen or the children's room, the audio/video sources of the central room can be controlled and accessed remotely. In this way, there is no need to buy, e.g., two CD players. Figure 3.1 sketches a house in which a BeoLink network connects four different audio/video devices. The figure is actually the top level of the CPN model. It contains eight places and five substitution transitions, of which *Network* is drawn in a rather non-standard way.

The project began with intensive CPN training of three engineers from B&O. They attended a six-day course, distributed over three weeks. During the course, practical application of CP-nets was emphasised much more than theoretical and mathematical aspects. The format was a mixture of lectures and practical exercises with the CPN tools. The first two days introduced the basic CPN concepts through small toy examples, using some of the introductory chapters from Vol. 1 of this book. Approximately half the time was spent getting hands-on experience at the computers, modifying and simulating small models. The engineers were experienced C-programmers, but unacquainted with the Standard ML language used by the CPN tools. Therefore, some time was devoted to an elementary introduction to this language. Also, other examples on industrial CPN projects were presented, to demonstrate the potential of the method. The two days in the second week covered more advanced topics such as hierarchical CPN models and CPN models with time. Again, half of the time was used at the computers, playing with small examples. Finally, each of the three engineers used the

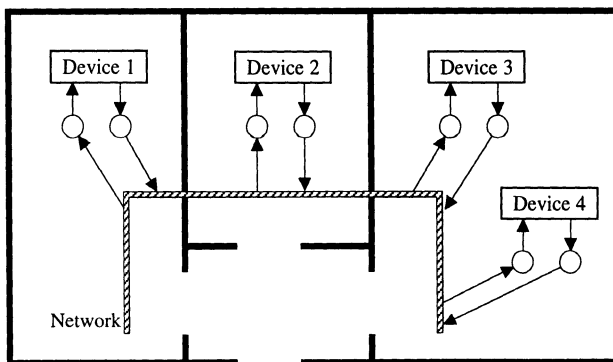


Fig. 3.1. BeoLink network connecting four audio/video devices

last two days of the course to model and simulate a small system that he already knew from his daily work at B&O. During the course, each engineer was assisted by a person who was fluent in CP-nets and the use of the CPN tools. In this way, technical problems with the use of the CPN language and the CPN tools were solved in a fast and efficient manner. From this and similar courses, it is our experience that most engineers greatly benefit from learning the CPN method and the CPN tools by practical experiments with small systems chosen from an application area they already are familiar with. In parallel with the CPN course for the B&O engineers, the CPN group at Aarhus University learned about the BeoLink concept by reading technical documentation and getting demonstrations.

At the end of the CPN course, two of the projects from the third week were selected for further work. Each project was pursued by one of the B&O engineers, while the third engineer was assigned to other duties in the company. In this chapter, we concentrate on one of the two projects. It dealt with the **lock management** protocol, which is a vital part of the BeoLink system. It is used to prevent different kinds of disorder, e.g., that track 11 on a CD becomes selected when two users simultaneously request track 1. The basic idea is that the protocol administrates a single logical **key**, also called a lock, which must be requested by any device before executing certain actions. Examples of such critical actions are selection of audio/video source (e.g., change from radio to CD player) and control of source (e.g., change of track on a CD).

The device that currently possesses the key is called the **lock manager**. The devices exchange messages over the network. These messages are called **telegrams**. Moreover, the devices interact with different users. These actions are called **events**. Figure 3.2 shows a typical sequence of actions. It could occur, e.g., when *User 1* wants to change track on a CD player, *Device 1*. To do this the user presses a button on the remote control, generating a *KeyWanted* event at *Device 1*. This device is not the current lock manager and hence it requests the key by broadcasting a *KeyRequest* telegram over the network. *Device 3* is the lock manager. It is ready to give away the key, and hence it sends a *KeyTransfer* telegram to *Device 1*. When this telegram is received, *Device 1* becomes the new

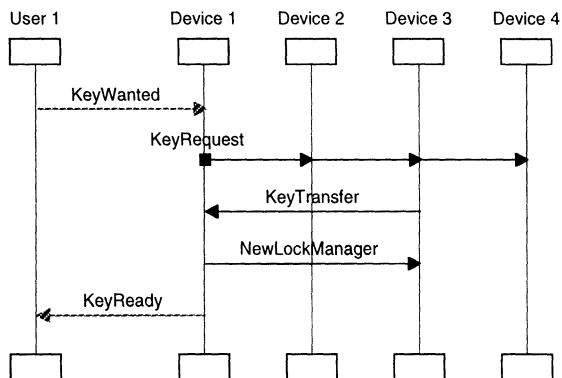


Fig. 3.2. A typical message sequence for the lock management protocol

lock manager, and it acknowledges the receipt of the key by sending a *New Lock Manager* telegram to *Device3*. Then the actions are finished by a *KeyReady* event, implying that the requested change of CD track can take place.

There are, of course, many other cases which must be dealt with. What happens if the lock manager is not ready to give away the key? What if the key is already reserved for another device? What if the key is lost? By creating a CPN model of the protocol we get a sound basis for dealing with all these problems.

Parallel to the six-day CPN course, a member of the CPN group made a rough sketch of a CPN model for the lock management protocol. The input for this work was a state/transition matrix and a flow diagram, extracted from the existing specification. The CPN model made the discussions with the B&O engineer much more concrete. He could immediately participate in the discussions of the pros and cons of this model. After the CPN course came a period with close and frequent contacts. Within the next month, the B&O engineer spent eight full days at our department, creating the first parts of his CPN model. During these visits he was primarily working on his own. However, when problems were encountered, he got assistance from a person within the CPN group. In this way, he made a very smooth and effective start on the project.

The lock management project had a duration of nine months, during which the B&O engineer used 50% of his time on it. Nearly every week, the engineer met with a person from the CPN group. This provided an opportunity to discuss different ideas and to solve technical problems. In addition, there were monthly meetings involving a larger group of people and with more general agendas. At these meetings, we discussed the CPN models and the simulation results – generating a considerable number of proposals for improvements, extensions, and further experiments. Moreover, the project plan was discussed and further developed.

In chronological order, the focus of the lock management project was on modelling, on simulation, and on occurrence graph verification. Modelling and simulation were done by the B&O engineer alone, while the occurrence graph verification was done in cooperation with the CPN group. This was primarily due to the fact that the occurrence graph tool was rather new, and hence there was only limited experience with its use in larger projects. At the end of the project, a modified version of the protocol was designed and analysed by simulation and occurrence graphs. During this phase the engineer worked totally on his own.

3.2 CPN Model of Audio/Video System

The CPN model of the lock management protocol has the page hierarchy shown in Fig. 3.3.

The *BeoLink* page contains the most abstract view of the system. It has already been shown in Fig. 3.1. The *Network* page models the primitives for sending, receiving, and broadcasting telegrams. The *Device* page models selected

aspects of the individual devices. The *User* page represents the users of the devices. The details of the lock management protocol are modelled by the *LockManagement* page and its subpages. The first five subpages describe how five different kinds of telegrams are handled. The next two subpages describe how to handle incoming events from users, who want to either get the key or release it. The last subpage handles time-outs. There are four instances of the *Device* page (and all its subpages) – one for each device. In total, the BeoLink system is modelled by 13 pages, with 46 page instances (for a system with 4 devices).

Now let us consider page *KeyRequest* shown in Fig. 3.4. Incoming telegrams are represented by a token on place *ReceiveBuffer*. The colour set is a list of telegrams. They are added to the end of the list and removed from the front, as explained in Fig. 1.18 of Vol. 1. This guarantees that the telegrams are served in a FIFO manner. When the list is non-empty a transition on one of the subpages of *LockManagement* becomes enabled. The transition in Fig. 3.4 handles *KeyRequest* telegrams in the *LockManagement* part of the protocol. This can be seen from the first line of the guard (situated inside the transition). Other kinds of telegrams are handled by transitions on other subpages.

When a telegram is received, the actions to be performed depend upon the function-lock state of the device (represented by place *FLstate*), the configuration of the device (represented by place *FLconfig*), and/or a timer (represented by place *FLtimer*). The action may change the state of the device (by modifying the colour of the token on place *FLstate*), send a telegram (by appending the

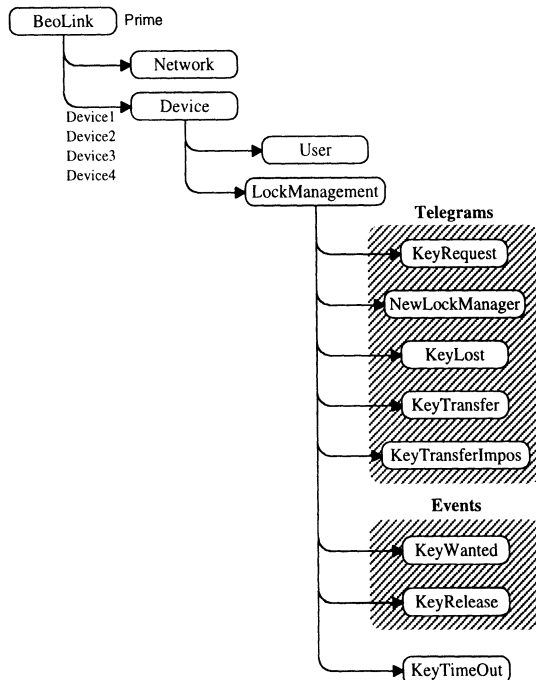


Fig. 3.3. Page hierarchy for audio/video system

telegram to the list of telegrams in the colour of the token on place *SendBuffer*), and/or set the timer (by adding a token to place *FLtimer*). The case-statements in the arc expressions in Fig. 3.4 cover three different situations:

- If *FLstate* = *Key Free*, the device is the current lock manager and willing to give the key away. The state of the device changes to *KeyTrans* and a *KeyTransfer* telegram is sent to the requesting device. The contents of the telegram are described by the second line of the guard (which determines the value of a variable *tlg1* used to generate the outgoing telegram). A timer is started with a duration determined by the constant *TransTimeOut*.
- If *FLstate* = *Key Used*, *KeyTrans* or *KeyTransSE*, the device is the current lock manager, but unable to give the key away, because it is in use or already in the process of being transferred to another device. The state of the device remains unchanged and a *KeyTransferImpos* telegram is sent to the requesting device. The contents of the telegram are described by the third line of the guard (which determines the value of a variable *tlg2* used to generate the outgoing telegram).
- For all other *FLstates*, the device is not the current lock manager, and no action is taken. This means that the incoming telegram is consumed, without changing state, without sending a reply telegram, and without starting the timer (cf. the default parts of the three case-statements).

The other four pages for incoming telegrams are similar to the page in Fig. 3.4, but the details are of course different. The three pages for user events and time-

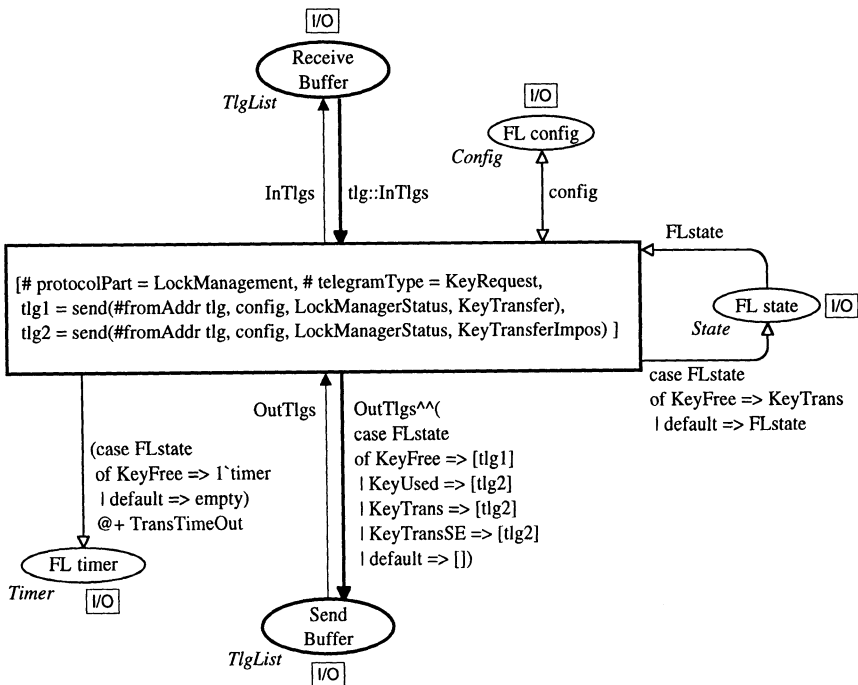


Fig. 3.4. CPN page for *KeyRequest*

outs are simpler. Each of the eight subpages of *LockManagement* contains only a single transition. Instead of using case statements, we could have split each transition into a number of transitions taking care of the different cases. One of the early versions of the CPN model did this. However, we found that the similarities and differences between the different cases were easier to see when we used only one transition per page. Similar arguments lead to the use of the two variables *tlg1* and *tlg2* to avoid repeated description of telegrams. They only become necessary because Standard ML does not allow case-statements where the left-hand part of a rule is a list of values. Otherwise we could have written:

```
case FLstate
of KeyFree => [send(#fromAddr tlg,....., KeyTransfer)]
 | KeyUsed,KeyTrans,KeyTransSE
   => [send(#fromAddr tlg,....., KeyTransferImpos)]
 | default => []
```

As explained above, the B&O model contains a page for each kind of telegram. The page describes the actions to be performed when such a telegram is received, covering all possible states of the system. It is interesting to notice that this structure is the opposite of that used for the BRI protocol presented in Chap. 9. The CPN model of the BRI protocol has a page for each state of the system (or more precisely for each state of the user/network part). The page describes the actions to be performed when the system is in this state, covering all possible kinds of telegrams.

Early in the B&O project, we compared the two structuring methods to each other. One of the main arguments for our choice was the fact that it was believed to be more likely to need new kinds of telegrams than to need new states. With the chosen structure a new kind of telegram can be added by introducing a single new page – instead of modifying a significant number of existing pages. For the BRI model the opposite choice was taken, because the CPN model is based on an SDL description which has a page for each system state.

An obvious extension to the CPN model would be to cover other parts of the BeoLink protocol, e.g., selection of audio/video source. To model this we would introduce a new group of pages, similar to the *LockManagement* page and its eight subpages (see Fig. 3.3). The only necessary modifications of existing pages would be the addition of a new substitution transition to the *Device* page and an update of the *User* page.

3.3 Simulation of Audio/Video System

In the early modelling phase interactive simulations were used intensively to investigate the behaviour of the model on the token game level. Often, the B&O engineer and a person from the CPN group sat together, at the computer, studying the detailed effects of the individual transitions. In this way the model was debugged, improved, and extended.

After approximately three months, the CPN model was considered to be reasonably correct and complete, and it was now time for more automatic simulations. To provide a fast overview of the simulation results, a transition on page *Network* was augmented with a code segment to draw message sequence charts like those shown in Figs. 3.2 and 3.5. The code segment is very simple, with just a few lines of Standard ML code, which primarily consist of function calls to the Message Sequence Chart library of the CPN simulator [15].

From this point, nearly all simulations were automatic, using the message sequence charts as feedback. To illustrate the ease and usefulness of this approach, let us consider the message sequence chart shown in Fig. 3.5. It illustrates the rules which ensure that a single key is created when the BeoLink system is started from scratch by turning the power switch on. The time unit is milliseconds. At time 112, *Device 4* requests the key by broadcasting a telegram to the other devices. Shortly after, the three other devices do the same. However, when the BeoLink system is started from scratch, no key exists, and hence none of the devices receive a positive answer from a lock manager, since such one does not yet exist. After waiting for 1500 milliseconds, *Device 3* experiences a time-out and broadcasts a *KeyLost* telegram. A few milliseconds later, also *Device 2* and *Device 4* make a time-out, broadcasting their *KeyLost* telegrams. At time 1648, *Device 1* notifies the other devices that it has generated a key and hence has become the current lock manager. This is done by broadcasting a *New Lock Manager* telegram. *Device 1* takes this initiative, because the configuration data specifies that it is the **power master**, i.e., the device that delivers electricity to the data connection of the BeoLink system.

From the discussion above, it should be obvious that the use of message sequence charts provides the B&O engineer with a fast way to overview the results of automatic simulations by offering a customised, graphical representation of

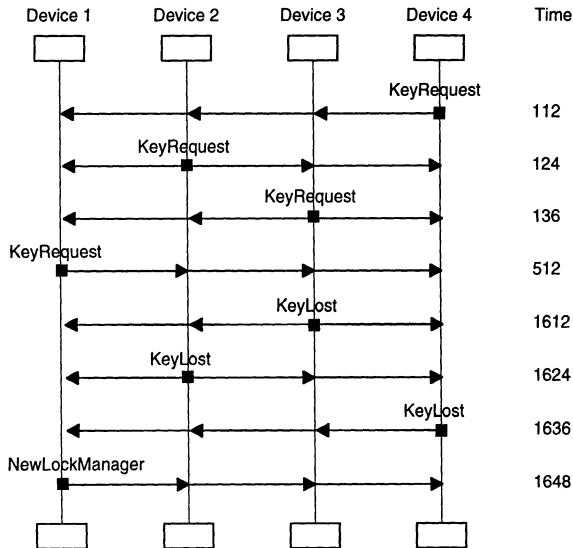


Fig. 3.5. Message sequence chart from audio/video system

the telegrams being exchanged between the devices. The use of message sequence charts is very common in the protocol area. In protocol specifications they are often used to display typical message sequences. We use message sequence charts in a rather different way – to display message sequences encountered during a simulation of the system.

After having made a considerable number of automatic simulations, the engineer was convinced that the model behaved correctly in all normal cases. His next objective was to investigate whether the protocol was able to deal with a number of special cases. As an example, he investigated the situation in which two keys simultaneously exist in the system. This may arise when certain telegrams are lost, and hence it is important that the protocol is sufficiently robust to handle such a situation. It indeed turned out to be. When a lock manager receives a *New Lock Manager* telegram, it immediately accepts that somebody else has a key and consequently destroys its own. The real-world effect of two simultaneously existing keys could be some noise in a set of loudspeakers during a short time interval, e.g., sound from the radio and the CD player at the same time. Also, it could be that a signal from a button on a remote control is lost.

A BeoLink system installed with the wrong kind of cable may accidentally have two power masters. This case was also investigated by simulations, and it was found that it may lead to a live-lock. If the users of the two power master devices never request the key, an infinite repetition of generating a key and immediately destroying it may take place in each of the two power masters. This property of the protocol was already known from experience with the real physical system. It is not that critical, because it only happens in an incorrectly installed system. However, to make the protocol more robust, it would be nice to remove the problem, and a detailed study of the simulation results showed that this would be quite easy.

From our simulation runs, it can be seen that the behaviour of the protocol is highly dependent on the length of the time-out period. If the CPN model is simulated without time, i.e., with all actions instantaneous, the protocol malfunctions. Several keys appear, and no proper lock management is in effect.

Except for the last one, the observations above were as the engineer expected. He found it very interesting to recognise the real-world behaviour of the protocol in the CPN model. He believes that B&O would have got a better protocol, had the CPN model existed prior to the implementation of the system. Then it would have been possible to experiment with different solutions easily and cheaply.

3.4 Occurrence Graph Analysis of Audio/Video System

In this section we describe how occurrence graphs were used to investigate the CPN model of the lock management protocol. As mentioned above, the CPN model is timed. The global clock advances and the protocol is intended to run forever. Hence, the occurrence graph for the lock management model is ex-

pected to be acyclic and infinite – usually, each new state has different time stamps and a different global clock value than earlier states. Hence, it is impossible to calculate all reachable states and instead we used partial occurrence graphs. To do this we used the branching criteria described in Sect. 1.7 of Vol. 2. They allow the user to specify a predicate, which is evaluated before calculating the successors of a node. One possibility is to use a predicate that tells the occurrence graph tool to find successors only for those states that exist before a certain time, e.g., 2000. Such a predicate allows us to investigate the possible actions which may occur inside an initial time interval. As we shall see below, there are several other interesting ways to use the branching criteria predicate.

In addition to using branching criteria, we also had to limit the number of telegrams that can be waiting at places *ReceiveBuffer* and *SendBuffer* in Fig. 3.4. Otherwise, we would use a lot of computation time to investigate situations in which a device uses all its time to generate a large number of requests. Another question is how many devices to include in our occurrence graph models. However, here it turned out that no modifications were necessary. It was sensible to have four devices – the same number as we had when we were doing simulations.

First, we proved that no matter what happens during the initialisation phase, the system always reaches a state where a key exists. To do this we used a branching criteria predicate instructing the tool to find successors only for those states in which no key exists. For a system with four devices, this partial O-graph had 13 420 nodes and 41 962 arcs. To facilitate experiments with different numbers of active devices, the B&O engineer augmented the CPN model with a mechanism to turn devices on and off. For each of the cases considered, it was verified that the occurrence graph was acyclic, and that all terminal nodes had a key. This means that in a finite number of steps, we always reach a state in which a key exists. For four devices, the minimum time was 1 600 and the maximum 2 000 milliseconds. Thus the analysis showed that it takes between 1.6 and 2.0 seconds before a key is generated. These times are in accordance with the times known from the real world, i.e., the actual time it takes from power is switched on until a BeoLink system is ready. This indicates that reliable performance measures are indeed derivable from the CPN model.

The next step in our protocol verification was to try to show that there is always at most one key. To do this, we used a branching criteria predicate to construct a partial occurrence graph containing all states that exist up to the first key transfer. We had already verified the initialisation phase, and hence we started from a state with a single key and a single lock manager. For a system with four devices, this partial O-graph had 2 578 nodes and 5 335 arcs. We checked that there were no dead markings and that the bounds on all places were as expected. Unfortunately, this does not provide a total proof of the at-most-one-key property. One could hope to be able to use our partial O-graph as the basis in a proof using induction over the number of key transfers. However, this is not that easy. After a key transfer, the system may be in many different states, which all must

be investigated as the initial marking of a new occurrence graph. It turned out that there were too many such states to make the approach usable in practice. However, even though we did not succeed in making a full verification of the at-most-one-key property, our confidence in its correctness was strongly increased.

When the existing version of the lock management protocol had been investigated, as described above, the B&O engineer used the CPN tools to design and investigate a small revision of the protocol. Approximately once a year, B&O releases a new series of products. However, the typical customer does not replace all devices at the same time. Hence, new and old devices must be able to coexist. A radio some years old and a brand new TV set installed by the dealer yesterday must be able to function together in the same BeoLink system without causing communication problems. Without having an executable model of the protocol, it is cumbersome and expensive to make revisions, since a system must be implemented before an efficient compatibility test can be made.

In the revised protocol, the concept of a power master is abandoned. Instead, there is now a **video master** and/or an **audio master**. The video master has the obligation and the right to generate a new key, immediately when it discovers that none exists. Also the audio master may generate a new key, but only after some time period has elapsed. This asymmetry between the masters is introduced to ensure that only one key is generated.

The CPN model of the new protocol was created by the engineer without any support from the CPN group. At this stage, the engineer was fully capable of working with the CPN method and tools all by himself. It took only about two weeks (four weeks half-time) to design the new version of the protocol, and to create and investigate the corresponding CPN model. To model the revised protocol, the engineer modified the four pages: *NewLockManager*, *KeyLost*, *KeyWanted*, and *KeyTime Out*. Since, he wanted to experiment with a mixture of old and new devices, the four pages now existed in two slightly different versions, which both were part of the CPN model. By a simple modification of the *LockManagement* page, the engineer created a mechanism by which he could vary the mixture of new and old devices, without changing the structure of the CPN model. When the engineer presented the results of his efforts for the CPN group, the behaviour was conveyed using message sequence charts. Many simulation runs had been made, investigating five configurations with different mixes of new/old devices and presence/absence of video/audio masters. The simulations confirmed that the new version of the protocol works correctly, both when all devices are new and when the devices are mixed.

The revised CPN model was also verified by means of occurrence graphs, in a similar way as the original model had been. The engineer considered the same five configurations as he had simulated. The occurrence graph verification was done by the engineer alone, and it only took about half a day.

3.5 Conclusions for Audio/Video Project

Towards the end of the project, the B&O engineers wrote a status report for their managers. Here they summarised the goals and the results of the project. The goal was to improve B&O's methods for specification, validation, and verification of protocols. The main conclusion is that this goal was met. Being able to create well-defined, graphical, and executable models is recognised as a valuable basis for obtaining better results faster.

The project demonstrated that the CPN method can be presented to industrial engineers in such a way that they are able to work independently after a reasonably short amount of time. The hardest part for the B&O engineers was to cope with the Standard ML language. As experienced C-programmers, they were used to think in terms of imperative languages, and the shift to a functional language with recursive functions turned out to involve some difficulties. A lot of the consultancy work of the CPN group consisted in assisting with Standard ML tasks. We believe that many of these difficulties could have been avoided, had we devoted two extra days in the initial CPN course to Standard ML. In future courses of this kind, we intend to do this.

The project also demonstrated the usefulness of having customised, graphical feedback from simulation runs. The message sequence charts made the work of the B&O engineer more pleasant and also much more efficient. Moreover, they allowed him to present and discuss the simulation results with colleagues who were totally unfamiliar with CP-nets.

Finally, it was demonstrated that occurrence graphs of timed CP-nets can be used in an industrial setting. During the investigation of a revised version of the lock management protocol, the engineer used this verification technique on his own. He investigated five different configurations, establishing strong evidence that devices running the new protocol can coexist with devices running the old version.

B&O's participation in the described project was triggered by the fact that communication protocols are gaining more and more attention within the company. Today, the BeoLink concept offers access to sound and vision throughout a private home. In the future, B&O anticipates that many audio/video products will be used to disseminate more general kinds of data, e.g., via access to the Internet. For this purpose a number of suitable application-level protocols must be developed.

Chapter 4

Transaction Processing and Interconnect Fabric

This chapter describes a project accomplished by *Ludmila Cherkasova, Vadim Kotov, and Tomas Rokicki, Hewlett-Packard Laboratories, Palo Alto CA, USA*. The chapter is based upon the material presented in [12]. The project was conducted in 1993.

We present our experiences in modelling an on-line transaction processing system and a scalable interconnect fabric by means of CP-nets and the CPN tools. We show how net modelling is used both for performance evaluation of existing applications and for architectural exploration of proposed hardware designs.

We have performed industrial-sized simulations of tens of thousands of transactions running on databases with millions of records. On-Line Transaction Processing applications (OLTP) were chosen because the OLTP workloads emphasise update-intensive database services with up to thousands of concurrent transactions per second. The goal of the experiment was to develop a methodology for the net modelling of OLTP-like applications – investigating the ability of different computer systems to meet the OLTP benchmarks requirements.

Petri-net based tools are very useful in the initial design phases when key decisions about the system structure and behaviour need to be evaluated from simulation studies. They provide both graphical and mathematical system modelling views which inherently enable both quantitative and qualitative analysis of the design. To demonstrate this claim, a CPN model for performance analysis of a particular scalable interconnect fabric is presented.

Section 4.1 presents the basic ideas behind the OLTP system and discusses how an efficient simulation model can be obtained. Section 4.2 presents the CPN model of the OLTP system. Section 4.3 contains an introduction to the basic ideas behind a scalable interconnect fabric. Section 4.4 presents the CPN model of the interconnect fabric. Finally, Sect. 4.5 presents a number of findings and conclusions for the project.

4.1 Introduction to Transaction Processing

On-Line Transaction Processing (OLTP) characterises a category of information systems with:

- Multiple interactive terminal sessions.
- Intensive I/O and storage workload.
- Large volumes of data stored in databases.
- Transactions, i.e., well-defined units of activity satisfying such properties as atomicity, consistency, durability, and serialisability.

We used CP-nets to model an OLTP system known as the TPC-A benchmark, [30]. We concentrated on specification and performance evaluation.

The TPC-A benchmark describes a hypothetical bank that has multiple branches with multiple tellers at each branch. The bank has many customers, each of whom has an account. The database represents the cash position of each entity (branch, teller, and account) in correspondent records and a history of recent transactions run by the bank. The transaction represents the work done when a customer makes a deposit or withdrawal against his or her account. The transaction is performed by a teller at some branch. TPC-A performance is measured in terms of transactions per second (tps). The main performance characteristics are as follows:

- There is 1 branch, 10 tellers, and 100 000 account records per reported tps.
- The response time must be less than 2 seconds for 90% of all transactions.
- Transaction arrivals are determined by a truncated exponential random distribution.
- Realistic implementations should be able to run thousands of tps.

Most of the model of the TPC-A benchmark is of the database and computer system on which the benchmark runs. With multiple concurrent transactions, the system we model generates several different types of concurrent processes including:

- Top-level application processes.
- Processes in the operating system (supporting transaction processing).
- Updating processes in the database (storing the accounts tables).
- Processes supporting I/O and communication activities (which are substantial in this application).

To provide transaction integrity, access to branch, teller, and account records is locked, so that multiple transactions cannot access the same record simultaneously. The database logs store data on transactions performed in order to allow the system to reconstruct database contents after a crash. When the log information has been written, the changes generated by the transaction (or a group of transactions) are committed, and the correspondent records are unlocked. The update activity is substantial for TPC-A, hence such database processes as page cleaners are significant. These processes “clean” the database record buffer by writing dirty (i.e., modified) pages to disk.

For our project, we used the CPN tools described in Chap. 6 of Vol. 1. We found that these tools provided all the necessary capabilities, such as:

- Means for adequately detailed and comprehensive specification of the system (in different stages of the design).
- Means for formal verification of system properties.
- Simulation vehicles for performance evaluation.

However, it was not a trivial task to directly combine these capabilities in a single model for a system of this complexity. For the TPC-A system, we constructed a number of related CPN models, beginning with a specification model (preserving the basic structural features of the system) and ending with a detailed behavioural model (used for performance evaluation).

The specification model describes a typical, basic schema of transaction processing to be implemented on different multiprocessor architectures. The model has two main parts representing:

- A general, functional schema of OLTP transactions, maximally independent of possible implementations and with a common schema of the data access using typical I/O procedures.
- The architectural features of a database and its implementation on a specific configuration of a computer system.

The central point of the functional part of the model is related to the locking mechanisms which prevent multiple transactions from simultaneously accessing the same database records. Each transaction, referring to a specific account, locks the access to this account record, then to the teller record associated with this transaction, and, finally, to the corresponding branch record. When the transaction commits (or completes), it unlocks all three records.

The database logs are formed and stored on a disk to allow the system to roll forward from any consistent state (perhaps from a recent backup or checkpoint) to the correct database state reflecting all committed transactions and no uncommitted transactions. For efficiency, a group commit scheme is used to collect several transaction logs for writing to a disk. This allows the use of a smaller number of efficient large writes rather than numerous small disk writes. The number of transactions in a group commit is normally fixed, but a time-out mechanism prevents excessive waiting for any single transaction.

A straightforward specification of the locking mechanism for accounts, tellers and branches is shown in Fig. 4.1. The variable t denotes a transaction. The places *Accounts*, *Tellers*, and *Branches* contain tokens representing the individual accounts, tellers, and branches, respectively. An account is locked when transition *Lock Account* removes the corresponding token from place *Accounts*. The function *Account(t)* determines the account to be used for the transaction t . Analogously, transitions *Lock Teller* and *Lock Branch* lock tellers and branches by removing tokens from places *Tellers* and *Branches*.

When the locking has been done the transactions are executed. This is modelled by the substitution transition *Transaction Processing* that represents all the lower levels of the hierarchical model. The tokens emerging from the substitution transition represent completed transactions and enable the transition *Unlock*

which returns the tokens that were removed from places *Accounts*, *Tellers*, and *Branches*.

The CPN model presented in Fig. 4.1 is natural and straightforward. However, it turned out to be inadequate for our simulations – when we had hundreds of branches, thousands of terminals, millions of accounts, and thousands of ongoing transactions. When the arriving rate for transactions increased, the number of tokens that queued in the input places of transitions *Lock Account*, *Lock Teller*, and *Lock Branch* increased. This led to a significant degradation in the simulation speed. The problem was caused by the fact that the CPN simulator, at that time, was not designed to effectively handle CPN models in which some places contain thousands of tokens. Since then the algorithms and data structures of the CPN simulator have been totally redesigned. This means that the new CPN simulator is much more efficient for CPN models with large amounts of tokens.

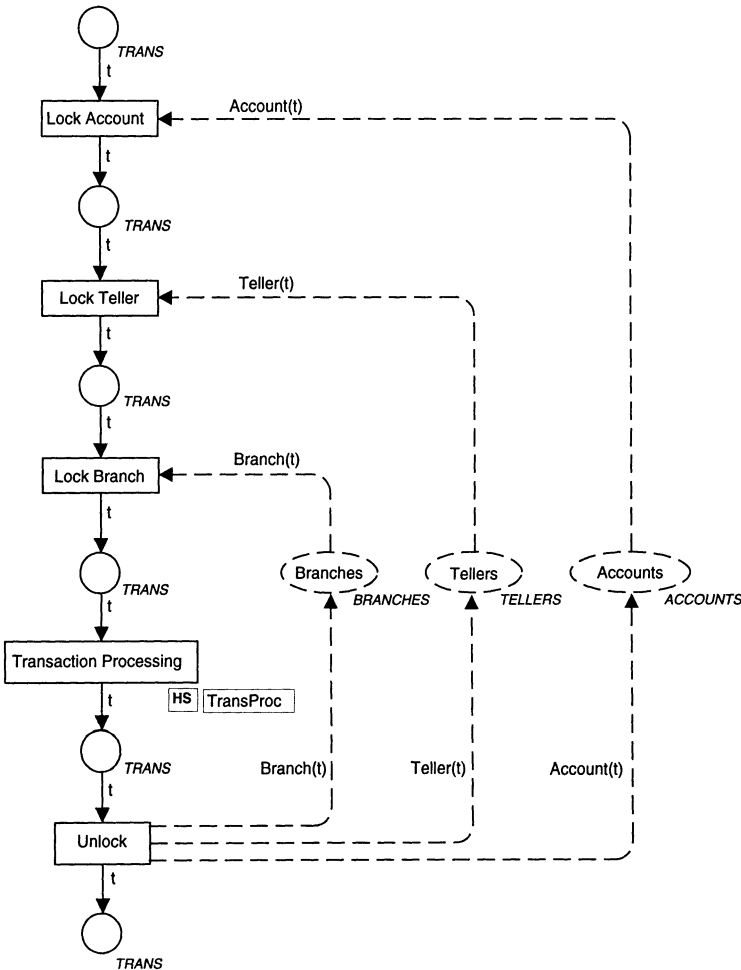


Fig. 4.1. CPN page for locking mechanisms

Such models often run more than one thousand times faster with the new simulator. This means that a simulation takes a few seconds instead of several hours.

To make our simulations efficient, we modified the specification model – without changing the behaviour of the CP-net. If we had been able to use the new CPN simulator, these modifications would not have been necessary.

By using our specific knowledge of the system we were modelling, we could modify the CP-nets to dramatically reduce the amount of brute-force search for enabled bindings. As an example, consider the locking scheme from Fig. 4.1. The original simulation engine selects a particular token representing a transaction and then considers all possible lock tokens, looking for a match. Luckily, we know some things about the lock tokens that we can use to speed this up:

- The account identifiers come from a limited integer range.
- The actual lock tokens do not carry any additional data values. It is only the absence or existence of a particular lock token that matters.
- The lock tokens do not carry time stamps, and hence they are always ready to be used.
- There are a large number of lock tokens (on the order of tens of millions).

Because of these characteristics, we can perform a simple transformation to speed up the simulation. Instead of using an individual token for each account lock, we now use a single token to represent the entire lock table. This means that the place *Account Lock Table* in Fig. 4.2 only contains a single token. Analogously, *Teller Lock Table* and *Branch Lock Table* only have one token each, representing the lock tables for tellers and branches. The guards of the three lock transitions check that the account, teller, and branch in question are not locked. Each of the three lock tables is represented as a data structure programmed directly in Standard ML and accessed via a reference variable. In this way the locking operations become very efficient to simulate. The code segments of the lock transitions update the lock tables to reflect that the records are now locked. The unlock operation is implemented in a similar way. Transition *Unlock* has no guard, since no check has to be performed. The code segment modifies the three lock tables to reflect that the account, teller, and branch no longer are locked.

Various representations of the lock table were considered, including arrays, bit vectors, lists, and hash tables. Since typically a very small percentage of the database is locked, we decided to use a hash table, listing only those records that were locked. Thus, the *locked?* database predicate was implemented with the *lookup?* hash table predicate, the *lock* operation with the *add* hash table operation, and the *unlock* operation with the *remove* hash table operation. We used an adaptable hash table implementation where each of these operations was very fast (constant time on average).

Representing an entire lock table (which is just a set of record locks) is not as simple as representing a single record lock, but neither is it essentially complicated. The net change to our model is the addition of a few arcs, the guard expressions, and the code segments. In addition, a small amount of Standard ML code was written to implement the hash tables. By making the modification we

obtained a tremendous improvement of the simulation time. If we have m database records and n waiting transactions, the CPN simulator has to consider $m * n$ possible bindings for a transition in Fig. 4.1 while it only has to consider n bindings for a transition in Fig. 4.2. Since m is typically in the millions and n is typically just a few, this yields a dramatic speedup. In addition, it is a general scheme that can be (and has been) used in many other portions of the model. The improved CPN simulator actually uses the idea described above. The tokens of *Accounts*, *Branches*, and *Tellers* are represented in balanced trees of a kind that can be effectively searched and updated.

In addition to the hash tables described above we have also modified the CP-nets to handle waiting processes in a more efficient way. The CP-net in the

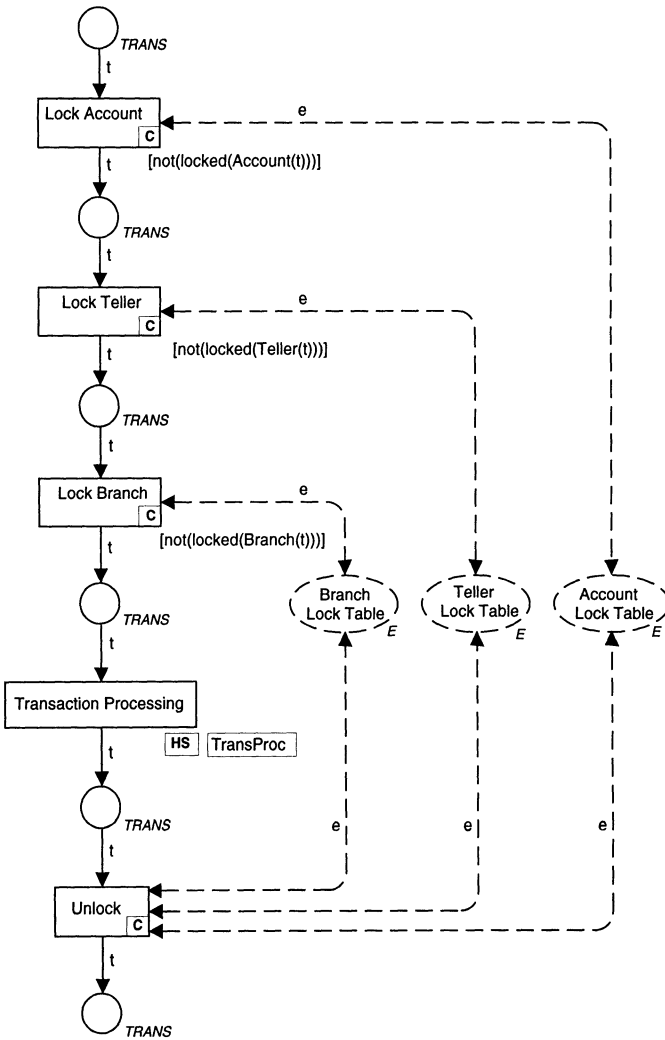


Fig. 4.2. More effective CPN page for locking mechanisms

left-hand part of Fig. 4.3 has a place *Trans Req* which may contain a significant number of tokens representing transaction requests waiting at transition *Process Request*. Each token carries a time stamp indicating the earliest moment at which the request may be processed. This means that most of the tokens at *Trans Req* are unavailable to the transition *Process Request* because they have a time stamp which is higher than the current model time. To avoid calculation of bindings for these tokens, we insert an extra transition as shown in the right-hand side of Fig. 4.3. This moves the problem of the extra binding calculations from *Process Request* to *Time Filter*. However, the latter transition only has one input place and hence there are many fewer bindings to be considered. The new improved CPN simulator mentioned above represents the marking of a timed place by means of a priority queue, where the tokens are sorted according to the time stamps. This means that the new simulator only calculates bindings for those tokens which are available. Hence, there is no longer any need for the kind of transformation described in Fig. 4.3.

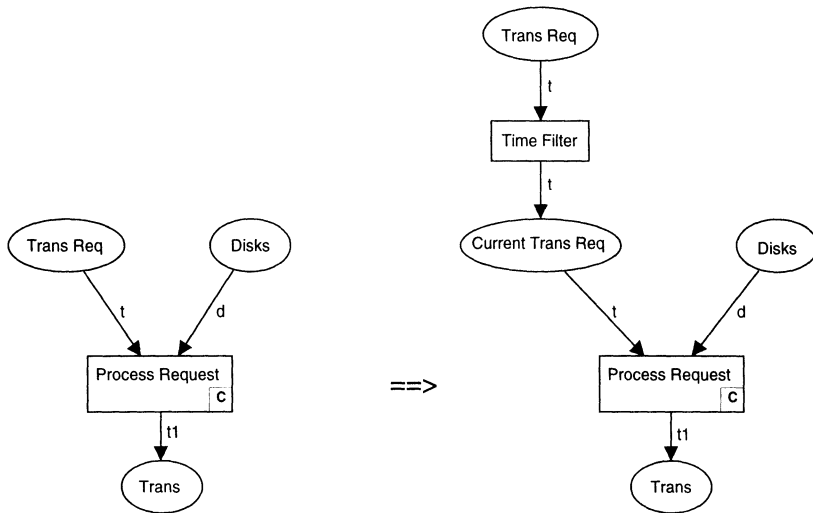


Fig. 4.3. Transformation to get a more effective timed CPN model

4.2 CPN Model of Transaction Processing

The core of our OLTP simulation model consists of 9 pages containing the following:

- 1) Declarations of types, variables, functions, and reference variables (approximately 1000 lines of ML code).
- 2) Initialisation allowing us to run the model with different system parameters.
- 3) Overall transaction processing schema (shown in Fig. 4.4).

- 4) Record locking and access for account, teller, and branch records (shown in Fig. 4.5).
- 5) Disk interface that prepares the disk access requests and collects and distributes the ready data.
- 6) Disk service based on HP-UX scheduling strategy (shown in Fig. 4.6).
- 7) Page cleaning (shown in Fig. 4.7).
- 8) Group commit and time-out, specifying a strategy for writing log information to disk and committing transactions.
- 9) Collection of results and statistics.

In the following we describe some of these pages. We start with the overall transaction processing schema in Fig. 4.4. New transactions are generated by transition *Generate Transactions*. The transition *Prepare Account Request* calculates the parameters to be used during the account update which is performed by transition *Get Account Data*. Analogously, the transaction prepares and gets data for the teller and for the branch. Finally, the transaction goes through *Group Commit* and *Update Statistics*. Note that the substitution transitions *Get Account Data*, *Get Teller Data*, and *Get Branch Data* all refer to the same page.

This page, called *Record Lock*, is shown in Fig. 4.5. It is the most complex page of the entire CPN model. Requests for records arrive at place *P1* (at the top). The requested record is either for an account, a teller, or a branch (depending on the page instance). The transition *Get Record Lock* checks that the record is not locked and then it locks the record. The *Lock Table* is represented by means of a hash table, as explained in Sect. 4.1. The guard and the code segment accesses the hash table via a reference variable. The place *Lock Table* is a global fusion place. This means that the three instances of transition *Get Record Lock* (on the three page instances corresponding to substitution transitions *Get Account Data*, *Get Teller Data*, and *Get Branch Data*) have mutual exclusive access to the lock tables.

When the record has been locked the request proceeds to place *P2*. Here there are several possibilities. If the record is already in memory (i.e., record already has been buffered) then the left-hand transition *Bypass Disk Access* will occur. Otherwise the transitions *Queue Request* and *Reserve Page* will occur. The first of these transitions adds the request to a FIFO-queue at place *P3*. Because of the potentially large number of tokens in this queue, we are again faced with the importance of efficient modelling because a more straightforward approach may dramatically slow down the speed of simulation.

Before a record is read from disk, a free memory page must be reserved for the data. This is done by transition *Reserve Page*. The marking of place *Buffer Table* indicates the number of free memory pages. If there are no available pages *Reserve Page* is disabled, because of the guard $[i>0]$. This implies that the requests in place *P3* are delayed – waiting on the page cleaning processes to free a memory page. Page cleaning is represented on a separate page to be presented below. It updates the marking of the fusion place *Buffer Table*.

When a memory page has been reserved, the request proceeds to place *P4*, which is input socket for the substitution transition *Record Read*. This transition refers to page *DiskAccess* which models the disk behaviour.

When the record is found and read from the disk, the request proceeds to place *P5*. The fact that the data is now available in memory is recorded (in the

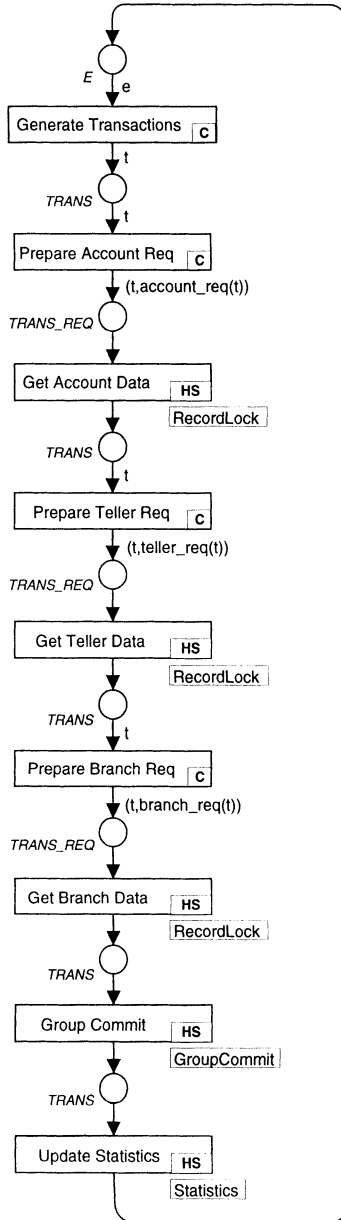


Fig. 4.4. CPN page for *Transaction Processing*

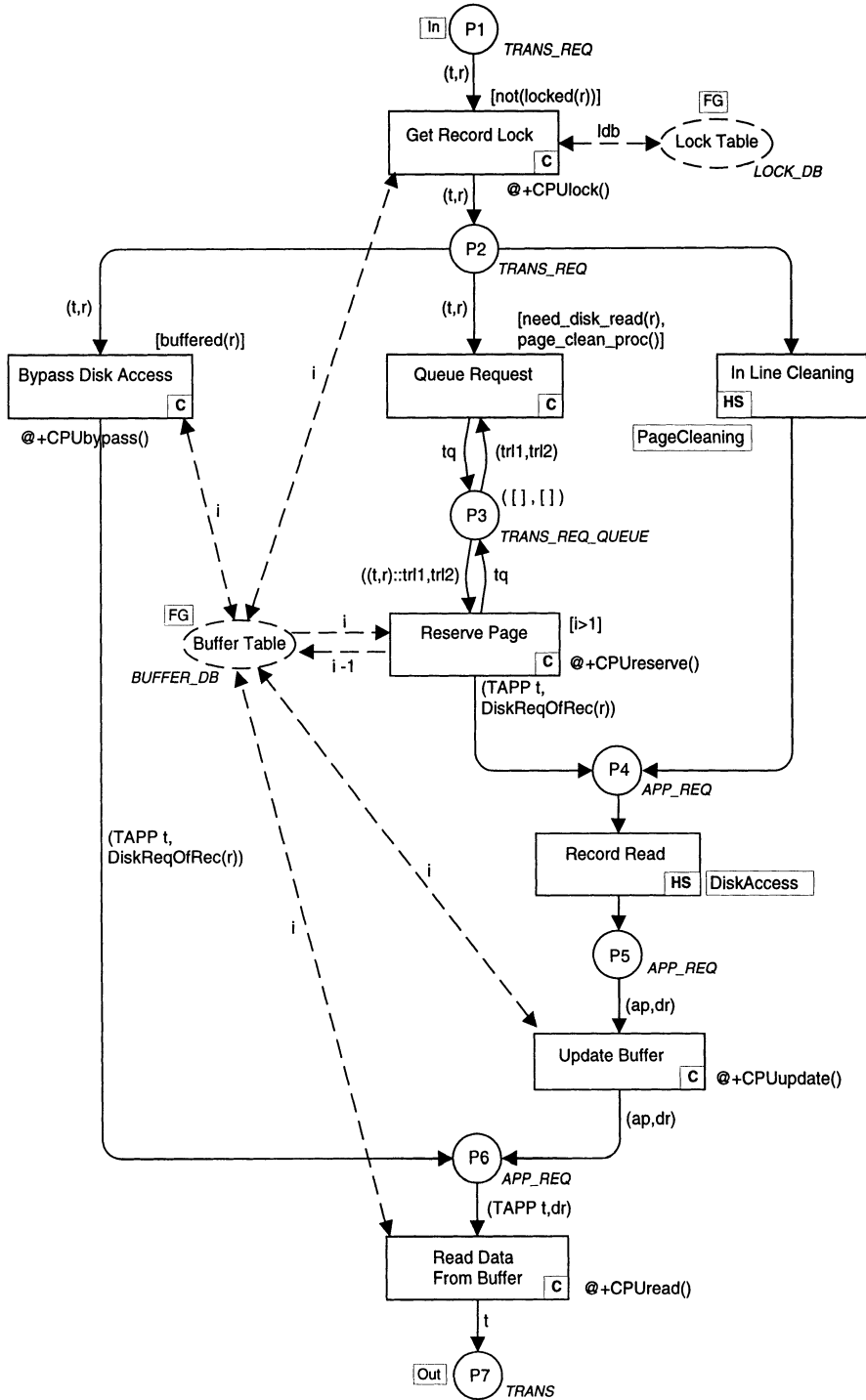


Fig. 4.5. CPN page for RecordLock

lock database) by transition *Update Buffer*. Then the request proceeds to place *P6*, which indicates that the data is buffered. The last stage consists in reading a record from the memory. This is modelled by transition *Read Data From Buffer*.

The substitution transition *In Line Cleaning* (in the right-hand part of Fig. 4.5) represents a page cleaning strategy where the transaction itself is responsible for finding and cleaning a dirty page. The transition is included to investigate the performance of different page cleaning strategies.

The token in place *Buffer Table* carries a time stamp which is used to model the sharing of the CPU. All operations that require CPU time update the time stamp of the token in *Buffer Table*. This is done in the time expressions of the transitions by means of a number of ML functions (*CPUlock*, *CPUbypass*, etc.). Each ML function determines the length of the corresponding operation, i.e., the period of time during which the CPU is busy executing the operation. The actual time delays are input parameters of the model and hence they can easily be changed.

Figure 4.6 shows page *Disk Service*, which is rather straightforward. Place *Requests* receives requests from different parts of the CPN model – it might be a request for a branch, teller, or account record, or a request for writing to log or writing a dirty page. In order to know where to return the result, each kind of requests carries a unique identifier (in the fourth element of the product colour set *TICKET_REQ*).

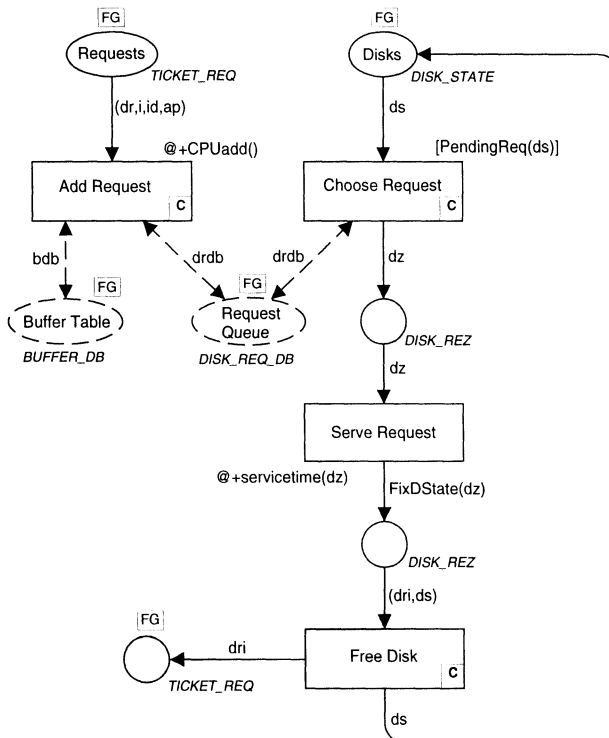


Fig. 4.6. CPN page for *DiskService*

The occurrence of transition *AddRequest* puts the request in a *Request Queue*. Place *Disks* contains a token for each disk in the system. A *DISK_STATE* is a pair indicating the disk number and the current head position. How disk requests are organised in a queue and how they are chosen from this queue depends on the concrete implementation of the system. We used the HP-UX scheduling strategy that orders the requests according to the disk position of the data. This means that a new request may be satisfied before an older request – if the data of the new request is closer to the disk head. To change the scheduling strategy, a single ML function has to be modified – without changing the structure of the CPN model. One of the attractive features of CP-nets is the fact that it is so easy to change interpretations.

Figure 4.7 shows how *Page Cleaning* is performed. Place *Cleaning Processes* contains a token indicating the number of idle cleaning processes, while place *Buffer Table* represents the memory to be cleaned. Transition *Start Cleaning* activates a cleaning process, searches the memory, and moves an unused dirty page to place *P8* which is input socket for the substitution transition *Write Dirty*. This transition refers to page *Disk Access* which again refers to page *Disk Service* in Fig. 4.6. This means that the dirty page is queued to be written to a disk, in the way described above. When the dirty page has been written to the disk, the cleaning process is finished. Transition *Finish Cleaning* increases the number of available clean pages (in place *Buffer Table*) and returns the cleaning process to idle (in place *Cleaning Processes*). Again, it is easy to change the interpretation,

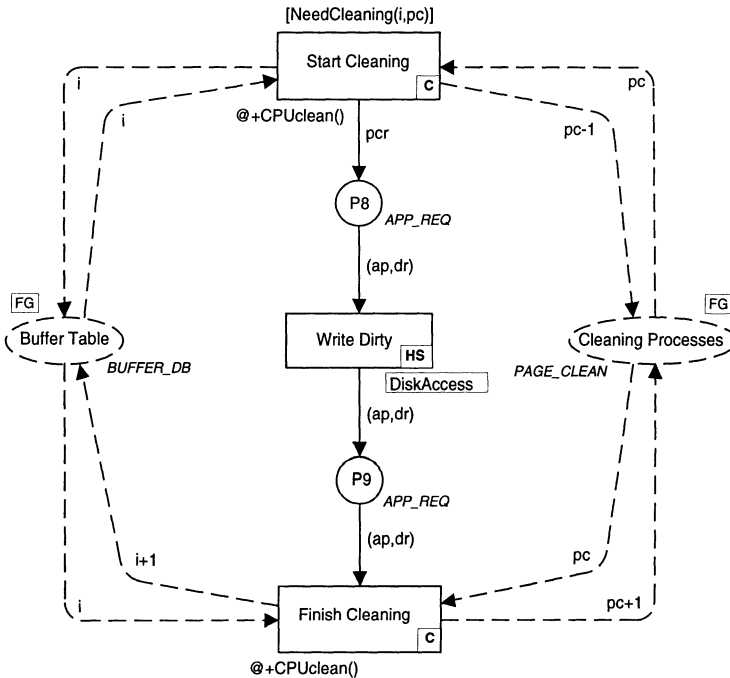


Fig. 4.7. CPN page for Page Cleaning

i.e., the details of the page cleaning, without changing the structure of the CPN model. It is easy to model different page cleaning strategies and to change the number of page cleaning processes. Hence, tuning is possible by determining the most appropriate parameters and strategies.

The simulation model briefly presented in this section contains a modest amount of programming. However, the model is in many ways universal and flexible to use and modify. One of the crucial points in the successful use of CP-nets is to determine how to divide the information between the net structure, the net inscriptions and the declarations. As an example, the disk service model represents the basic disk features by means of net structure and net inscriptions, while the detailed request scheduling is captured by an ML function (i.e., by writing ordinary sequential code). As explained, this makes it easy to experiment with different scheduling strategies.

The goal of the OLTP simulation model was to provide us with the necessary information to analyse the functioning of the system under different transaction rates and conditions – to identify bottlenecks and potential inefficiencies in the system design. In order to accomplish this goal, we collected detailed statistics showing the progress of the individual transactions. This information includes, for example, how long each transaction spent obtaining each record lock, how long it spent waiting for a disk or buffer read, how long it took to make a group commit, etc. We also collected history information on disk requests, including how long each request spent in the disk queue and how long it took for the physical disk to satisfy the request. All this information was written to files and we developed a few simple tools to analyse these history traces, to perform some simple statistical analysis, and to illustrate the results graphically.

An important property of CP-nets is how easy it is to extend an existing model by adding additional abstraction levels. To illustrate this we show how our model can be extended to cope with a multiprocessor architecture. To do this we modify the basic transaction scheme in Fig. 4.4 so that the three topmost substitution transitions now refer to a new page, called *Net Access* (instead of referring to *Record Lock*). The new page is shown in Fig. 4.8. The left-hand transition is a substitution transition which refers to page *Record Lock*. This models

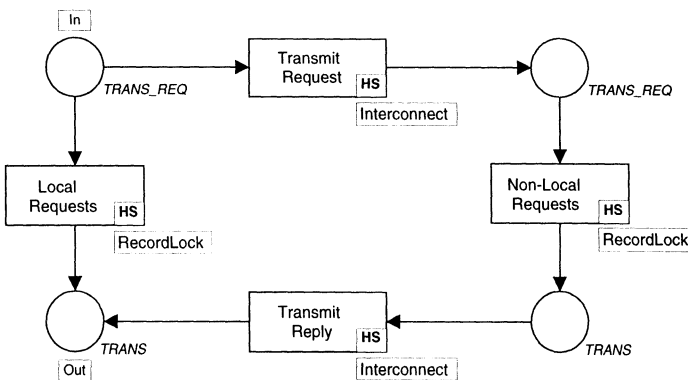


Fig. 4.8. CPN page for *Net Access*

requests that can be satisfied locally. The remaining three transitions model requests that have to be handled by another computer. The upper transition transmits the request. The right-hand transition performs the request by means of page *Record Lock*. Finally, the lower transition transmits the result back to the original site. The upper and lower transitions are substitution transitions. This allows us to model the detailed behaviour of the interconnection network. If this level of detail is not desired, the two substitution transitions can be replaced by ordinary transitions with an appropriate time delay.

Making a detailed model of the interconnect network allows us to identify its properties and potential bottlenecks under the specific OLTP workload. We can also experiment with it and change its parameters to improve the performance. In addition to the modification described above, we also have to modify the page for *Group Commits*, because the group commit protocol is different for a multiprocessor systems.

4.3 Introduction to Interconnect Fabric

The second part of the chapter presents a CPN model for performance analysis of a scalable multiprocessor interconnect fabric. The CPN model of the interconnect can be directly integrated into the CPN model of the OLTP system. This is done by letting the transitions *TransmitRequest* and *TransmitReply* in Fig. 4.8 be substitution transitions referring to the most abstract page of the *Interconnect* model. However, we did not do this. Instead we made an independent, detailed analysis of a stand-alone CPN model of the interconnect fabric. The results from this study was then used to specify appropriate time delays for transitions *TransmitRequest* and *TransmitReply* (which were ordinary transitions instead of substitution transitions). In this way we obtained a good approximation of the performance of the interconnect – without including too much overhead in the simulation model of the OLTP system.

The fundamental component of fabric is an adaptive routing interconnect, called R2, based on the interconnection scheme used in the Mayfly multiprocessor [20]. CP-nets were used to study both behavioural and performance aspects of the proposed R2 design. The R2 interconnect topology is a continuous hexagonal mesh which permits each node in the fabric to communicate with its six immediate neighbours. The hexagonal mesh inherently has three axes of interconnect and the topology permits the construction of similarly hexagonal surfaces. Off-surface connections are wrapped to the opposite side of the surface to create the desired continuous mesh property. Figure 4.9 illustrates a sample surface containing 19 nodes (where only one axis is wrapped for clarity). The topology permits arbitrary surface sizes to be constructed and also permits an arbitrary number of surfaces to be connected by the off-surface switch. The present discussion will be restricted to single surface properties.

Messages are split into packets of fixed length. For this study the packet size is 36 words. A four-word header contains the source and destination addresses, leaving 32 words for the payload. Each node in the fabric contains a single R2

adaptive routing device [26], whose design is the subject of our study. Each node in the surface is assigned a unique location which is used to calculate the appropriate route through the fabric. We are interested in router designs which scale to large node numbers. Hence, we do not use a global route table. Instead each node chooses one of its neighbours as the next node to be visited. The choice is made by a simple algorithm described in [21].

In order to restrict the scope of this discussion, certain R2 design aspects will be fixed as follows. Each node will have a FIFO queue containing twenty packet-sized buffers. Moreover, the node has an adaptive router controller capable of calculating the appropriate exit ports. There are fourteen different ports. Twelve external ports support bi-directional traffic on the six possible paths to adjacent nodes in the interconnect topology. The other two internal ports permit bi-directional traffic to the local CPU. There are also two crossbar switches. One connects the internal ports to the buffer pool and the other connects the buffer pool to the external ports. The head and tail of each FIFO buffer can be active concurrently, as can communication on all fourteen ports.

Adaptive routing is inherently non-deterministic. When the header of a packet reaches an intermediate node, the choice of the next node depends on the appropriateness of the route and on local congestion properties. The R2 design distinguishes three possible route continuations:

- *Best Path* sends the packet to a node which is closer to the destination (there are often two such choices).
- *No Farther* sends the packet to a node without increasing the distance to the destination (again there are often two such choices). The intent of this option is to bypass intervening congestion in order to reduce the latency of the packet.
- *Random* sends the packet to any adjacent node except the one it came from. In severe congestion situations this choice permits the packet to back away and circumvent the congestion.

The output router starts with the natural attempt to send the packet via a *Best Path*. Two conditions need to be met in order to send a packet via port x to a

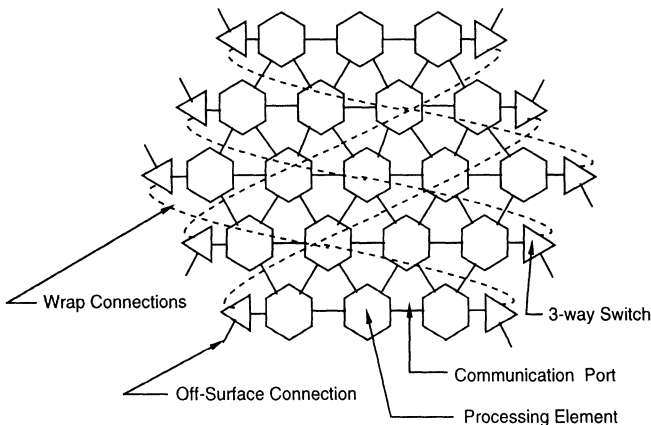


Fig. 4.9. Topology of the interconnect fabric

node y : port x must be free and a buffer must be available in node y . The control protocol on R2's external port interfaces combines these two conditions into a single port-grant signal. If the port is granted then the packet is sent. If the port is not granted, due either to being busy or to failure, a stagnation counter is incremented. The stagnation count is an ageing mechanism that can be used to prioritise packet delivery with the goal of reducing the average and worst case packet latency. The router continues to try for a *Best Path* route until the stagnation count exceeds a threshold parameter *Stag1*, in which case *No Farther* routes also can be used. When the stagnation count exceeds a second threshold *Stag2*, it also becomes possible to use *Random* routes. Under any option, if a *Best Path* port is available it will be taken before any other available but suboptimal candidate.

The R2 design also supports virtual cut-through. This means that the next route decision can be made as soon as the destination address has been received in a buffer. The decoupled input and output of the R2 FIFOs permit the head of the packet to be forwarded concurrently with reception of the packet tail. Once the packet leaves a buffer and correct receipt has been signalled by the receiving node, the buffer and port become free for subsequent transactions.

The performance analysis of the R2 design is based on the following implementation estimates:

- External ports have a bandwidth of 200 MB per second. Since each packet is 144 bytes long, 720 ns are required to transfer the entire packet.
- To receive a packet header and to compute the routing procedure takes 120 ns.
- The output router tries to route the packet every 50 ns.
- Stagnation counts are: Stag1 = 50, Stag2 = 100.

The interconnect performance study presented here uses these parameters, but in the larger context design decisions such as these are tuned using the methodology presented in the next section.

4.4 CPN Model of Interconnect Fabric

The CPN model described in this section represents a typical, basic packet transfer schema operating according to the R2 routing strategy. The model consists of 7 pages:

- 1) Declarations of types, variables, functions, and reference variables (approximately 1500 lines of ML code).
- 2) Initialisation allowing us to run the model with different system parameters.
- 3) Packet generator used to feed the interconnect with packets under different sending frequency distributions (shown in Fig. 4.10).
- 4) Input PE port describing packet movement through the input PE port, including port locking, transfer to a node buffer, routing, and port unlocking (shown in Fig. 4.11).
- 5) Output router (shown in Fig. 4.12).

- 6) Node to node packet movement inside the interconnect (shown in Fig. 4.13).
- 7) Destination node completing the packet transfer to the destination CPU (shown in Fig. 4.14).

Tokens represent packets, requests for resources, control variables for locks over shared resources, etc. As an example, a token of type *PACKET* is a record with five fields: packet identifier, local node address, destination address, port identifier on which the packet was received, and a list of ports and node numbers through which the packet was transferred. Tokens of colour *e* carry no information. They are used for control purposes.

Figure 4.10 shows the *PacketGenerator* page. Locks are modelled in a similar way as in the OLTP model. The fusion place *Lock Table* contains a single token representing a table of all port locks in the interconnect. The fusion guarantees mutually exclusive access to the lock table. Analogously, the fusion place *Buffer Table* contains a single token representing a table of available buffers. The guard of transition *Generate Packet* implies that new packets are only generated when an input PE port and a buffer are available. Each time the transition occurs a new packet appears at place *P1*. The colour of the new token is determined by the code segment. It is of type *REQ_PORT* which is a pair of a *PACKET* and a *PORT*. The time intervals between generation of subsequent packets are regulated by the time expression in the arc returning an *e*-token to place *Next*. The ML function *interarrive()* implements an exponentially distributed inter-arrival time with a per-node average defined by an initialisation parameter.

The place *P1* is an input socket of the substitution transition *Input PE Port* which refers to the page shown in Fig. 4.11. Transition *Lock Input PE Port* checks and locks the corresponding input PE port, and computes a list of the packet's best paths. The variable *lpr* on the output arc is determined by the code segment as the value of the expression $(p, best_ports(p))$. The first component is of type *PACKET*, while the second is of type *PORT list*. Next, transition *Add Packet to Buffer* may occur. It updates the *Buffer Table* by adding the packet with the list of Best paths. Finally, transition *Unlock Input PE Port* removes the port lock.

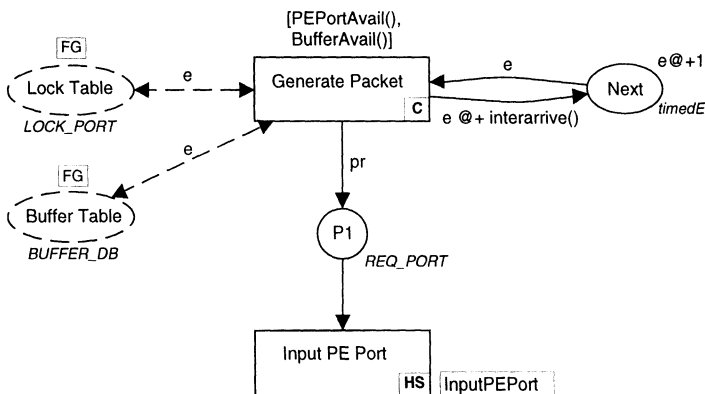


Fig. 4.10. CPN page for *Packet Generator*

The ML functions for the time delays, $pe_del1()$ and $pe_del2()$, use a boolean variable $VCAllowed$ to implement two different strategies: Virtual Cut-Through and Store-and-Forward. If $VCAllowed$ is true, $pe_del1()$ and $pe_del2()$ return the values 120 ns and 600 ns, respectively. This reflects a scenario where the packet can proceed as soon as the header has been received. If $VCAllowed$ is false, $pe_del1()$ and $pe_del2()$ return 720 ns and 0 ns, respectively. This reflects a scenario where the packet cannot proceed until the entire packet is in the buffer.

The *OutputRouter* is shown in Fig. 4.12. Every 50 ns the output router searches the *Buffer Table* to route the packets which are stored in it. The period is determined by the time expression $ms_del()$. Whenever an attempt to send a packet fails, the stagnation count assigned to the corresponding buffer increases by one. When an attempt succeeds, transition *Check Pack & Port* deposits two identical tokens representing the packet into places $P5$ and $P6$. Several packets may be routed concurrently, if port and buffer conflicts do not appear. In this case, multiple tokens are put in $P5$ and $P6$. The tokens in $P5$ and $P6$ have different time delays. The token in $P5$ gets a delay which is 120 ns for $VCAllowed = true$ and 720 ns for $VCAllowed = false$. The token in $P6$ gets a delay which specifies the time required to transfer the entire packet, i.e., 720 ns. Only then is it possible to *Free Buffer*.

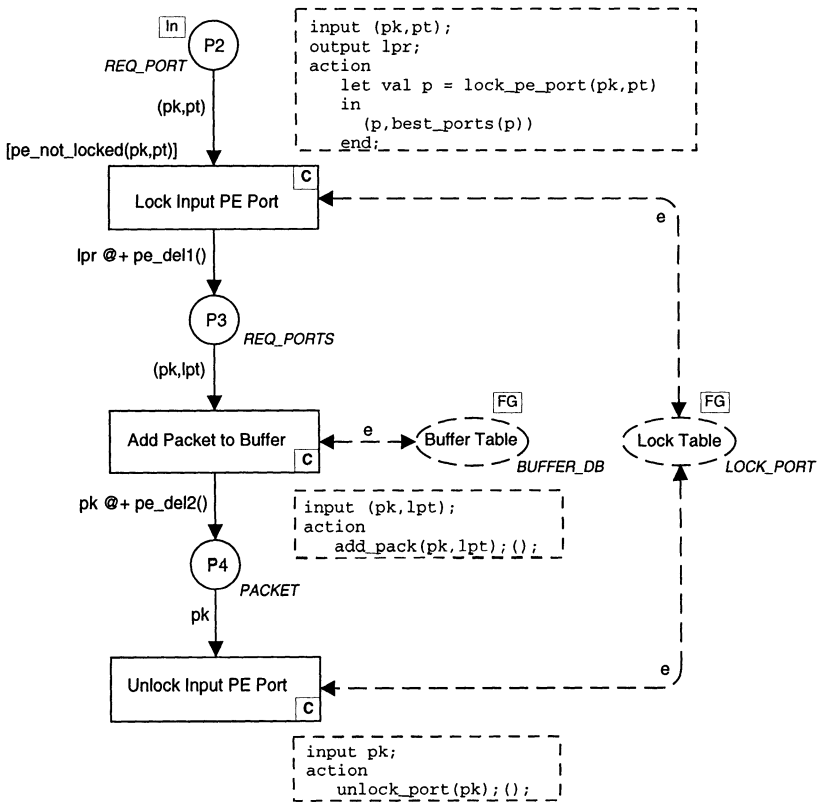


Fig. 4.11. CPN page for *Input PE Port*

The subpages for substitution transitions *Node to Node* and *Destination Node* are shown in Figs. 4.13 and 4.14. The second of these handles packets which reach their final destination node, while the first handles all other packets. The fusion place *Statistics* in Fig. 4.14 collects the packets which have completed their journey. The trace and time information of the packets are used for performance analysis.

The CPN model described above provides a detailed specification of the interconnect. To obtain efficient simulations the CPN model was tuned in a number of ways, which were similar to the transformations made to obtain an efficient simulation model of the OLTP system (see Sect. 4.1). The resulting simulation model is at least five times faster than the specification model and its execution speed is almost independent of the workload rates.

The simulation model presented in this section required a modest amount of programming. However, the simulation model is flexible and easy to modify. The basic features of the interconnect are modelled within the net structure and the net inscriptions, while the routing strategies are specified by means of ML functions. This implies that the same net model can be used to study both the Virtual Cut-Through and the Store-and-Forward strategies.

The goal of the interconnect simulation model was to develop a convenient and flexible model that is easy to modify in order to explore the potential R2 design space. All vital information was written to files, and we developed a few simple tools to analyse these history traces, perform some simple statistical analysis, and illustrate the results graphically.

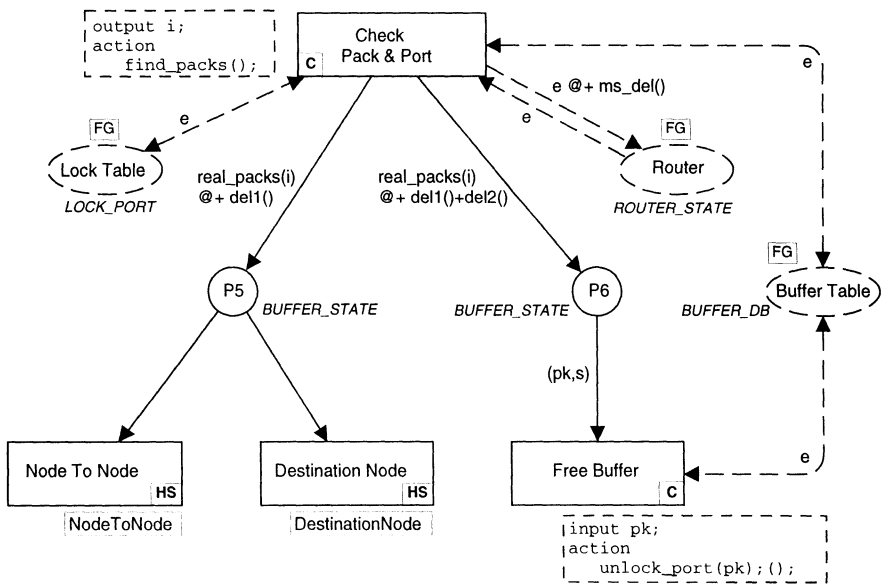


Fig. 4.12. CPN page for Output Router

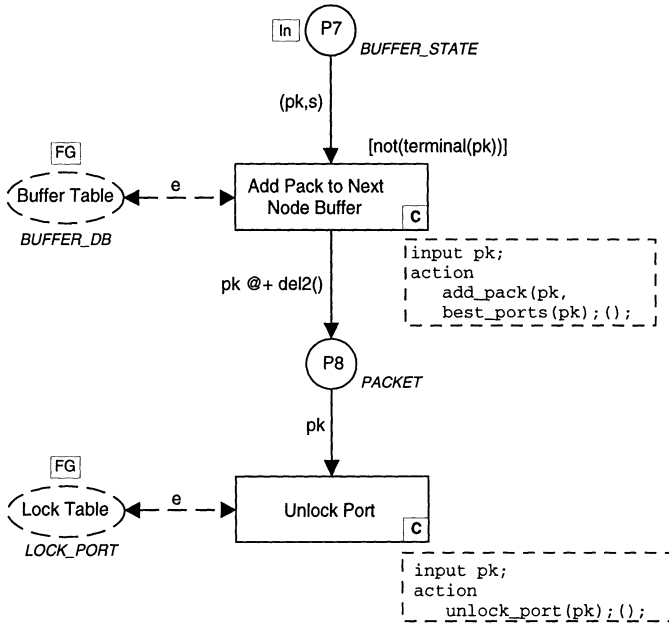


Fig. 4.13. CPN page for Node to Node packets

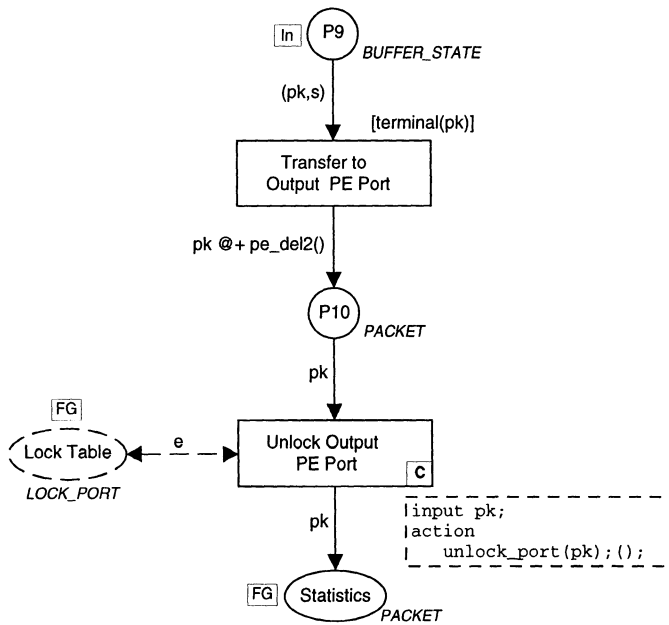


Fig. 4.14. CPN page for Destination Node packets

4.5 Conclusions for Transactions and Interconnect Project

Each of the two projects was accomplished by a team of three persons. Building a CPN model and debugging it took about two months (in both projects). Much time was spent to create appropriate data structures to speed up the simulation (taking into account the industrial size of the modelled systems). We also used significant time to debug the models. This time could be reduced through improved ML debugging facilities. All three participants had a good prior knowledge of Petri nets.

Modelling tools and environments based on Petri nets are very useful at the initial stages of a design process when the most important key hypotheses and decisions about system structure and functioning should be made and checked by rapid prototyping. Net models provide both graphical and mathematical views of system models. As a graphical tool, Petri nets are similar to flowcharts and diagrams (augmented by the token animation representing dynamic activities) and are, thus, familiar to many system practitioners. On the other hand, Petri nets can be used in a variety of analytical forms that can be manipulated in a precise, mathematical way.

We have presented our experiences in modelling an on-line transaction processing system and a scalable interconnect fabric using CP-nets and the CPN tools. We have performed industrial-sized simulations of tens of thousands of transactions running on databases with millions of records. Our success in this modelling task shows that Petri nets and currently available Petri-net tools provide a viable modelling environment for the performance analysis of large systems. However, the use of nets in the modelling was not straightforward. CP-nets were combined with ML code (when it did not harm the model clarity and preserved the basic net modelling paradigm) to improve upon a more direct but time-consuming imitation of system activities. The design of some components of the net model evolved from a specification model to a more efficient simulation model. This strategy has allowed us to understand the real power and limits of net modelling.

CP-nets and the CPN tools have proven to constitute a useful and effective platform for the analysis of complex concurrent system designs. The R2 interconnection scheme was presented as a case study example of the method. CPN models permit a wide variety of system detail to be analysed and incrementally extended. The use of Standard ML to specify net inscriptions (such as arc expressions, guards, colour sets, and initial markings) makes the model modular and easy to modify. It is easy to test different scenarios, e.g., to increase the number of active processes and the number of transactions to be performed. The CPN simulator permits detailed performance studies to be conducted. Care must be taken to divide the CPN model into submodels with well-defined interfaces. Further care must be taken in the choice of using net structure and net inscriptions versus ML functions. Net structure and net inscriptions are typically better for specification purposes since they are more intuitive and more clearly represent the structure of the concurrent system. However, ML code is in some cases more efficient as the basis for a simulation model. Both are important.

In general, our experience using CP-nets and the CPN tools showed that the tools may serve as a basis for further development of sophisticated environments adequate for the modelling of complex concurrent/distributed systems. Such an environment should include some set of methodology guidelines as well as a library of utilities, supporting statistical analysis of traces, model debugging, animation, etc.

Chapter 5

Mutual Exclusion Algorithm

This chapter describes a project accomplished by *Jens B. Jørgensen and Lars M. Kristensen, Aarhus University, Denmark*. The chapter is based upon the material presented in [34]. The project was conducted in 1996.

We describe how occurrence graphs of CP-nets were used to verify the correctness of a mutual exclusion algorithm designed by Lamport. It is proved that the algorithm has all the properties to be expected from a mutual exclusion algorithm, e.g., that there are no deadlocks and that at any time no more than one process is in the critical section. The construction and analysis of occurrence graphs are done by the CPN tools totally automatically. This means that occurrence graphs are easy to use, requiring very limited human work. Hence, this kind of verification is cheap and reliable.

It is also demonstrated how permutations of colour set values can be used to obtain a significant reduction of the size of the occurrence graph without losing analytic power. By exploiting the inherent symmetries of the system two equivalence relations are defined – one for markings and one for binding elements. The basic idea is to obtain a condensed occurrence graph, called an OS-graph, by lumping together symmetric markings and symmetric binding elements. Each node in the OS-graph represents an equivalence class of reachable markings (which can be mapped into each other by means of permutations). Analogously, each arc represents an equivalence class of binding elements.

Lamport's algorithm has also been studied by means of Petri nets in [4]. There the verification was done by means of place invariants while stochastic Petri nets were used to investigate its performance.

Section 5.1 contains an introduction to Lamport's mutual exclusion algorithm. Section 5.2 presents the CPN model of the mutual exclusion algorithm. Section 5.3 discusses how the CPN model was verified by means of occurrence graphs and occurrence graphs with permutation symmetries. Finally, Sect. 5.4 presents a number of findings and conclusions for the project.

5.1 Introduction to Mutual Exclusion Algorithm

Lamport's algorithm for mutual exclusion is specified in [36]. It is designed for a shared-memory multiprocessor architecture. Each of the N processes has a unique identifier in the interval $1..N$. Figure 5.1 shows the code executed by process i when the process is about to enter the critical section (lines 1–19) and leave it (lines 23–24).

The algorithm uses three global variables: x and y , which are integers, and an array $b[1..N]$ of booleans. The await-statements in lines 6, 13, and 16 represent a busy wait and can be seen as a shorthand for “while \neg cond do skip”. Angle brackets $\langle \dots \rangle$ are used to enclose atomic statements (i.e., reads/writes of x , y , and elements of b). A much more detailed explanation of the algorithm can be found in [36]. Here, we only show that the algorithm works by proving that it possesses a number of properties which mutual exclusion algorithms are expected to have, e.g., that there are no deadlocks and that at any time no more than one process is in the critical section.

```

1  start:
2  < b[i] := true >;
3  < x := i >;
4  if < y ≠ 0 > then
5      < b[i] := false >;
6      await < y = 0 >;
7      goto start;
8  fi;
9  < y := i >;
10 if < x ≠ i > then
11     < b[i] := false >;
12     for j := 1 to N
13         do await < ¬b[j] > od;
14
15     if < y ≠ i > then
16         await < y = 0 >;
17         goto start;
18     fi;
19 fi;
20
21 critical section;
22
23 < y := 0 >;
24 < b[i] := false >;

```

Fig. 5.1. Lamport's algorithm for mutual exclusion of shared-memory multiprocessors (taken from [36])

5.2 CPN Model of Mutual Exclusion Algorithm

In this chapter we show how Lamport’s algorithm can be modelled by means of CP-nets. This is actually quite straightforward.

The control flow of the N processes is represented by a state machine, i.e., a subnet where each occurring binding element removes exactly one token and adds exactly one token (cf. Def. 4.5 (iv) in Vol. 1). Each place in the state machine has a colour set which contains all the possible process identifiers:

$$PID = \{1,2,\dots,N\}.$$

Each of the three global variables is represented by its own place. The simple variables x and y can take the values $0..N$. Hence they are represented by two places x and y with the colour set:

$$PID0 = \{0,1,2,\dots,N\}.$$

Initially the values of x and y are set to zero. At any time each of these places has exactly one token, denoting the present value of the variable. The boolean array b is represented by a place with the colour set:

$$PID \times BOOL.$$

At any time this place contains exactly N tokens with colours:

$$(1,b_1), (2,b_2), \dots, (i,b_i), \dots, (N,b_N),$$

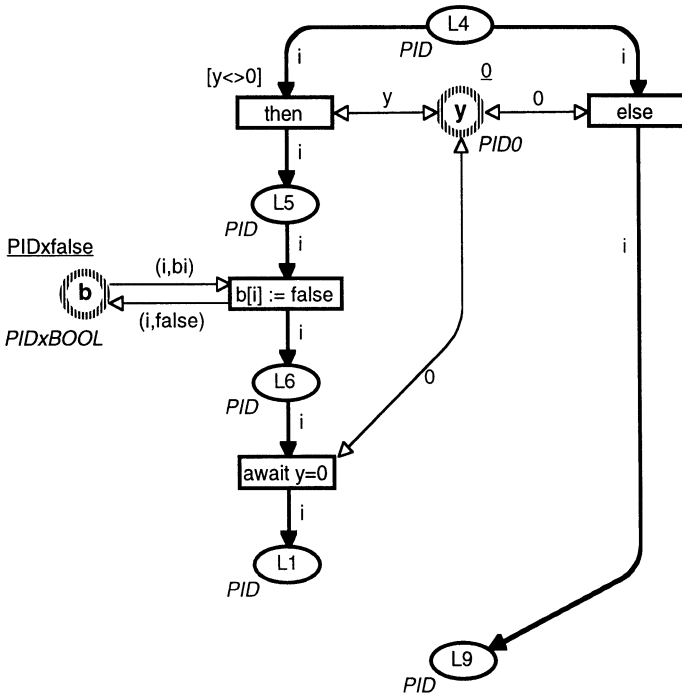


Fig. 5.2. CPN representation of the statements in lines 4–8

where b_i is the present value of the i -th array element. Initially all array elements are set to false.

Next, we discuss how the different statements can be modelled. To illustrate this, we consider the statements in lines 4–8. They are represented by the subnet in Fig. 5.2. When a token is present at place $L4$ the corresponding process is ready to start the execution of the statement in line 4 of the algorithm. If $y = 0$, the upper right transition occurs, and since the else-part of the if-statement is missing, we immediately reach line 9. If $y \neq 0$, the upper left transition occurs, followed by the two transitions below. They model the assignment-statement and the await-statement, respectively. Finally we reach line 1, due to the goto-statement. We could have modelled the goto-statement by a separate transition, but we have chosen not to do this, to keep the occurrence graph as small as possible.

As a second example, let us consider the statements in lines 12–13. They are represented by the subnet in Fig. 5.3. Transition *for begin* prepares the execution of the for-statement. It adds N tokens:

$$PID \times \{i\} = \{(j,i) \mid j \in PID\}$$

to place *ready*. Each of these tokens represents an execution of the body of the for-loop. The first element in the colour denotes the value of the loop variable j , while the second element denotes the executing process i . It is necessary to include the second element to prevent the tokens being mixed up with tokens from other processes simultaneously executing the for-statement. Transition *await not b[j]* represents the statement in the for-loop. It is executed N times, one for each value of the loop variable j . Finally, the transition *for end* finishes the execution of the for-statement. It occurs when all j values have been processed by the for-loop (i.e., when all the tokens at *ready* have been moved to *done*).

Note that we have modelled the for-statement in a very general way, where the N different values for the loop variable j are processed in an arbitrary order. This generalisation may seem strange – in particular because it significantly increases the number of reachable states. However, the generalisation is necessary,

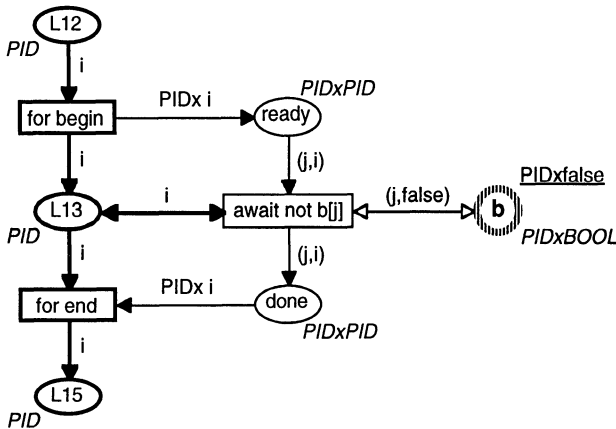


Fig. 5.3. CPN representation of the statements in lines 12–13

because we want to use permutation symmetries to reduce the occurrence graph. If the values of j are processed in a particular order, the system treats the processes in a non-symmetric way and this means that the CPN model does not fulfil the consistency properties which are necessary for using OS-graphs (see Def. 3.16 of Vol. 2).

In Figs. 5.2 and 5.3 we have shown how to represent some of the statements in Lamport's algorithm. The remaining statements can be represented in a similar way and this yields the CPN model in Fig. 5.4. Place $L21$ represents the critical section. To improve readability we have highlighted the control structure of the algorithm by drawing the state machine part of the CP-net with thicker lines. In this part we have omitted the arc expressions (which all are i) and the

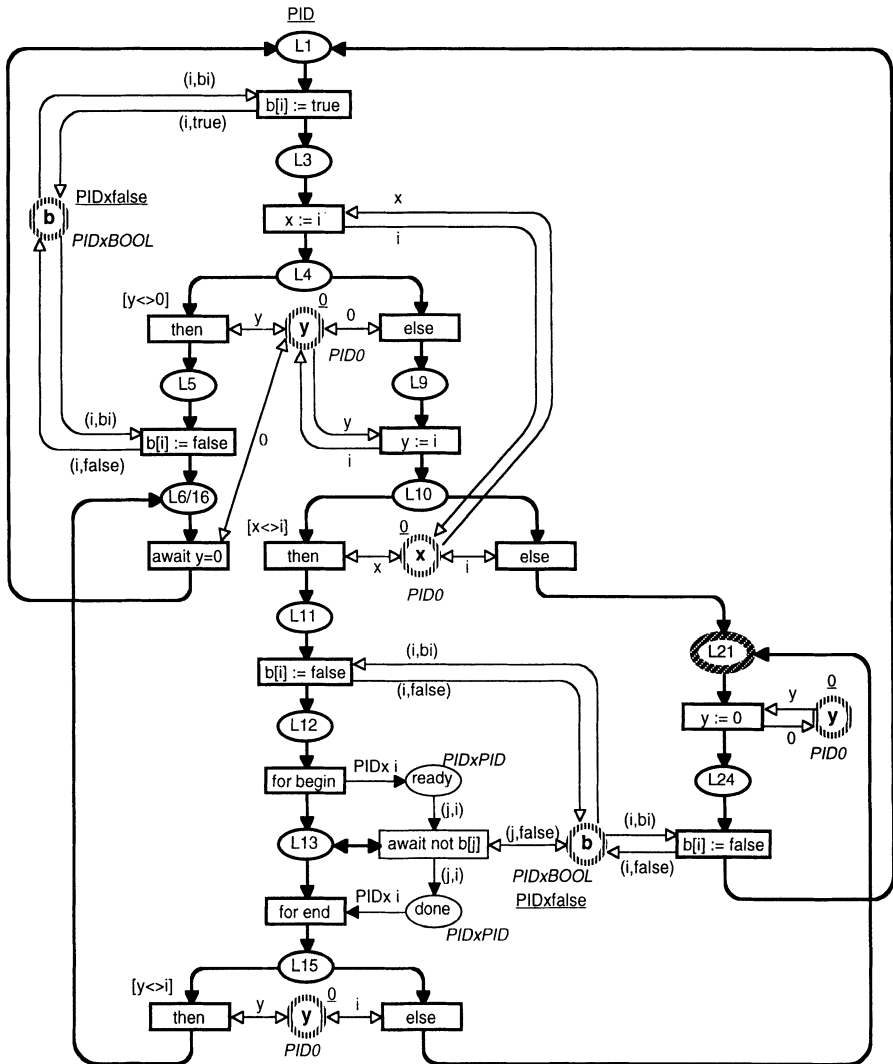


Fig. 5.4. CPN model of Lamport's algorithm

colour sets (which all are PID). The places representing the variables y and b have been drawn several times (using two fusion sets).

The declarations for the CPN model are shown in Fig. 5.5. Note that we define the colour set PID to be a subset of the colour set $PID0$, specifying that PID contains all non-zero elements of $PID0$. In this way we guarantee that the symmetries permute the elements of PID in the same way as the elements of $PID0$.

```

1  val N = 3; (* number of processes *)
2  fun nonzero i = (i <> 0);
3  color BOOL = bool;
4  color PID0 = int with 0..N declare ms;
5  color PID = subset PID0 by nonzero declare ms;
6  color PIDxPID = product PID * PID declare mult;
7  color PIDxBOOL = product PID * BOOL declare mult;
8  var i,j: PID;
9  var x,y: PID0;
10 var bi: BOOL;
11 val PIDxFALSE = mult'PIDxBOOL(PID,1`false);
12 fun PIDxi = mult'PIDxPID(PID,1`i);

```

Fig. 5.5. Declarations for the CPN model of Lamport's algorithm

5.3 Occurrence Graph Analysis of Mutual Exclusion Algorithm

To verify the correctness of Lamport's algorithm we consider the following properties:

- **No deadlocks.** No execution can lead to a situation in which all processes are blocked. To prove this property, we show that the CPN model has no dead markings.
- **Mutual exclusion.** At any time no more than one process is in the critical section. To prove this property, we show that place $L21$ has 1 as upper integer bound.
- **Possible to enter.** When several processes attempt to enter the critical section, eventually one will do so. To prove this property, we show that transition $y:=0$ (immediately below $L21$) is impartial. This implies that the critical section is left an infinite number of times (in each infinite occurrence sequence) and hence it is also entered an infinite number of times.
- **Possible to return.** In any execution it is always possible to return to a marking in which all processes are positioned at *start*, all entries in the b -array are false, and y is 0. To prove this property, we show that the markings described above constitute a home space.

- **No dead code.** Each statement always has the possibility of being executed – by some process sometime in the future. To prove this property, we show that each transition is live.
- **Independence.** No process is ever forced to start the entering of the critical section in order to prevent a deadlock. To prove this property, we show that when a marking has transition $b[i] := true$ (immediately below $L1$) as the only enabled transition, then all processes are in $L1$.

To investigate the six properties, we constructed occurrence graphs for $N = 2$ and $N = 3$. The first O-graph had less than 400 nodes while the latter had nearly 20 000 nodes. For these two O-graphs it was straightforward to verify the first five properties. This was done by using a set of standard queries that implement the O-graph proof rules from Sect. 1.4 of Vol. 2. The queries can be used without knowing the details of the proof rules (or the proof of their correctness). The user simply invokes the appropriate query function and gets back a result. As an example, the mutual exclusion property is verified by invoking a function *UpperInteger* with place $L21$ as argument, obtaining the result 1. For the independence property we used a search function to locate those nodes for which all outgoing arcs represent transition $b[i] := true$. For these nodes the search function checks that all processes are in $L1$. For more details about search functions, see the description of *SearchNodes* in Sect. 1.7 of Vol. 2. The verification of the six properties is done in a few minutes. Sometimes it takes more time to write down the name and the arguments of the query function than to execute it.

Having finished the investigation of the O-graphs for $N = 2$ and $N = 3$, we wanted to consider larger values of N . Unfortunately, this was not possible – at least not by using full occurrence graphs. The O-graph for $N = 4$ is so large that it could not be handled by the machine and tool support we were using.

Hence, we turned to occurrence graphs with symmetries, OS-graphs, defined in Chap. 3 of Vol. 2. The basic idea behind the use of OS-graphs is the observation that Lamport's algorithm treats all processes in the same way. Below we show two markings M_1 and M_2 that are nearly identical. The only difference is that in M_1 it is process 1 which has executed the assignment $b[i] := true$, while in M_2 it is process 2:

$$\begin{array}{ll}
 M_1(L1) = 1^2 + 1^3 & M_2(L1) = 1^1 + 1^3 \\
 M_1(L3) = 1^1 & M_2(L3) = 1^2 \\
 M_1(b) = 1^{\langle 1, true \rangle} + & M_2(b) = 1^{\langle 1, false \rangle} + \\
 \quad 1^{\langle 2, false \rangle} + & \quad 1^{\langle 2, true \rangle} + \\
 \quad 1^{\langle 3, false \rangle} & \quad 1^{\langle 3, false \rangle} \\
 M_1(x) = 1^0 & M_2(x) = 1^0 \\
 M_1(y) = 1^0 & M_2(y) = 1^0.
 \end{array}$$

It is easy to see that M_1 can be mapped into M_2 by means of the permutation (1 2) which maps 1 into 2, 2 into 1, and 3 into 3. Intuitively, the two markings are equivalent/symmetrical. If we know what can happen from one of them, we

also know what can happen from the other. As an example, each of the two markings has exactly three enabled binding elements:

For M_1 we have:	For M_2 we have:
$(x := i, \langle i = 1, x = 0 \rangle)$	$(x := i, \langle i = 2, x = 0 \rangle)$
$(b[i] := \text{true}, \langle i = 2, b_i = \text{false} \rangle)$	$(b[i] := \text{true}, \langle i = 1, b_i = \text{false} \rangle)$
$(b[i] := \text{true}, \langle i = 3, b_i = \text{false} \rangle)$	$(b[i] := \text{true}, \langle i = 3, b_i = \text{false} \rangle)$

The permutation (1 2) maps each of the three binding elements enabled in M_1 into one of those enabled in M_2 . The two sets of binding elements lead to two sets of direct successors, such that permutation (1 2) maps each direct successor of M_1 into a direct successor of M_2 . Hence, we have illustrated that symmetric markings behave in a symmetric way. They have symmetric sets of enabled binding elements and symmetric sets of direct successors. Using induction, this property can be expanded to finite and infinite occurrence sequences.

The CPN model of Lamport’s algorithm contains many markings that are symmetric to each other, in the sense described above. The basic idea in OS-graphs is to obtain a condensed occurrence graph by lumping together symmetric markings and symmetric binding elements. Each node in the OS-graph represents an equivalence class of reachable markings (which can be mapped into each other by means of permutations). Analogously, each arc represents an equivalence class of binding elements.

Now let us be more precise. First, we specify the exact set of symmetries that we allow. To do this, we make a permutation symmetry specification, as defined in Def. 3.11 of Vol. 2. Our model has two atomic colour sets. For *PIDO* we allow all those permutations that map 0 into itself. This indicates that all the N processes can be replaced by each other, while 0 is a special value. For *BOOL* we only allow the identity function, which maps false into false and true into true. Permutations of the structured colour sets *PID*, *PIDxPID*, and *PIDxBOOL* are derived from a permutation of *PIDO* in a straightforward way. As an example, the permutation (1 2) maps (1,3) into (2,3) and maps (2,true) into (1,true).

Next, we prove that our symmetry specification is consistent with the behaviour of the CPN model, i.e., that symmetric markings and binding elements really are treated in a symmetric way. This means that we need to prove the consistency properties defined in Def. 3.16 of Vol. 2. However, this is straightfor-

N	Nodes			Arcs			N!
	O-graph	OS-graph	Ratio	O-graph	OS-graph	Ratio	
2	380	191	1.99	716	358	2.00	2
3	19 742	3 367	5.86	58 272	9 788	5.95	6
4	1 914 784	83 235	23.00	9 046 048	383 030	23.62	24

Fig. 5.6. Size of O-graphs and OS-graphs for Lamport’s algorithm

ward, since all arc expressions, guards, and initialisation expressions are very simple.

Finally, we must provide the two ML functions, *EquivMark* and *EquivBE*, described in Sect. 2.7 of Vol. 2. The first function takes two markings and decides whether they are equivalent or not. The second function does the same, but for two binding elements. A straightforward way to implement the two ML functions is to test all permutation symmetries in turn. If one of the permutation symmetries maps the first argument of the ML function into the second argument, true is returned; otherwise false. A more efficient way to implement the two ML functions is to use restrictions sets as described in Sect. 3.5 of Vol. 2. For this purpose a library of useful ML functions exists.

Now we can construct the OS-graphs for $N = 2$, $N = 3$, and $N = 4$. They have the sizes shown in Fig. 5.6. As can be seen, the OS-graphs are much smaller than the corresponding O-graphs. $N!$ is the number of permutations that we allow. This means that each equivalence class can have at most $N!$ members, and hence we cannot obtain a reduction ratio larger than $N!$. The table shows that we are surprisingly close to this theoretical limit.

The size of the O-graph for $N = 4$ is calculated from the corresponding OS-graph (since the O-graph is too big to be constructed by the tool). This calculation is done automatically by the OS-graph tool. The only thing that the user has to do is to provide a function *SelfSym*, which calculates the set of self-symmetries for a given marking. The self-symmetries are those symmetries that map the marking into itself. For more information on self-symmetries, see Sects. 3.4–3.5 in Vol. 2.

In Sect. 5.2, we explained that the use of OS-graphs forces us to use a more general for-statement, increasing the number of reachable states. This means that it is interesting to compare the size of the OS-graph (with the generalised for-statement) with the size of the O-graph (with the ordinary for-statement). For $N = 3$, the latter has 11 978 nodes and 32 226 arcs. This means that it is more than three times as big as the OS-graph. Hence, we have gained more by OS-graphs than we have lost due to the more general for-statement.

The time used to calculate the different occurrence graphs are shown in Fig. 5.7. All calculations were performed on a Sun Sparc 20 with 256 MB physical RAM. Note that the OS-graph for $N = 3$ is calculated 27 times faster than the corresponding O-graph, although the OS-graph is only 6 times smaller than the O-graph. This means that the average processing time of an OS-graph node is

N	O-graph	OS-graph	Ratio
2	5 seconds	4 seconds	1.25
3	37.7 minutes	1.4 minutes	26.89
4	—	291.2 minutes	—

Fig. 5.7. Time to construct O-graphs and OS-graphs for Lamport's algorithm

significantly smaller than the processing time of an O-graph node. At first glance, this is rather surprising – obviously it should be more expensive to test whether the markings of two nodes are equivalent than to test whether they are identical. However, the reason is simple. The OS-graph contains many fewer nodes than the O-graph, and thus a newly constructed OS-graph node has to be compared, on average, to many fewer nodes than a newly constructed O-graph node.

When the OS-graphs had been constructed they were investigated. This was done in exactly the same way as for O-graphs. Again we used a set of standard queries – this time implementing the OS-graph proof rules from Sects. 3.2 and 3.3 of Vol. 2.

A later version of the OS-graph tool will include a check of the consistency properties, and it will automatically derive the ML functions *EquivMark* and *EquivBE* from the permutation symmetry specification. This will make the OS-graph construction fully automatic and remove the possibility of making errors in the consistency proof or in the implementation of the two ML functions. Meanwhile, we can use the OS-graph tool to calculate the size of some of the small O-graphs (e.g., for $N = 2$ and $N = 3$). Then we can compare these computed sizes with the sizes of the actual O-graphs (constructed by the O-graph tool). If the sizes match, we have a strong indication that our symmetry specification is consistent with the CPN model and that our two ML functions are correctly implemented.

To illustrate why the OS-graphs are smaller and faster to compute than the O-graphs, let us consider Figs. 5.8 and 5.9, which show the first few nodes in the O-graph and the OS-graph, respectively (for $N = 3$). The two drawings can be

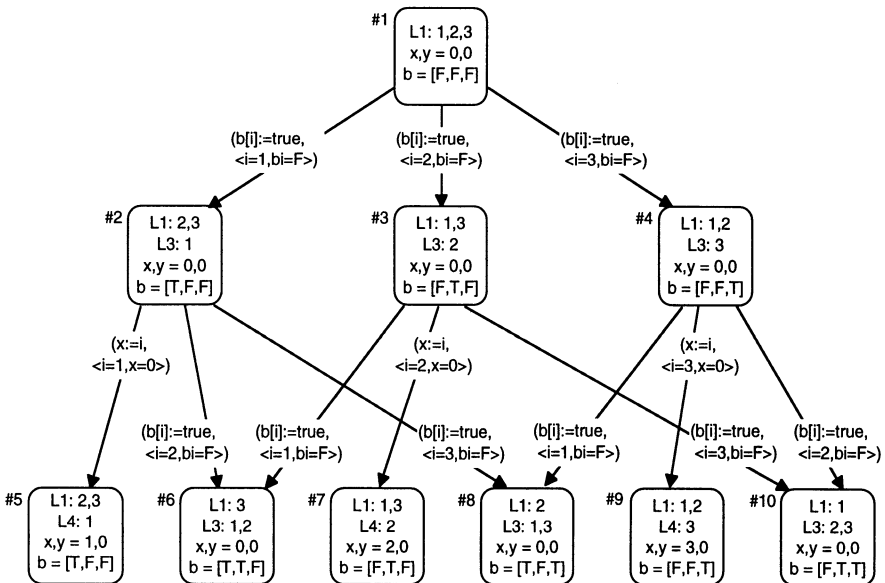


Fig. 5.8. The first few nodes in the O-graph (for $N = 3$)

produced by means of the O-graph tool and the OS-graph tool. The layouts have been manually improved.

Node #S1 in the OS-graph has the same marking as node #1 in the O-graph. Both nodes represent the initial marking of our CPN model. Node #S2 in the OS-graph has the same marking as node #2 in the O-graph. However, it represents not only this marking but also the markings of nodes #3 and #4 in the O-graph. Analogously, node #S3 represents the markings of nodes #5, #7, and #9, while node #S4 represents the markings of nodes #6, #8, and #10.

We would also like, of course, to be able to investigate Lamport's algorithm for $N > 4$. However, this is not possible, because also the OS-graph becomes too big to be handled by our present tool/machines. Hence, we have to simplify our model in some way. One possibility is to represent the for-statement in lines 12–13 in a less expensive way with respect to state explosion. In [4] the for-statement is represented by a single transition which simultaneously tests the values of all N elements in the array b . This solution is symmetric and it significantly decreases the number of reachable states. However, it is not a fully correct representation of Lamport's algorithm, because it collapses a sequence of independent tests into a single action. Hence, we cannot be totally sure that correctness of the algorithm, with the simple representation of the for-statement, implies correctness when the ordinary or general for-statement is used. However, it is still worth investigating the simplified model, because it is possible to deal with higher values of N . If the simplified model allows us to locate an error, then it is likely that this error also exists in the non-simplified model. With the simplified model we were able to handle $N = 5$ and $N = 6$. For the latter, the OS-graph has 83 895 nodes and 360 933 arcs. The O-graph has 34 258 216 nodes and 175 300 026 arcs.

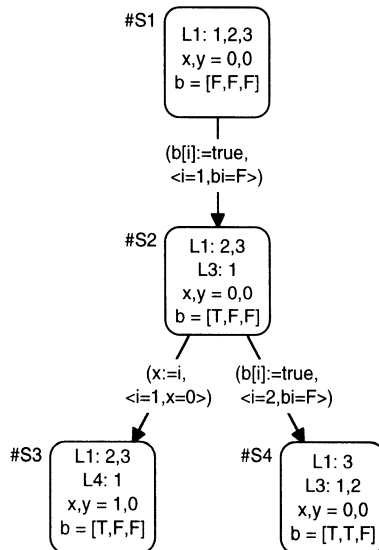


Fig. 5.9. The first few nodes in the OS-graph (for $N = 3$)

5.4 Conclusions for Mutual Exclusion Algorithm Project

In this chapter we have shown that it sometimes is possible to alleviate the state explosion problem by exploiting symmetries inherent to the system which we model. By using OS-graphs we were able to verify Lamport's mutual exclusion algorithm for four processes, instead of the three processes that could be handled by O-graphs. Although this result may not seem too impressive, it is definitely a step in the right direction, and it indicates that symmetries are useful. If they are combined with other occurrence graph reduction methods, e.g., stubborn sets, the results may become more convincing, significantly moving the border line between those models that are manageable for occurrence graph analysis and those models that are too complex.

The construction and analysis of OS-graphs are fully automatic. This means that occurrence graph analysis is cheap and reliable. For OS-graphs this only becomes totally true when the tool includes a check of the consistency properties and is able to derive *EquivMark* and *EquivBE* from the permutation symmetry specification.

In [4] Lamport's algorithm is verified by means of place invariants. This has the advantage that the verification is independent of the number of processes, and hence valid for all N . However, free rides are rare. The disadvantage of the place invariant method is the fact that the verification becomes far less automatic. The verification in [4] relies on a number of quite complex and lengthy mathematical arguments, which are prone to error and time-consuming. The same is true for the justification provided in Lamport's original paper [36].

Chapter 6

ISDN Supplementary Services

This chapter describes a project accomplished by *Greg Findlow and Geoff Gerard, Telstra Research Laboratories, Clayton, Victoria, Australia, with supervision and input from Jonathan Billington and Richard Fone*. The chapter is based upon material from the internal report [23] and the paper [24]. The project was conducted in 1991–92.

We describe how CP-nets and the CPN tools were used to model four different ISDN supplementary services. The Integrated Services Digital Network (ISDN) is a fast telecommunications network which customers access via a set of 64 kbps traffic channels. It supports a wide range of basic teleservices, such as telephony, facsimile, and data transmission. To enhance the basic services, a set of supplementary services may be offered.

In our project we considered the supplementary services known as Call Forwarding, Call Completion to Busy Subscriber, Closed User Group, and Call Hold. The creation and validation of a CPN model for the ISDN supplementary services turned out to be very helpful to investigate the potential interactions between the services. It also helped us to identify many ambiguities, omissions, and other shortcomings in the service recommendations.

The project was accomplished by two persons over a period of one year. We used a total of nine man-months, including initial training and writing of reports. The CPN model was validated by means of simulation and occurrence graph analysis. Due to a tight project schedule and rather limited resources, the validation did not become as thorough and detailed as desired. However, we did some limited amount of work and identified a number of useful techniques, which may be of interest to other projects.

Section 6.1 contains an introduction to the supplementary ISDN services. We also describe a number of decisions and assumptions made during the modelling process. Section 6.2 describes the CPN model of the supplementary services. Section 6.3 discusses how the CPN model can be validated by means of simulation and occurrence graph analysis. Finally, Sect. 6.4 presents a number of findings and conclusions for the project.

6.1 Introduction to ISDN Supplementary Services

Call Forwarding (CF) allows a call to be forwarded, i.e., redirected from one phone number to another. There are three kinds of forwarding. One of these is unconditional (CFU), while the other two only happen when the original destination is busy (CFB) or provides no reply (CFNR). All three kinds of forwarding may be active simultaneously, and they may specify different forwarding destinations.

Call Completion to Busy Subscriber (CCBS) allows a caller to temporarily postpone his call, in the case where the recipient is busy with another call. When the recipient becomes idle, the caller is notified via a **CCBS recall**. If the caller also is idle, the call is established as a **CCBS call**. Otherwise, the CCBS request is suspended until the caller becomes free again. It is only possible to request a CCBS for a call which has not been forwarded (or has been CFB forwarded at its original destination and eventually met a busy recipient, where it was not forwarded). In both cases the CCBS is towards the original destination.

Closed User Group (CUG) allows the users to be members of one or more user groups. Members of a group can communicate among themselves. Via **CUG restrictions** it may be specified that some group members cannot receive calls from other group members and/or make calls to these. It is also specified whether a user is allowed to receive calls from users outside his groups and/or make calls to these. When a CUG subscriber makes a **CUG call**, he specifies one of his groups (or uses a default group). To make sure that the above CUG restrictions are fulfilled each call must pass an outgoing CUG check (at the originating exchange) and an incoming CUG check (at the destination exchange). If the call is forwarded it will have to pass additional CUG checks.

Call Hold (CH) allows a user to interrupt an existing call (and free the channel that it is using) without clearing the call. This means that the interrupted call subsequently may be re-established, when and if desired.

The International Telecommunications Union (ITU) uses a three-stage methodology for defining ISDN services. The first stage describes the services from a user's perspective. Many service interactions are visible at this level. For example, when Alice invokes CCBS against Bob, and Bob subsequently activates CFU to Carol, should Alice then be offered a CCBS recall when Bob becomes free? The construction of a Stage 1 CPN model may help to identify many interactions between services and find ambiguities in the service descriptions. Moreover, the simulation and analysis of such a model may reveal a number of less obvious interactions between the services.

The second stage in the ITU methodology identifies the functional capabilities and information flows which the network must support to provide the services, while the third stage defines the detailed signalling protocols providing the capabilities and flows.

A Stage 3 CPN model may be used to verify the signalling systems, e.g., to check that they provide the intended functionality and information flows and cannot reach undesirable situations such as deadlocks. A Stage 2 or 3 CPN model

may also help to identify service interactions which are not visible from the user's viewpoint. For example, when Alice accepts a CCBS recall, to set up a CCBS call from her to Bob, the network sends out a new call request on Alice's behalf. From where does the network obtain the CUG information for this new request?

We decided to base our CPN modelling on the Stage 1 recommendations, although we also used the Stage 2 and 3 recommendations for clarification of some points. The main reason for this choice was to avoid a potential state explosion. It was felt that (at least initially) it was better not to model the explicit messages in Stage 2 and 3, since they would increase the size of the occurrence graph (making analysis of the model more difficult), but would probably not reveal a lot of extra interactions. Time constraints also influenced our choice of modelling level. With more time, we would have attempted also to construct and analyse some Stage 2 or 3 models.

To concentrate on the various states a call request goes through, we omitted details such as the geographical locations of exchanges from our model. With the network shown in Fig. 6.1, a call from Alice to Bob may pass through the Melbourne city exchange at the same time as a call from Carol to Daniel is leaving that exchange. However, the fact that the calls pass through the same physical exchange is irrelevant, since they remain logically separated from each other. Hence it is sufficient to record that Carol's call request is leaving its originating exchange while Alice's call request is in transit (passing through an intermediate exchange).

In our model a call request is represented by a token, which can be at three different CPN places. The first place represents the situation where the request is at the originating exchange, awaiting the results of an outgoing CUG check. At the second place the request is in transit between the originating and the terminating exchanges, or it is at the terminating exchange awaiting the results of an incoming CUG check. At the third place the request is ready to be offered to the called party. As a call request "moves" forward through a network, messages are sent in both directions. Hence, the position of a call request is an abstract concept, which (roughly speaking) corresponds to the furthest point reached by any message associated with the call.

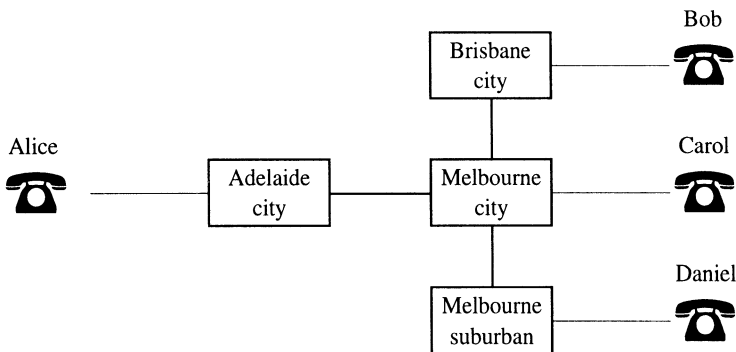


Fig. 6.1. A possible telecommunications network

Although the supplementary services are defined specifically for ISDN networks, they could also be provided in other kinds of networks. This would probably not change the majority of interactions, since they are determined by the (possibly conflicting) results the various services aim to achieve, rather than by the particular characteristics of the network in which the services are provided. For this reason, we abstracted a number of the ISDN characteristics out of our model. Below we discuss some of these simplifications.

A user has two traffic channels (which may each carry a single outgoing/incoming call) and a signalling channel (for establishing and terminating calls). We only modelled the traffic channels, and we did not consider primary rate ISDN (in which a user has 30 traffic channels). Increasing the number of channels per user is unlikely to significantly increase the possible service interactions. One might then ask why we modelled more than one channel per user. The answer is that the existence of multiple channels is one of the main differences between ISDN and standard telephone networks. However, restricting the number of traffic channels to one (and avoiding any ISDN specifics) would yield a model which could be useful for the design and implementation of simpler networks – since it would be possible to reuse work and results from our project.

We only modelled one basic kind of teleservice. Modelling different types (such as telephony, facsimile, and data transmission) is unlikely to contribute significantly to the service interactions – most, if not all interactions, at the Stage 1 level, should be able to occur anyhow.

A user can be busy because he has no free traffic channels. However, he can also be busy because the necessary equipment to handle a call is unavailable. This may happen, e.g., when two fax calls simultaneously reach a user with only one fax machine. Since we did not model different types of teleservices, we simply consider a user to be busy iff both traffic channels are in use.

We represented each user by a token describing the details of his subscription (e.g., specifying the groups to which he belongs and the permissions which he has with respect to these groups). Alternatively, we could have made a model where all checks were resolved non-deterministically (yielding a *pass* or *fail*). This would probably have decreased the size of the occurrence graph. However, it might also have hidden many potential interactions arising from particular subscriptions chosen by the users. For example, suppose that the CUG subscriptions allow Alice to call Bob or Carol, but forbid Bob from calling Carol. Can Bob still demand that his calls are forwarded to Carol? If so, should he be charged for the second leg of such a call?

The rest of this section lists a number of assumptions we made for the four supplementary services. Some of the assumptions are simplifications made to make analysis more tractable. Others were necessary because the ITU recommendations were unclear or incomplete. The number of assumptions reflects the considerable effort involved in creating a formal model from (mainly) textual specifications. The list can be skipped by readers who are not interested in the details of telecommunication systems. X and Y refer to the users involved in a CCBS. X has called Y, found him busy, and hence made a CCBS request.

- Crank-back of calls is not modelled (they allow a call that cannot be forwarded to be reoffered to the last destination at which a CFNR occurred).
- All users are modelled as CCBS subscribers and a CCBS must be invoked whenever possible. This removes innocuous sequences of events in which service interactions do not occur because CCBS is not used.
- A CCBS request includes the CUG information from the initial call request. This is necessary to allow the network to set up the CCBS call without requesting the information once more from X. The ITU version does not mention this dependency between CUG and CCBS. The problem was discovered during our modelling.
- Each user is only allowed to have one outstanding CCBS request made by him and one outstanding CCBS request towards him – at a time.
- A CCBS request is removed from the relevant register when the corresponding CCBS recall is accepted by X (which means that CCBS calls behaves in exactly the same way as ordinary calls). In the ITU version a CCBS request is intact until the corresponding CCBS call reaches Y (which means that CCBS calls must be distinguishable from ordinary calls).
- If a call request is cancelled while it is being serviced, then the CCBS recall is terminated in a reasonable way. The ITU specification does not cover this event – a deficiency in the specification discovered during our modelling.
- When X has a suspended CCBS request and frees a channel, the request is not unsuspended immediately. Thus it is possible for X to make/receive another call – leaving the CCBS request suspended even though X has been temporarily free. This corresponds to a periodic monitoring of X by some kind of polling method.
- A user is allowed to invoke CCBS on himself, and two users may invoke CCBS towards each other.
- Timers are modelled by transitions which generate time-outs in a non-deterministic way. This covers the service duration timer (cancelling old CCBS requests, e.g., after one hour), the destination idle timer (waiting approximately 10 seconds after Y becomes free until the CCBS recall notification is sent to X), and the CCBS recall timer (limiting the period in which X may accept a recall).
- The retention timer (limiting the period in which X may invoke CCBS after receiving a busy signal) was not modelled, since it is incompatible with our compulsory CCBS strategy described above.
- There are three CUGs. We chose this number because we felt it was small enough to avoid undue complexity but large enough to allow all possible interactions.
- It was assumed that every CUG subscriber belongs to at least one CUG. We also assumed that a caller never specifies a CUG to which he does not belong (violating requests can be considered as being immediately rejected).
- Multiple networks were not modelled. This implies that only two CUG checks are performed for each call (unless it is forwarded). In reality, more checks might be performed, e.g., for calls crossing national boundaries.

- A user may put an unlimited number of calls on hold and the retrieval order may differ from the order in which the calls were held.
- A call may only be put on hold after it has been connected, i.e., has been answered.
- A user's CH subscription is checked when he puts a call on hold, but not when he retrieves the call.

6.2 CPN Model of ISDN Supplementary Services

During the construction of the CP-net, we tried to avoid state explosion, since we wanted to use occurrence graph analysis. Hence we tried to model sequences of network events by a single transition whenever this was feasible. For example, when Alice invokes CCBS against Bob, a single transition checks that Bob is free and sends the notification to Alice.

We used a number of places to represent the different states in the basic and supplementary services. As an example, we could have used a single place to represent all call requests. Instead, we used three different places, to emphasise the three stages which a call request passes through.

Our CPN model is atypical – in the sense that we did not use substitution transitions. Instead the pages are glued together by means of a large number of global fusion sets. In the CPN pages shown below, we have hidden all fusion tags (places with identical names belong to the same fusion set). The absence of substitution transitions makes it a bit difficult to get an overview of the structure of our CPN model, and hence we have arranged the page nodes in groups (on the hierarchy page) according to the purpose of the corresponding pages:

- Four pages describe the basic calling facilities (dialling, calls being answered or finding the destination busy, and clearing of calls).
- Five pages describe initialisation and modification of subscriptions.
- Six pages describe Call Forwarding.
- Five pages describe Call Completion to Busy Subscriber.
- Nineteen pages describe outgoing and incoming CUG checks.
- Three pages describe Call Hold.

Most of the pages contain only one or two transitions. Below we describe the details of four of the pages (belonging to different groups). However, first we discuss the colour sets, which are shown in Fig. 6.2.

The first group of colour sets is used to identify *Users* and *Channels*. For example, $(u(2),1)$ denotes the first traffic channel of user $u(2)$. The zero value in *ChannelNo* is used by the Call Hold service to represent a call which is being held by a user without occupying a channel.

The second group of colour sets deals with the three different kinds of forwarding described in Sect. 6.1. The *FWDno* counts the number of times a call has been forwarded.

The third group deals with all the information necessary to handle the user groups. *GroupNo'* is similar to *GroupNo* but contains an extra *NULL* element

which is used in cases where no group number is provided. *InBar* and *OutBar* specify whether the user is allowed to receive calls from other group members and make calls to these (*ICB* means that incoming calls are barred, while *ICA* means that incoming calls are allowed). *InAcc* and *OutAcc* specify whether the user is allowed to receive calls from users outside his groups and make calls to these. As an example, we may have the following *CUG subscription*:

(u(3), [(g(1),ICB,OCA), (g(2),ICA,OCA)], E(g(2)), noIA, OAe).

It tells us that user u(3) is member of the groups g(1) and g(2). In the first group he cannot receive calls from other group members, but he can make calls to them. In the second group he can receive and make calls to other group members. The element (E(g(2))) specifies that the second group is the default. Finally, the user cannot receive calls from users outside his groups (noIA), but he is allowed to make calls to such users (OAe).

OAI and *OAe* stand for implicit and explicit outgoing access. If a user has *OAI* in their *CUG subscription*, the network will always try to connect a call by

```

color User = index u with 1..4;
color UserxUser = product User * User;
color ChannelNo = int with 0..2;
color Channel = product User * ChannelNo;

color FWDtype = with CFU | CFNR | CFB | noFWD;
color FWDno = int;
color FWDinfo = product User * FWDtype * FWDno;

color GroupNo = index g with 1..3;
color GroupNo' = union E:GroupNo + NULL;
color InBar = with ICB | ICA;
color OutBar = with OCB | OCA;
color InAcc = with IA | noIA;
color OutAcc = with OAI | OAe | noOA;
color CUG = product GroupNo * InBar * OutBar;
color CUGlist = list CUG;
color CUGsubsc = product User * CUGlist * GroupNo' * InAcc * OutAcc;

color CUGserv = with useCUG | noCUG;
color OAreq = with useOUT | noOUT;
color CUGinfo = product CUGserv * GroupNo' * OAreq;

color CCBStag = with ACTIVE | SUSP | SERV;
color CCBSreq = product User * User * CUGinfo * CCBStag;

color Request = product Channel * CUGinfo * User * FWDinfo;
color ShortReq = product Channel * CUGinfo * User;
color Calls = product Channel * Channel;

```

Fig. 6.2. Colour set declarations for ISDN supplementary services

that user as a non-CUG (i.e., normal) call if it cannot be connected as a CUG call. If a user has OAc in their CUB subscription, the networks will not do this unless that user specifically requests outgoing access (*useOUT*) in their call request.

The fourth group of colour sets deals with the CUG information from the initial call request. As explained in Sect. 6.1 this information is necessary in connection with the CUG checks. The first two elements of the *CUGinfo* specify whether the call should be made via CUG and if so identifies the group to be used. The last element specifies whether the call should be made via outgoing access if it cannot be made via CUG. The fifth group of colour sets deals with CCBS, while the last group deals with *Requests* and *Calls*. A short request is a request without any forwarding information.

In addition to the colour sets, we declared a number of variables, two constants, and two simple ML functions. The two functions recursively search through a *CUGlist*. One of them checks whether a specified group appears in the list. The other checks whether the list contains repetitions (i.e., two entries for the same group). By convention, colours and constants are written with mostly uppercase letters (e.g., ICB, noCUG and MAXFWD), while variables are written with mostly lowercase letters (e.g., cuglist, inacc and OrigPty). Most variables have a name which is identical to the name of their colour set (except for their capitalisation). The only exceptions are variables where the name ends with *Pty* or *Chan* (the former are of type *User* while the latter are of type *ChannelNo*).

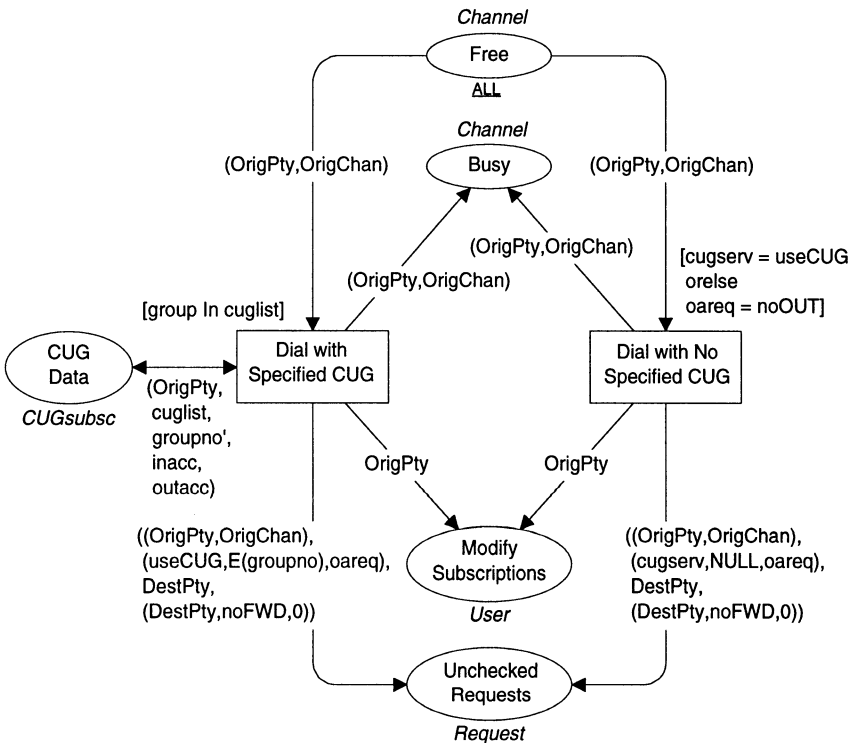


Fig. 6.3. CPN page for Dialling

The CPN page shown in Fig. 6.3 describes the dialling process, i.e., the creation of requests. Initially, each channel is represented by a token on place *Free*. However, as calls are made, some channels become *Busy*. The left-hand transition describes a call in which a CUG is specified. The guard checks that the user is a member of the specified group (*In* is an infix version of one of the recursive functions mentioned above). The right-hand transition describes a call which does not explicitly specify a CUG. However, it is still possible to get a CUG call by setting *cugserv = useCUG*. From the guard, we see that *cugserv = noCUG* implies *oareq = noOUT*. This reflects that non-CUG calls (by convention) get a *CUGinfo* which looks as follows: (noCUG, NULL, noOUT). Both transitions create an *Unchecked Request*. They also add a token to *Modify Subscriptions*. This place is used to control the number of subscription changes (which would otherwise dominate the simulations). The choice to allow one subscription modification per call is fairly arbitrary.

The CPN page in Fig. 6.4 describes how a Call Forward on No Reply (CFNR) is handled. The call to be forwarded has already undergone both outgoing and incoming CUG checks. Hence, it is represented by a token on place *Final Requests*. After the forwarding, the call will have to undergo an additional incoming CUG check at its new destination. Hence, it will be presented by a token on place *Intermediate Requests*. To be able to *Perform CFNR* a number of conditions must be met. They are modelled by the guard and the three double arcs.

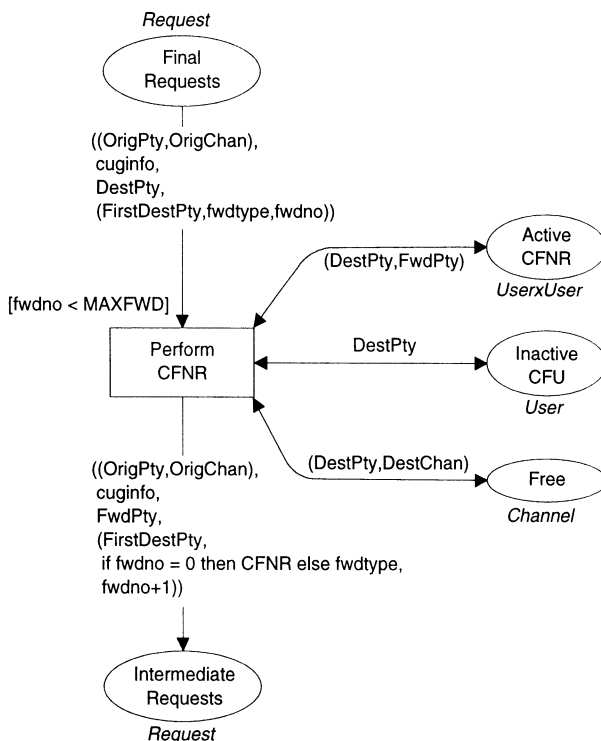


Fig. 6.4. CPN page for *Call Forward on No Reply*

First of all, the number of previous forwards must be less than the forward limit. Secondly the destination must be active for CFNR and inactive for CFU (since CFU takes precedence over CFNR). Finally, the destination must have a free channel (so that the call can be offered, i.e., the phone can be rung at the destination). We do not model this channel as becoming occupied (although it does so for a short period). This means that the transition models (as one indivisible action) the entire sequence of events starting with the call being offered, and finishing with the call being forwarded because it remained unanswered.

What happens to calls that have been forwarded the maximum number of times and then reach a destination where some type of forwarding would be enabled, had the maximum not been reached? For CFU or CFB, the call request must fail, since it can be neither offered nor forwarded. Thus the model has two transitions *Fail CFU* and *Fail CFB* that remove the call request from *Final Requests* and add a channel token to *Dead Channels* (implying that the only thing the user can do is to hang up and thus free the channel). For CFNR such a transition is not needed. Instead the phone keeps ringing at the destination after the CFNR timer has expired.

A token on *Final Requests* represents a call request which is ready to be answered, forwarded, or produce a busy signal. In this state a number of different activities may take place. As an example, the destination exchange may be checking for Call Forwarding, or it may be checking whether the called party is free or busy. It is also possible that the request already has been offered to the destination via a ringing phone. As a consequence of this abstraction, we found

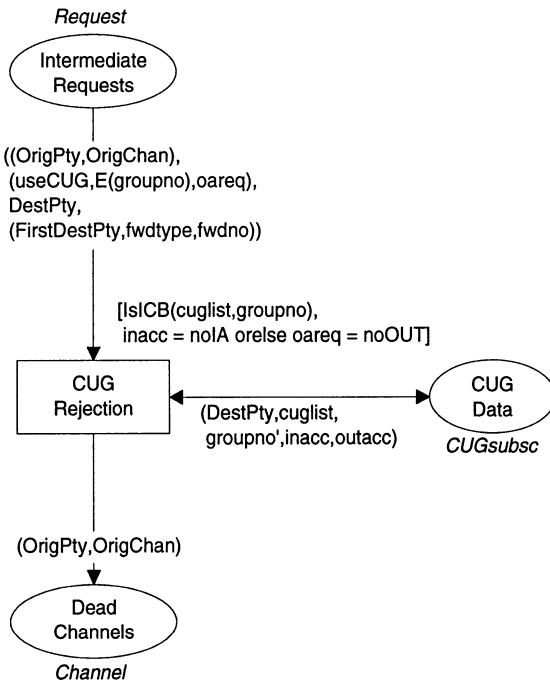


Fig. 6.5. CPN page for *Unsuccessful Incoming CUG Check*

that the CPN model contained interactions which do not exist in the real ISDN network. This could have been avoided by splitting *Final Requests* into a number of places, yielding a slightly more complex net structure.

The CPN page in Fig. 6.5 describes one way in which a call may fail during an incoming CUG check. It is one out of nine pages, which each describe one particular combination of CUG information (in the call request) and CUG subscription (for the destination party). Five of the pages correspond to successful checks while the remaining four correspond to CUG violations. Each of the pages contains a single transition, and these nine transitions are mutually exclusive – in the sense that exactly one of them is enabled (for a given token at place *Intermediate Requests*). This reflects that the network is supposed to work in a deterministic way. Obtaining mutual exclusion between CPN transitions is actually not that easy. In a programming language we could have used a set of nested if-then-else statements, where the “final else” would cover all remaining cases. This is not possible for CP-nets unless we describe all nine combinations with a single transition (or use one transition for the five success cases and another for the four failure cases). Such an approach was discarded, because we thought that the arc inscriptions and guards would become too complex and hence decrease the readability of the CPN model. Analogously, ten different pages describe the details of outgoing CUG checks. Seven of these correspond to successful checks while the remaining three correspond to CUG violations.

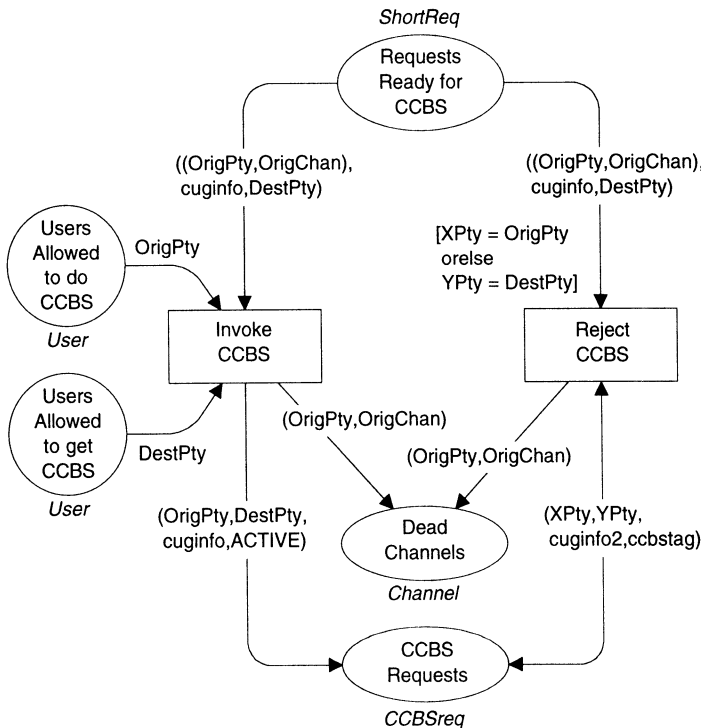


Fig. 6.6. CPN page for *Call Completion to Busy Subscriber*

Now let us consider Fig. 6.5 in more detail. It describes a call specifying a CUG to which the called party actually belongs. However, the call is rejected, because the *CUGlist* for the called party specifies that the group is barred for incoming calls (ICB). This is checked by the function in the first line of the guard. The second line checks that the call is also unacceptable as a non-CUG call, because the destination party has no incoming access (*inacc* = *noIA*) or the calling party is not allowed outgoing access for the call (*oareq* = *noOUT*).

The CPN page in Fig. 6.6 describes the invocation of Call Completion to Busy Subscriber. The left-hand transition describes the cases in which it is possible to *Invoke CCBS*, while the right-hand transition describes the cases in which it is necessary to *Reject CCBS*. Both transitions check call requests positioned at the place *Requests Ready for CCBS*. This is because we modelled invocation of the CCBS service as a two-step process. In the first step (covered on a separate page), the network checks whether the calling user has the option of invoking CCBS. When this is the case, the call request is moved from *Final Requests* to *Requests Ready for CCBS* (simultaneously the *FWDInfo* is removed since it is no longer needed). The transitions in Fig. 6.6 model the situation in which the calling party actually makes a *CCBS Request* towards the called party, after the network has decided that the forwarding history of the call makes this permissible. However, the CCBS invocation may still fail, under certain conditions described by the guard of the right-hand transition. Notice that the arc between this transition and *CCBS Requests* is a double arc. The two transitions in Fig. 6.6 are mutually exclusive in a similar way as described above. As other *CCBS requests* are made or cancelled the enabling of the two transitions may change (without the transitions occurring).

We had planned to perform an analysis based on the underlying PT-net (defined in Sect. 4.6 of Vol. 2). Hence, we avoided non-uniform transitions, i.e., transitions involving a variable number of tokens (cf. Sect. 4.5 of Vol. 1). Using the full power of CPN ML, might have given a more concise CPN model, e.g., by reducing the number of transitions needed to model the CUG checks.

6.3 Validation of ISDN Supplementary Services

When the CPN model had been constructed, it was validated by means of simulation and occurrence graph analysis. The main purpose of this work was to identify interactions between the different services. In particular, we were looking for undesired interactions, i.e., situations in which the activities of one service could harm other services. Due to a tight project schedule and rather limited resources, our validation was not as thorough and detailed as desired. However, we did a limited amount of work and identified a number of useful techniques, which we now describe.

Firstly, it is possible to augment our CPN model by adding a number of **facts**, i.e., transitions which are not expected to be able to occur, because their enabling requirements correspond to states that we do not expect the ISDN services to be able to reach. Detecting a state in which a fact transition becomes enabled means that we have found an unexpected/unwanted state, which may be

caused by an undesired service interaction. To use this technique the modeller needs to acquire some idea about the undesired interactions and the bad states to be avoided. Experience seems to be the key here – the more interactions you find, the easier it is to choose the appropriate fact transitions. A good starting point is facts that detect violations of implicitly assumed CUG restrictions, caused by undesired service interactions.

Secondly, we modified our CPN model by eliminating as many uninteresting occurrence sequences as possible. For example, the number of subscription changes that a user is allowed to make was limited, as explained in Sect. 6.2. In this way we cut down the size of the occurrence graph, making interactions easier to detect – because they occur faster in a simulation and become more visible in an occurrence graph.

Thirdly, it is easy to augment our CPN model to reflect the charging of the individual users, i.e., the amount of money which they have to pay for the services. This is also a way to determine harmful interactions. As an example, placing a user in a local Closed User Group (to prevent expensive long-distance outgoing calls) will not work unless that user is also prohibited from forwarding calls to distant locations (since in ISDN the forwarding user is charged for the forwarded leg of a call). Charging can be modelled by adding an extra place, receiving a token for each charge being made. The token should reflect the identity of the charged user, the amount to be charged, and other details necessary for the subsequent analysis (e.g., information about other users involved and the type of charge).

6.4 Conclusions for ISDN Supplementary Services Project

The creation of a CPN model for the ISDN supplementary services turned out to be very helpful to investigate the potential interactions between the services. It also helped us to identify many ambiguities, omissions, and other shortcomings in the service recommendations.

The entire project was accomplished by two persons over a period of one year. We used a total of nine man-months, including initial training and writing of reports. If we had been able to use more time, we would have made a more thorough validation of the CPN model. We would also have considered the feasibility of modelling the Stage 2 and 3 recommendations. The creation of such models would probably be easier than Stage 1 modelling, since they could be based on the SDL diagrams provided in the Stage 2 recommendations – as done for the BRI protocol presented in Chap. 9. Complexity, however, might limit the number of services which could be analysed together.

Based on the experiences from our project, we have obtained the following wish-list for service specifications:

- An explicit specification of the rules which each service should obey is needed, making it easier to determine whether the behaviour of one service compromises the intentions of another.

- A more precise (mathematical) correspondence between the three stages of a service specification (and their CPN models) is desirable.
- Network-independent Stage 1 recommendations for the services would be useful. This is illustrated by our CPN model, which ignored almost completely the specific characteristics of the ISDN network. The basic analysis of the service interactions could then be performed once, instead of being repeated for different networks.

Chapter 7

Intelligent Network

This chapter describes a project accomplished by *Carla Capellmann, Heinz Dibold, Bettina Hebing, and Eckart Prinz, Deutsche Telekom AG, Technologiezentrum Darmstadt, Germany*. The chapter is based upon the material presented in [11]. The project was conducted in 1993.

We present a project which studied the feasibility of using CP-nets and the CPN tools as part of an object-oriented method for the specification of services in an intelligent telecommunications network. The basic idea of the Object-Oriented Petri Net Method (OOPM) is to regard a system as a set of interacting roles (which are parts of objects). First the individual roles and their interactions are identified. Then each role is specified by means of CP-nets and these nets are validated, both individually and as a whole. Finally, the roles are mapped into objects.

One of the main benefits of the OOPM method is the fact that the same language, CP-nets, is used in the analysis, specification, and design phases. This means that the method avoids the so-called “case gap” where one kind of description has to be translated into a totally different kind of description. This saves work, ensures a large degree of consistency between the different descriptions, and makes it much easier to work in an iterative way.

The project was successful. The new object-oriented method was used, evaluated, and improved. Moreover, a formal and executable model of a network service was constructed and simulated. The project demonstrated the feasibility of the new method and showed that CP-nets can be used to provide a specification of network services. The project also demonstrated the necessity of complementing description languages (such as CP-nets) with system development methods, i.e., guidelines for the use of the language.

Section 7.1 contains an introduction to the intelligent network and the Object-Oriented Petri Net Method. Section 7.2 presents the CPN model of the intelligent network. Finally, Sect. 7.3 presents a number of findings and conclusions for the project.

7.1 Introduction to Intelligent Network

In addition to the classical transmission and switching services, today's telecommunication networks are facing the additional challenge of providing the ability to collect and compute information, i.e., to provide network intelligence. To meet these demands we use Intelligent Networks (IN), which is a concept for the fast and economical provision of new services, going beyond the limits of conventional telephony. Starting from the assumption that each IN service can be made up of a number of service-independent components, a logical and a physical architecture for service provision are defined.

To specify the IN architecture, it is necessary to describe the individual service components as well as their interaction. At the present time, the IN standards describe the architecture and services only in an informal way. As a consequence different interpretations arise and detailed investigations are difficult. Hence, a formal description of the behaviour of parts of the IN standards would be a great improvement.

This description has to be made independently of the technology of the network components. Therefore each component should be regarded as a black box, with a well-defined behaviour but no details about design or implementation. Due to the highly distributed and concurrent nature of telecommunication systems and the complexity of the new services, the specification method must provide:

- Means for a comprehensible description of distributed systems, including the concurrency of the actions in the different components.
- Powerful structuring facilities allowing the user to cope with the complexity of a large system.
- Implementation-independent descriptions.
- Means for execution and verification of the specifications.
- Support for reuse of analysis, specification, design, and implementation.

Based on these considerations, we developed the Object-Oriented Petri Net Method (OOPM), by elaborating an existing functional specification method [10] known as the Open Petri Net Method (OPM). To test the new method we used OOPM to specify and investigate a typical, but fictitious, IN service known as Universal Access Number (UAN).

The UAN service allows a subscriber to use a single universal phone number yet have incoming calls mapped to different terminating line numbers (i.e., routed to different phones). The chosen terminating line depends on the geographical origin of the call (e.g., north, middle, or south zone) and the time at which the call is received (e.g., business, evening, night, or weekend). Two different functionalities are provided: a one-dimensional translation of a given UAN into a terminating line number (as function of the call origin), and a two-dimensional translation (as function of origin and time of the call). The translation scheme can be changed by the subscriber over the phone network (by presenting an authorisation code). The UAN service should be able to coexist with other services, e.g., the *Queuing* service, which distribute incoming calls to a group of terminating lines. If none of the group members are free the call is

queued (up to a maximum queue length). Otherwise, the call is connected to one of the free lines in the group (randomly chosen).

The basic idea of OOPM is to regard a system as a set of interacting roles (which are parts of objects). First the individual roles and their interactions are identified, using the object-oriented method described in [49]. Then each role is specified by means of CP-nets and these nets are validated, both individually and as a whole. Finally, the roles are mapped into objects. Figure 7.1 provides an overview of the OOPM method. A more detailed description can be found in [11].

The upper part of Fig. 7.1 shows that first the abstraction level is chosen, determining the amount of details in the specification. Then the system and its relevant environment are described. The system is regarded as a black box and the desired input/output behaviour is specified. Next the system is divided into subtasks. For example, the UAN service is divided into *Service Subscription*, *Service Modification*, *Service Cancellation*, and *Service Usage*. For each subtask a **role model** is developed which describes the subsystem as a structure of objects that play certain roles and interact with each other, to fulfil the task of the subsystem. The roles describe partial behaviours, i.e., behaviour in a certain context or seen from a certain view.

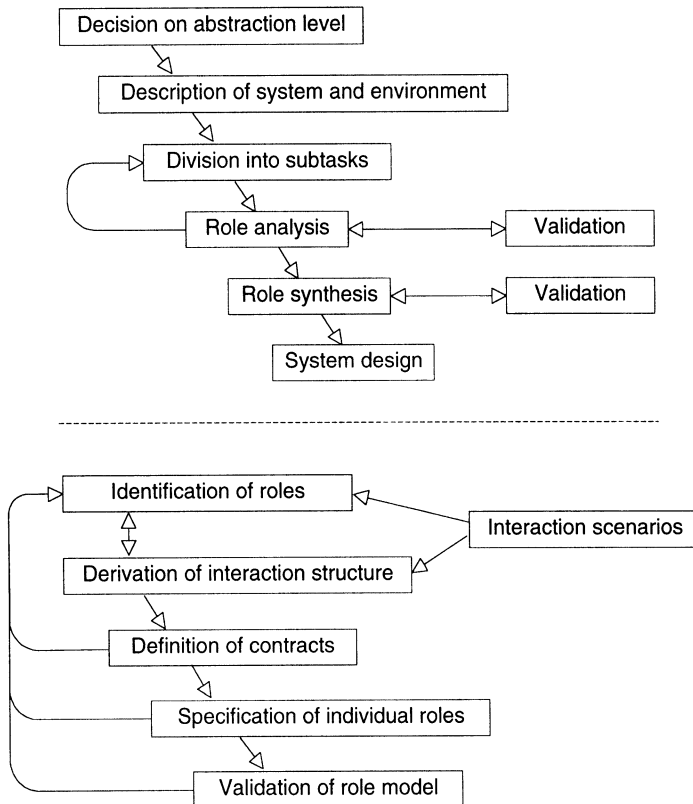


Fig. 7.1. Overview of Object-Oriented Petri Net Method (OOPM)

The role analysis is performed as shown in the lower part of Fig. 7.1. First we identify the individual roles and derive the required interactions between these. This can be done, e.g., by considering a number of interaction scenarios. The result is a **role diagram**. Then the possible interactions between the roles are defined by means of **contracts** which specify the set of messages to be used. Next, a **role specification** is made for each individual role. It consists of a CP-net that describes the detailed externally perceivable behaviour of the role, conserving the maximum amount of concurrency. We then have a **role model** for the considered subtask. It consists of a role diagram, together with a number of contracts and role specifications (more details will be provided below).

When the role models for all subtasks are constructed and validated, they are integrated into a role model for the entire system. During this synthesis process, it may be adequate, or even necessary, to merge related roles of different role models. The final role model serves as input for the system design. Here, the roles are mapped onto classes, thereby determining the architecture of the system. The individual classes may then be specified by reapplying the steps of OOPM using a different, more detailed abstraction level. Note that validations are done throughout the entire specification process and not just afterwards.

Roles are similar to classes, in the sense that they have an internal state and a number of methods that can be used to inspect and modify the state. Each role is described by a CP-net, which often consists of a single page. The state is represented by a number of places known as **internal places**, while each method is represented by one or more transitions, which receive request messages and return reply messages via a number of **interface places**. The state of a role can only be changed or inspected by means of the methods. This is reflected by the fact that the CP-net for a role is an open subnet having the interface places as border nodes and the internal places as non-border nodes (when considered part of a large non-hierarchical net).

An object playing a role may be seen as an instance of that role. Hence, the object is represented by an instance of the CPN page which represents that role. The marking of the internal places represent the state of the object, while the marking of the interface places represent messages to or from the object. The communication is asynchronous, since messages are represented by tokens which are deposited by one transition and later removed by another.

For the development of industrial systems, elaborate tool support is mandatory. Hence, we would want a tool which supports all steps in OOPM. However, since OOPM is a new method, such a tool does not exist. Instead we used the existing CPN tools – often in a rather untraditional way. As an example, we constructed the role diagrams in such a way that the roles were represented as substitution transitions (drawn as large circles), while the other elements were auxiliary objects. This automatically created a link from each role to a page containing the detailed CPN specification of the role.

7.2 CPN Model of Intelligent Network

This section presents the results of applying OOPM for the specification of the Universal Access Number (UAN). We concentrate on the role analysis, presenting the role models for *Service Subscription* and *Service Usage*. The results of other parts of the OOPM will only be sketched.

Service Subscription is one of the administrative tasks that the UAN service has to perform. Triggered by a request for service subscription, the system checks whether a subscription is possible, performs the necessary actions, and makes an appropriate response. The role diagram for this subtask is given in Fig. 7.2. The large circles represent different roles in the subsystem and its environment (the roles of the environment are drawn with a thicker border line).

- The *Subscriber* belongs to the environment of the UAN system. This role initiates a service subscription by sending a request.
- The *Subscription Manager* coordinates all actions that are needed to perform the different kinds of subscriptions (e.g., a UAN subscription).
- The *UAN Provider* delivers, on request, an available UAN. Analogously, the *AuthCode Provider* delivers an available authorisation code.
- The remaining three roles are data managers that store different kinds of data.

The lines between the roles represent the possible interactions (thick lines are used for interaction with the environment, while thin lines are used for interactions inside the system). The small circles represent the contracts that define the messages to be used for the interaction (a double circle indicates that the sending role knows several objects which play the role of the receiver, while a single circle tells us that there is only one such object). We see that the *Subscription Man-*

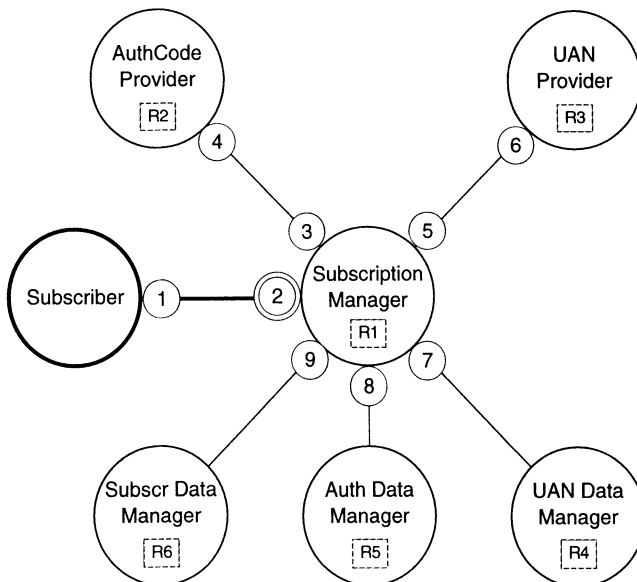


Fig. 7.2. Role diagram for *Service Subscription*

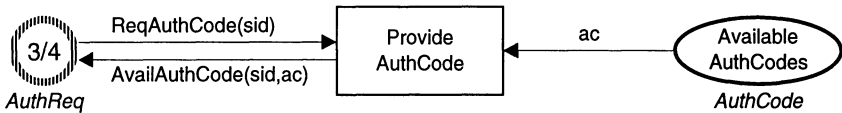
ager is able to send messages to all the other roles, while it is only able to receive messages from three of them.

The upper part of Fig. 7.3 shows the role specification for the *AuthCode Provider* (which is very simple). The internal state of the role is represented by the place *Available AuthCodes*. It contains a token for each of the available authorisation codes. With respect to the *Service Subscription* role there is only one method, represented by transition *Provide AuthCode*. The method is invoked by sending a *ReqAuthCode* message (via the interface place *3/4*, which is used to hold messages defined in contracts 3 and 4). When the transition occurs, the method is executed (as an atomic action). The *sid* variable is an input parameter specifying the subscriber identity, while the *ac* variable is an output parameter returning an authorisation code.

The colour set declarations in Fig. 7.3 specify messages to be exchanged via the interface place *3/4*. This means that they reflect parts of contracts number 3 and 4. In our project we specified the contracts directly in Standard ML (as a set of colour set declarations). However, it would probably be better to specify the contracts in a different way, and then derive the colour set declarations from the contracts (preferably totally automatic). The integer type *AuthCode* represents authorisation codes (with six digits), while the union type *AuthReq* represents request and reply messages. It contains all elements which are on the form *ReqAuthCode(sid)* or *AvailAuthCode(sid,ac)*, where $sid \in SId$ and $ac \in AuthCode$. For more details on union types, see Sect. 1.4 of Vol. 1.

The role specification of the *UAN Provider* is very similar to the specification of the *AuthCode Provider*. The main difference is that *AuthCode* is replaced by a type which represents the possible UANs.

AuthCode Provider



```

color SId = string;
color AuthCode = int with 100000..999999;
color SIdxAuth = product SId * AuthCode;
color AuthReq = union ReqAuthCode : SId +
                    AvailAuthCode : SIdxAuth;
var sid : SId; var ac : AuthCode;
    
```

UAN Data Manager

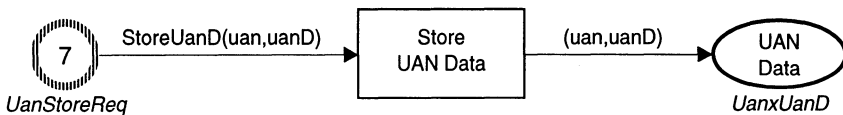


Fig. 7.3. Two role specifications for *Service Subscription*

The lower part of Fig. 7.3 shows the role specification for the *UAN Data Manager*, which is responsible for the storage of *UAN data*, i.e., the translation schemes that map call origin (and possibly call time) into terminating lines. For this (and all following) role specifications we have omitted the colour set declarations. They provide details about the format of the stored data. The variable *uan* denotes a universal access number, while the variable *uanD* denotes a translation scheme (i.e., a piece of *UAN data*). The two other data managers (in the lower part of Fig. 7.1) have role specifications which are similar to the *UAN Data Manager*.

Figure 7.4 shows the role specification for the *Subscription Manager*. It has a single method which interacts with six different roles (via the six interface places). A subscription is performed in two steps. When a subscription request $ReqSubscr(sid, uanD)$ is ready (at place 1/2), the upper transition occurs. It reads the request in which *sid* represents the subscriber identity while *uanD* is the subscription data. Two messages are sent to the *AuthCode Provider* and the *UAN Provider* (via places 3/4 and 5/6). When these roles reply (via messages at the same two interface places), the lower transition becomes enabled. It sends a subscription confirmation $ConfSubscr(sid, uan, ac)$ to the *Subscriber* (via 1/2). Moreover, three messages are sent (via 7, 8, and 9). They request the three data managers to update their data for *uan*. The left manager stores the subscriber identity *sid*, while the other two store the authorisation code *ac* and the translation scheme *uanD*.

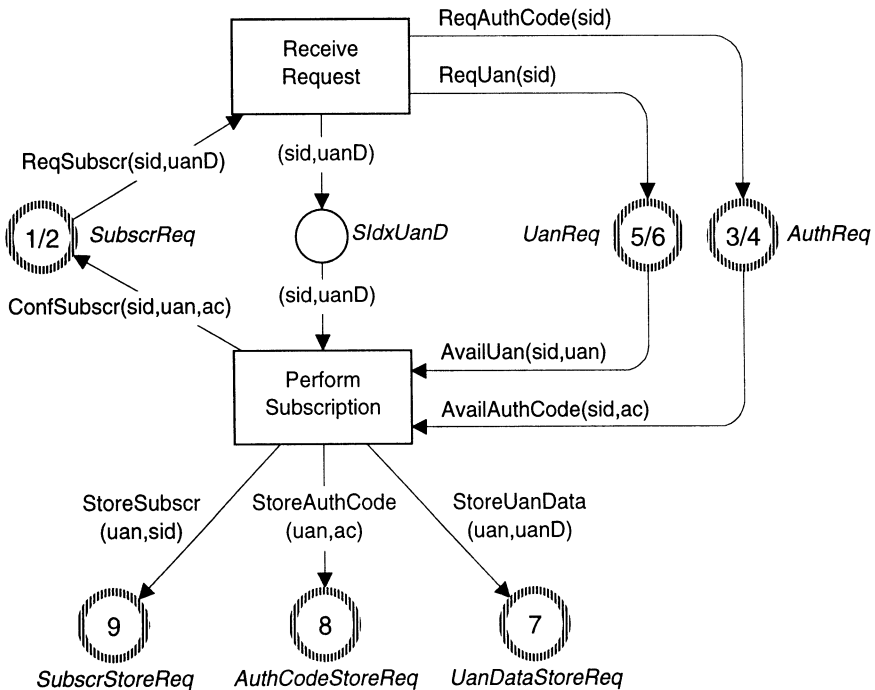


Fig. 7.4. Role specification for *Subscription Manager*

Note that the method in Fig. 7.4 is specified in such a way that the maximum degree of concurrency is maintained. The *AuthCode* and the *UAN* are requested in parallel and so are the three database updates. Moreover, it is possible to perform the upper and lower transitions concurrently for two different requests. We have only modelled the successful case, in which the subscription is possible (because an *AuthCode* and a *UAN* both are available). However, it is easy to extend the CP-net in Fig. 7.4 by adding an extra transition to take care of the situation in which one or both replies (on places 3/4 and 5/6) represent reject messages.

Each of the role specifications is validated by means of simulation. This is usually done as soon as the role specification has been finished (or even before). To make such a simulation, the modeller “emulates” the surrounding roles, by using the Change Marking command (in the CPN simulator) to create tokens representing messages from these roles. It is also sometimes necessary to add tokens to some of the internal places. This is the case, e.g., for a role which retrieves data that are stored by another role.

When all roles of a role model have been specified and validated, the entire role model is validated to see whether the individual parts are consistent and interact in the expected way. This is done by means of simulations in which the CPN pages for the individual roles communicate via the interface places, which are global fusion places. To improve readability we have drawn these places with a special line pattern and hidden the fusion tags (since they provide no additional information).

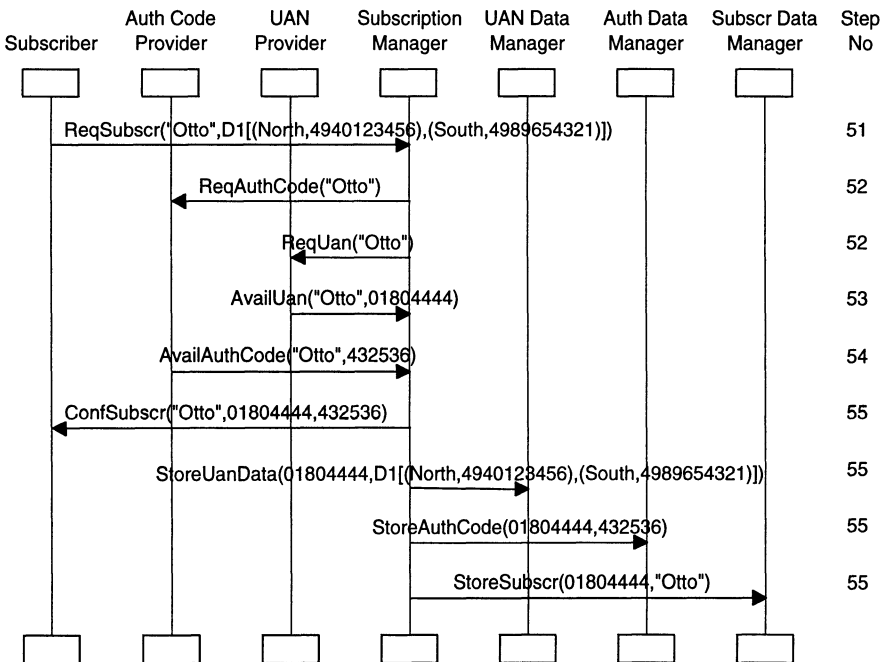


Fig. 7.5. Message sequence chart from simulation of *Service Subscription*

To inspect the simulation results we used message sequence charts like the one shown in Fig. 7.5. The diagram displays the messages which are sent between the different roles. First we have a *ReqSubscr* message from a *Subscriber* “Otto” to the *Subscription Manager*. The request specifies a one-dimensional translation scheme mapping calls from *North* to 4940123456 and calls from *South* to 4989654321. Then we have a *ReqAuthCode* message from the *Subscription Manager* to the *AuthCode Provider* and a *ReqUan* message from the *Subscription Manager* to the *UAN Provider*. Both of these occur at step 52, and hence they are concurrent. In the next two steps the *Subscription Manager* receives messages containing an available UAN and an available authorisation code. Then the *Subscription Manager* confirms the subscription, and sends messages to the three database managers. All four messages are concurrent (at step 55).

Message sequence charts can now be automatically created by the CPN simulator, by making calls to a Standard ML library (from code segments attached to the individual transitions). The diagrams make it very easy and fast to investigate whether simulations have the expected behaviour. For more information about message sequence charts, see the User’s Manual [15].

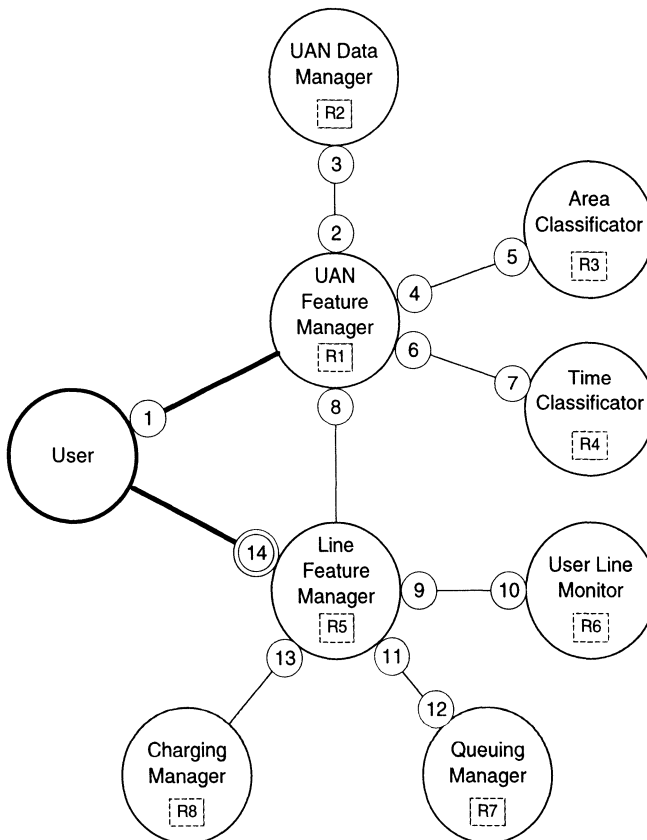


Fig. 7.6. Role diagram for *Service Usage*

Above, we have presented the role model for *Service Subscription*. Now, we consider the role model for *Service Usage*, i.e., the subtask that describes the actual use of the UAN service. When a user dials a UAN, the service connects the call to a terminating line, which is determined by first computing the call destination specified by the translation scheme of the UAN, and then taking possible queuing for a group of lines into account. The role diagram for *Service Usage* is shown in Fig. 7.6. It defines the following roles:

- The *User* belongs to the environment of the UAN system. This role initiates a service usage by dialling a UAN.
- The *UAN Feature Manager* coordinates all actions that are needed to compute the call destination, while the *Line Feature Manager* is in charge of establishing the connection.
- The *UAN Data Manager* retrieves the UAN data, i.e., the translation scheme for a given UAN.
- The *Area Classifier* and the *Time Classifier* determine the area from which the call was made and the time at which it was received.
- The *User Line Monitor* administers the status (idle/busy) of the individual lines. It also records how lines are grouped.
- The *Queuing Manager* administers a queue of calls for each group of lines, while the *Charging Manager* is responsible for all charging aspects.

Figure 7.7 shows the role specifications of the *UAN Data Manager*, the *Area Classifier*, and the *Time Classifier*. All of them are very simple, and similar to the specification of the *AuthCode Provider* in Fig. 7.3 (except that the rightmost arc now is a double arc). The variable *cli* is used to denote the number of the calling line.

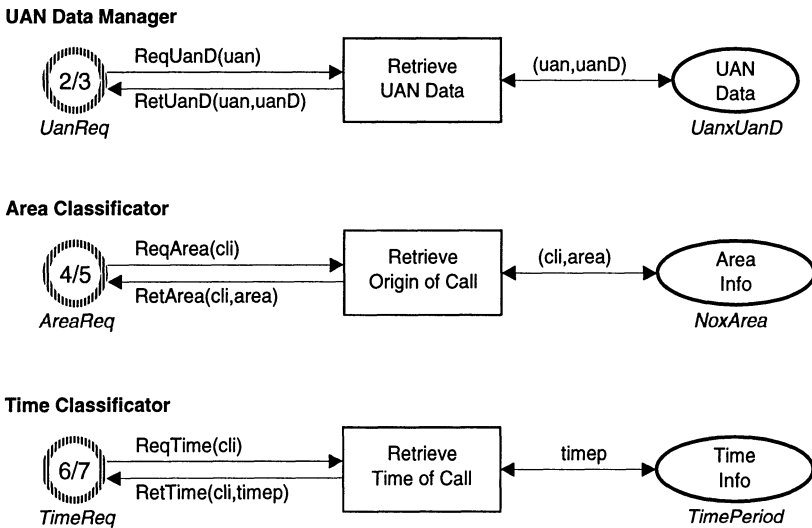


Fig. 7.7. Three role specifications for *Service Usage*

Figure 7.8 shows the role specification for the *UAN Feature Manager* which is responsible for the computation of the destination for a UAN call. Triggered by an *IncomingCall* specifying the calling line *cli* and the dialled universal access number *uan*, the upper transition sends a message to the *UAN Data Manger* (via the interface place 2/3). The reply contains the translation scheme *uanD* for *uan*. Next, a message is sent to the *Area Classifier* (via 4/5). If the translation scheme is two-dimensional another message is sent to the *Time Classifier* (via

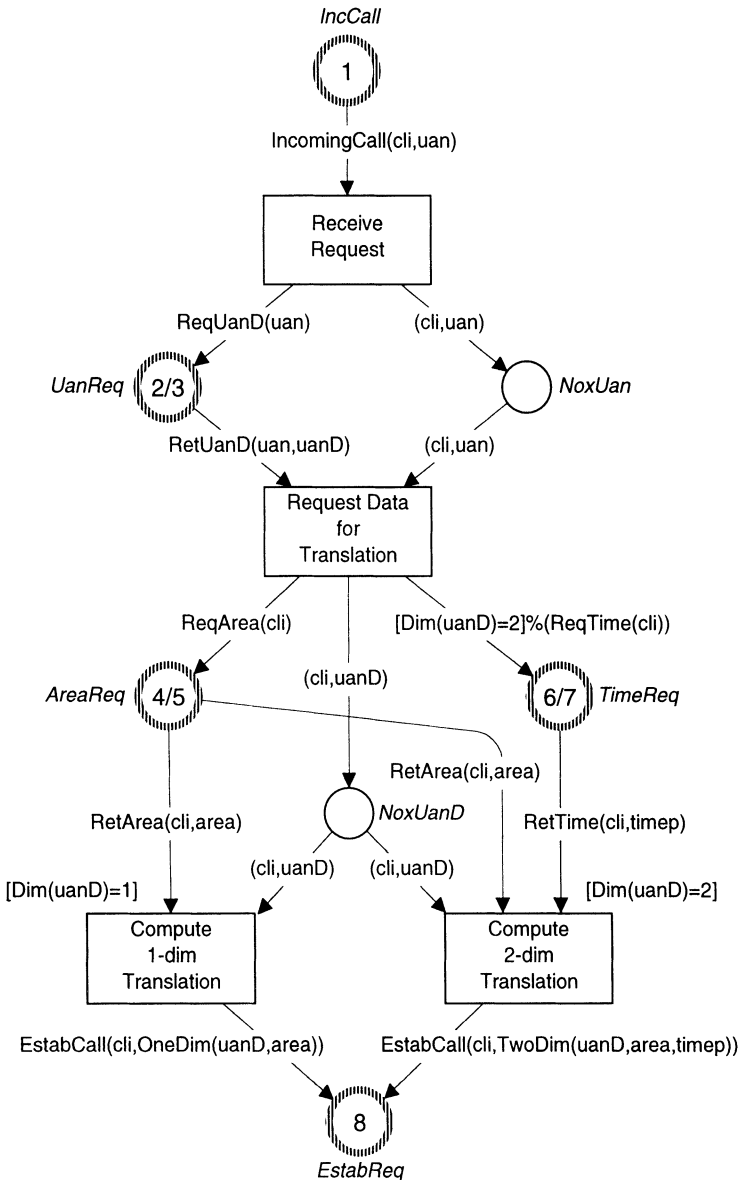


Fig. 7.8. Role specification for *UAN Feature Manager*

6/7). This arc expression uses the %-operator, which is a shorthand for an if-then-else construction. When the left-hand expression evaluates to true, the value is the value of the right-hand expression. Otherwise the value is the empty multi-set.

The reply/replies are handled either by *Compute 1-dim Translation* or by *Compute 2-dim Translation* (using the ML functions *OneDim* and *TwoDim*, respectively). In both cases, we end up with an *EstabCall* message on place 8. The token colour contains a pair where the first element is the calling line, while the second is the terminating line to which the call should be connected (unless modified by the *Queuing* service).

Figure 7.9 shows the role specification for the *Line Feature Manager* which either offers the call directly to the terminating line or passes it through a wait-

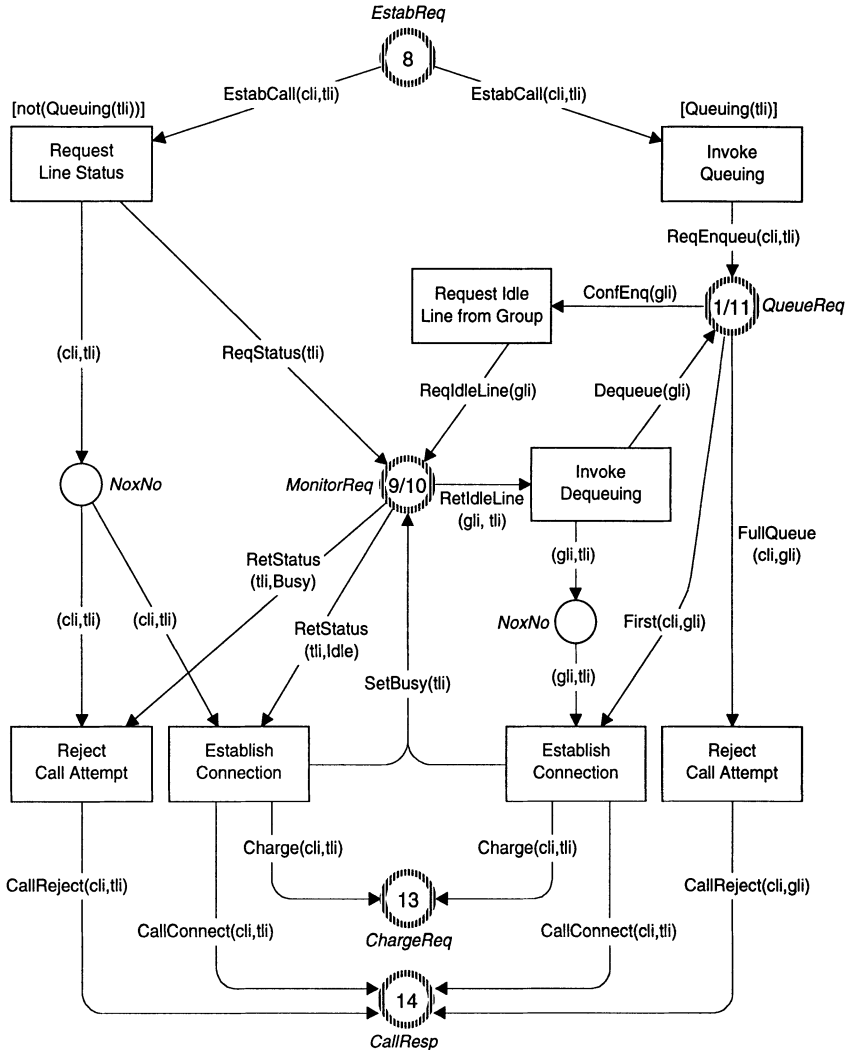


Fig. 7.9. Role specification for *Line Feature Manager*

ing queue for a group of lines. Direct calls are handled by the left-hand part of the CP-net, while group calls are handled by the right-hand part. For a direct call, we first *Request Line Status* for the terminating line *tli*. Then we either *Reject Call Attempt* or *Establish Connection*. A group call is handled in a similar way, but now there are additional transitions to *Invoke Queuing*, *Request Idle Line from Group*, and *Invoke Dequeuing*. In all cases, we end up by sending a call response to the *User* (via 14). If the call attempt was successful, we also send a message to the *Charging Manager* (via 13).

Figure 7.10 shows the role specification for the *User Line Monitor* which administers the status of terminating lines. It also records how lines are grouped. The role offers three methods to *Check Line Status*, *Set Status Busy*, and *Retrieve Idle Line from Group*.

We have now specified all the individual roles from the role diagram in Fig. 7.6 except *User*, *Queuing Manager*, and *Charging Manager*, which we shall omit. A typical message sequence chart for a simulation of the combined roles (i.e., the subtask *Service Usage*) is shown in Fig. 7.11. It represents a successful UAN call which uses the one-dimensional translation scheme from Fig. 7.5 and no queuing.

The role models for the remaining two subtasks (*Subscription Modification* and *Subscription Cancellation*) are specified, in a similar way as presented above. When they have been specified and validated we are ready for the synthesis phase of OOPM. In this phase the four role models are integrated into a single CPN model which represent the entire UAN service and its environment. The final role model is shown in Fig. 7.12. During the integration some roles are merged. As an example, Fig. 7.12 contains a *UAN Data Manager* with the role specification shown in Fig. 7.13. This role is obtained by merging a number of *UAN Data Managers* described in the different role diagrams (e.g., the *UAN Data Manager* from Fig. 7.3 and the *UAN Data Manager* from Fig. 7.7).

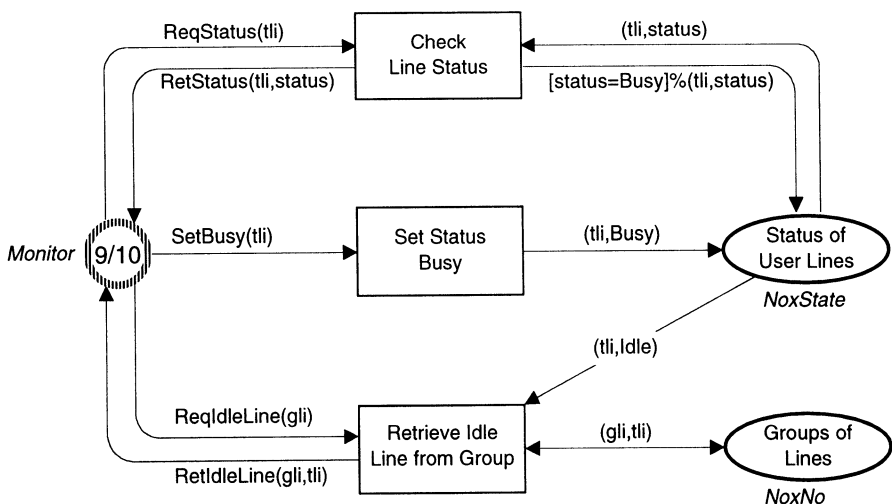


Fig. 7.10. Role specification for *User Line Monitor*

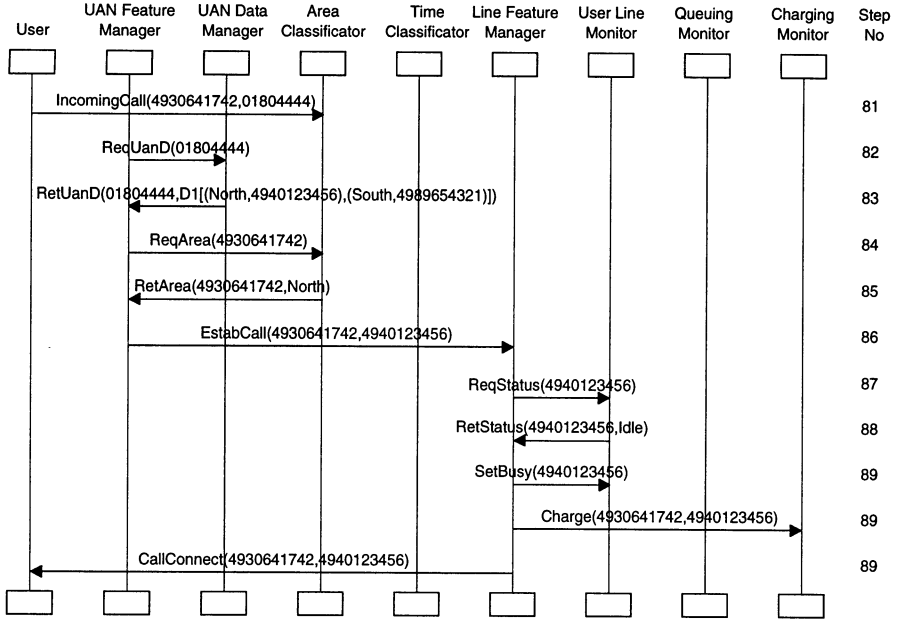


Fig. 7.11. Message sequence chart from simulation of *Service Usage*

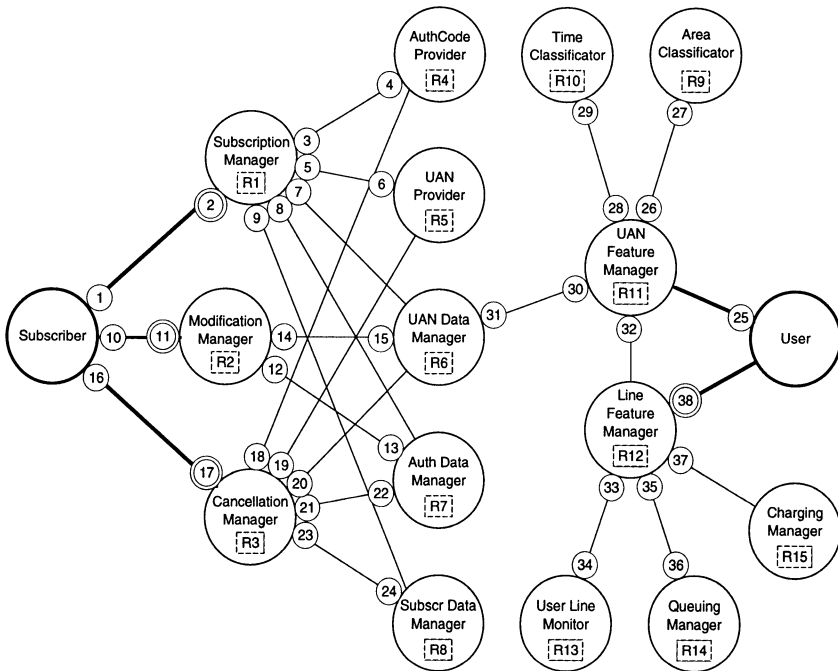


Fig. 7.12. Synthesised role model of UAN

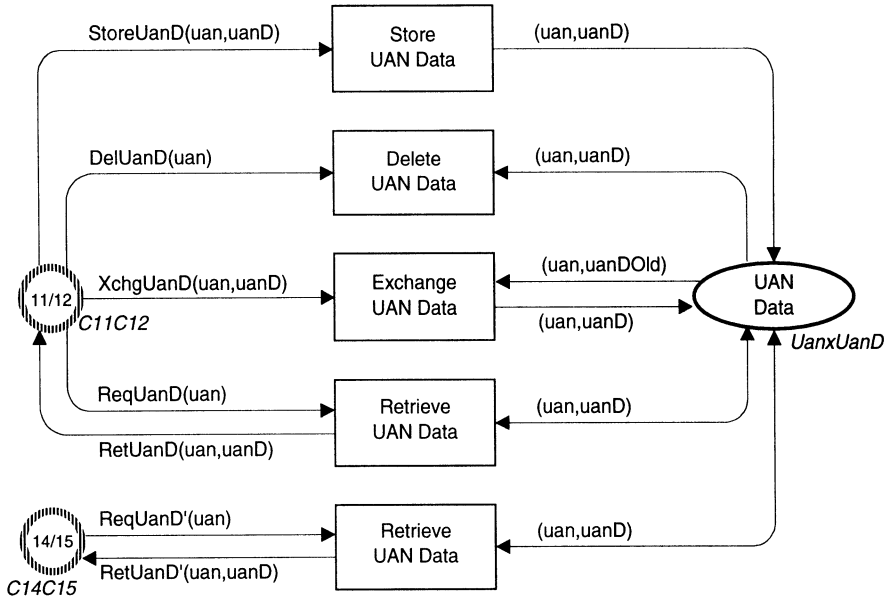


Fig. 7.13. Role specification for synthesised UAN Data Manager

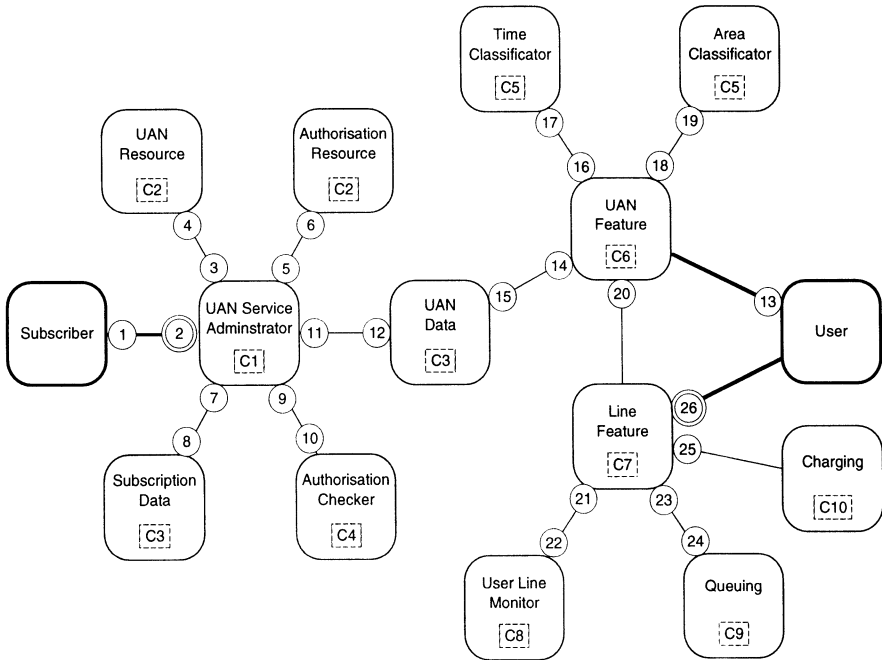


Fig. 7.14. UAN design model

During the design phase, the roles are mapped onto classes to be used during the design and implementation. This gives us the design model shown in Fig. 7.14. A class library facilitates the reuse of class specifications and implementations. As examples, we see that *UAN Resource* and *AuthCode Resource* are both based on class *C2*, while *UanData* and *SubscriptionData* are based on *C3*.

7.3 Conclusions for Intelligent Network Project

The project leader was a senior expert, who was knowledgeable on both intelligent networks and CP-nets. He supervised the work and advised the project group, but otherwise he did not participate. The other three project members had less than one year of experience with intelligent networks. With respect to Petri nets, two of them had only rudimentary knowledge while the third had a theoretical background in Petri nets, but no modelling experience. None of them knew the CPN tools. The project was carried out over a period of one month in which all three persons worked part-time on the project. The total use of manpower is estimated as five weeks. During this time the project members learned how to use the OOPM method and how to model and simulate by means of CP-nets. They also became acquainted with the details of the UAN service.

The final system model consists of 17 roles, of which 14 were modelled by means of CP-nets. As illustrated by the examples in Sect. 7.2, some of the nets are very simple while others are a bit more complex. Altogether, the CPN model contains 41 interface places, 20 internal places, and 43 transitions. The model constitutes a formal specification of the UAN service. It enables a precise investigation of the details of the service and makes it possible to locate points where the informal descriptions are vague, ambiguous, or even erroneous.

Since OOPM is a new method, which is still under development, the project group had to make a lot of decisions concerning the detailed mapping of object-oriented concepts into CPN concepts. A number of alternatives have to be investigated before a full assessment can be given. However, a couple of general problems have been recognised:

- Some tasks are divided into actions performed by different objects. When this is the case, it is difficult to obtain atomicity of transactions.
- In our approach, the recipient of a message is addressed by directing the message token to a particular interface place. This differs from object-oriented languages, where objects usually are addressed via object identifiers.

Although the CPN tools know nothing about the OOPM method, the tool set was able to support the complete modelling and validation process. With a modest amount of work the tools could be extended to provide a better support for role diagrams, including a check of their consistency with role specifications. We only encountered a few tool-related problems:

- The turnaround time between the editor and simulator was quite long. We gradually extended the declarations to represent more and more contracts. Each time new declarations were added, we had to recheck the entire model and generate the simulation code from scratch. Since our project, a new version of the CPN tools has been developed, providing a faster and more incremental syntax check and code generation.
- We found no tool support for inheritance. Several roles have the same behaviour – except for the data types that they use. Examples are the *AuthCode Provider* and *UANProvider*. We would have liked the CPN tools to support page instantiation with parametrisation of the involved colour sets.
- In Standard ML it is not possible to reuse constructor names. Hence, we sometimes had to invent two slightly different message names. Examples can be found in Fig. 7.13, where the two lower transitions receive the messages *ReqUanD* and *ReqUanD'* and send the messages *RetUanD* and *RetUanD'*.

The two main goals of the project were both successfully achieved. The Object-Oriented Petri Net Method was used, evaluated, and improved. Moreover, a formal and executable model of the UAN service was constructed and simulated. The project demonstrated the feasibility of OOPM and showed that CP-nets are suitable for the specification of IN services. The project also demonstrated the necessity of complementing description languages (such as CP-nets) with system development methods (such as OOPM). Among the other achievements, the following points should be mentioned:

- It was possible to construct a well-structured, formal and executable model of the UAN service using rather few resources.
- The CPN model was validated by means of simulation (and it could probably also have been verified by means of occurrence graphs). Hence our trust in the correctness of our UAN specification is quite high.
- The CPN specifications of roles describe the pre- and post-conditions of each method. The nature of Petri nets make it straightforward to maintain the maximal degree of concurrency.
- The executability of the CP-nets makes it easy to check the different parts of the system specification towards our own (informal) conception of the desired behaviour – of a role, role model, or the entire system.
- Validation of the CPN models was performed throughout the entire project and not just at the end. This provided immediate feedback to the project group, yielding additional insight which could be used to improve the specification of the remaining system parts.

A brief comparison of OOPM with two other system development methods (including OPM) can be found in [11].

Chapter 8

Communications Gateway

This chapter describes a project accomplished by *Daniel J. Floreani, Defence Science and Technology Organisation, Adelaide, Australia, in cooperation with Jonathan Billington and Arek Dadej, University of South Australia, Adelaide, Australia.* The chapter is based upon the material presented in [25]. The project was conducted in 1995.

We present a project in which a gateway between a Tactical Packet Radio Network and Broadband ISDN is being designed – as part of a large project that aims to bring modern telecommunications services to the Australian Defence Force. We used CP-nets and the CPN tools to investigate the gateway architecture and behaviour prior to implementation. In particular, we investigated the call control application of the gateway. We first developed a CPN model specifying the service to be provided by the gateway. Then we developed a CPN model constituting a refined specification, involving more architectural aspects of the gateway.

We compared the behaviour of the refined specification with the original specification. The two CPN models are at different levels of abstraction. However, the interfaces to them are common, and hence we can compare the languages determined by the interface primitives. After several iterations of editing, simulation, and occurrence graph analysis, the two models were proven to have the same languages, i.e., the same sequences of interface primitives. During this work, we removed many minor errors and we also located three more fundamental modelling errors.

The use of CP-nets and the CPN tools within the specification stage of the gateway has been successful. The fact that the first attempts to refine the specification did not meet the original gateway specification shows how easy it is to make mistakes when specifying the behaviour of systems.

Section 8.1 contains an introduction to the communications gateway. Section 8.2 presents the two CPN models of the gateway. We also discuss how the models were validated and how they were compared to each other. Finally, Sect. 8.3 presents a number of findings and conclusions for the project.

8.1 Introduction to Communications Gateway

The Australian Department of Defence has chosen Asynchronous Transfer Mode (ATM) as the target communications architecture for its command and control in the 21st century. To provide modern and flexible communications to the soldier in the field, a Tactical Packet Radio Network will be integrated with Broadband ISDN (B-ISDN). To do this, a gateway between the two kinds of networks must be specified and implemented.

We describe the specification of the Call Control Application (CCA) of the gateway. It is responsible for the transfer of call control intent between the radio network and B-ISDN. As can be seen from the left-hand side of Fig. 8.1, the Call

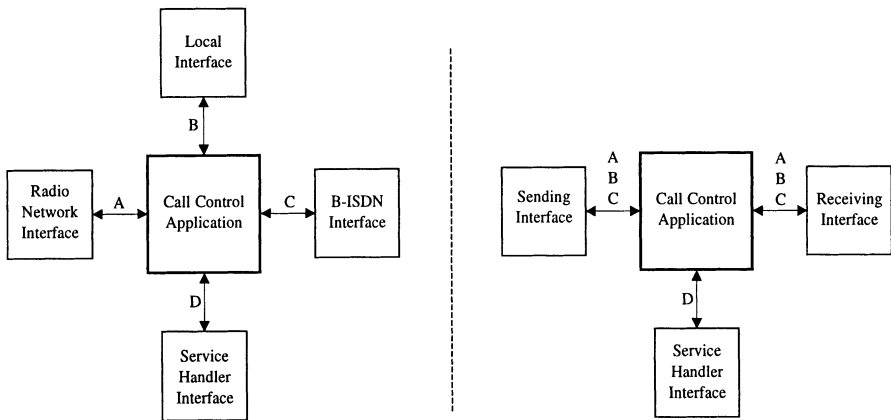


Fig. 8.1. Call Control Application and its interfaces

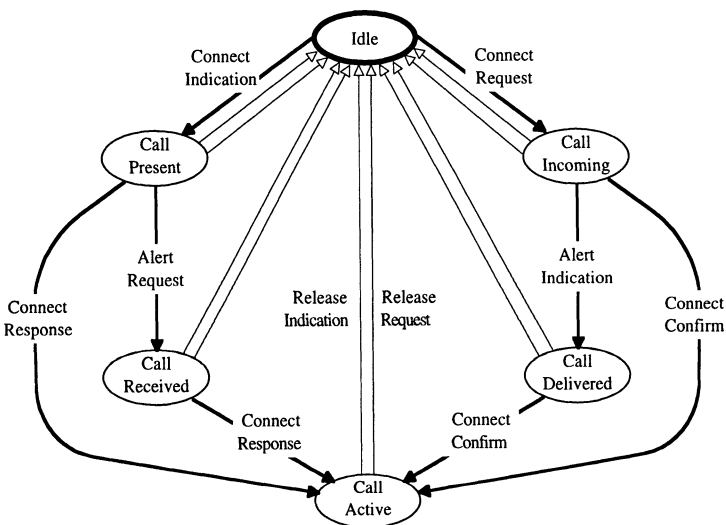


Fig. 8.2. State/transition diagram for interfaces A-C

Control Application has four interfaces – to the radio network, to the local user, to the B-ISDN, and to the service handler. The latter is a process that manages the specific end user service transfer through the gateway (voice, data, messaging). It turns out that the first three of these interfaces have the same service specification, and hence it is more appropriate to view the interfaces as shown in the right-hand side of Fig. 8.1. Here, we have merged interfaces A, B, and C into a single interface. However, simultaneously, we have split this common interface into a sending interface and a receiving interface. This simplifies the modelling task without any loss of generality, since sending and receiving are performed by two totally independent processes. The state/transition diagram of the common interface is shown in Fig. 8.2. The left-hand part constitutes the sending interface while the right-hand part constitutes the receiving interface. It is possible to return to state *Idle* from all other states via a *Release Indication* or a *Release Request*. For readability, this has only been shown on the arcs from *CallActive*.

A much more detailed description of the gateway can be found in [25]. There we also describe all the different steps in the methodology by which the gateway is developed.

8.2 CPN Model of Communications Gateway

A CPN model of the sending interface is shown in Fig. 8.3. It uses the colour set *Istate* which has a value for all possible states of the sending and receiving interfaces:

```
color Istate = with idle | callPres | callInc | callRec | callDel | callAct;
```

The CP-net has the occurrence graph shown in Fig. 8.4. It is straightforward to see that it matches the left-hand part of the state/transition diagram in Fig. 8.2.

The receiving interface and service handler interface were modelled in a similar way and compared to their state/transition diagrams.

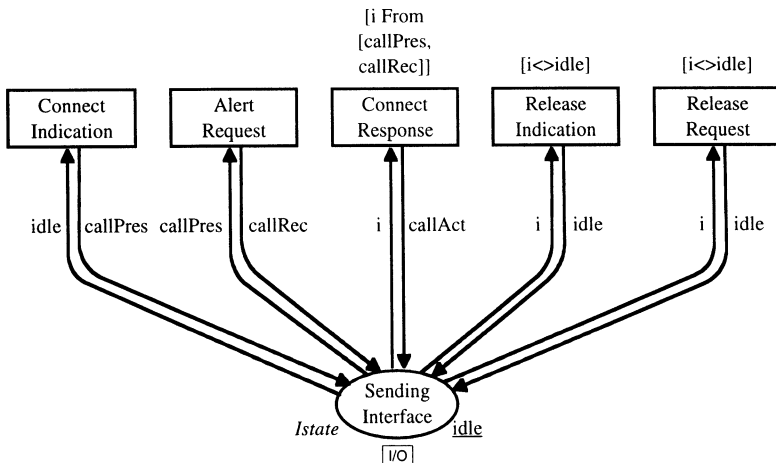


Fig. 8.3. CPN model of the sending interface

When this had been done, the next task was to develop the service specification of the gateway. This required several modelling iterations. The main problem was to determine an adequate level of abstraction and the actual type of services to be modelled. To avoid making too many design decisions during the service specification, we focused on the information flow, ignoring gateway states, messages, and internal primitives. Previously, each interface was considered in isolation. Now we describe how the primitives interact with each other.

The most abstract view of the new CPN model is shown in Fig. 8.5. It shows that all communication between the *SendingInterface*, the *ReceivingInterface*, and the *ServiceHandlerInterface* passes through the *CallControlApplication*. All transitions are substitution transitions and the subpage for *SendPrimitives* is shown in Fig. 8.6. The lower part is identical to Fig. 8.3. Now the transitions no longer just change the state of the *SendingInterface*. They also add and remove information from the *CallControlApplication*.

The colour set declarations are shown below. *Info* has an element for each of the service primitives. The argument to *CallRelease* tells whether it is the sender or the receiver that initiates the release. The *Cstate*, *Count* and *Clean* colour sets are explained below.

```

color SR = send | receive;
color Info = union CallInfo + CallAccept + CallResponse + CallReject +
             CallRelease: SR + Alert + ServiceRelease + Configure;
color Cstate = with init | clean | normal;
color Count = int with 0..10;
color Clean = product Cstate * Count;
    
```

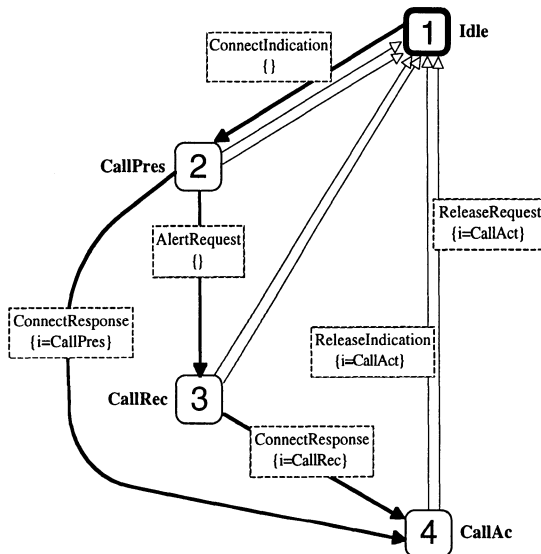


Fig. 8.4. Occurrence graph for the sending interface

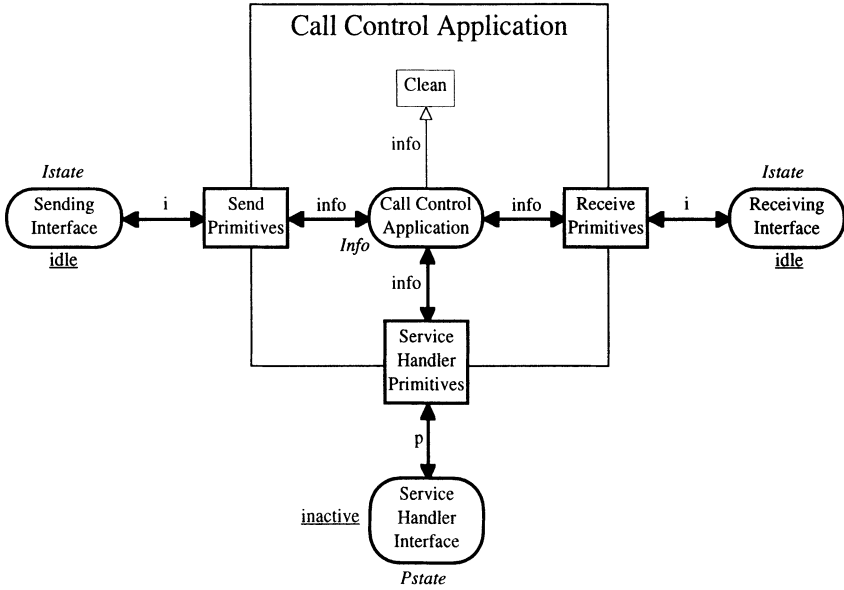


Fig. 8.5. Most abstract CPN view of *Call Control Application*

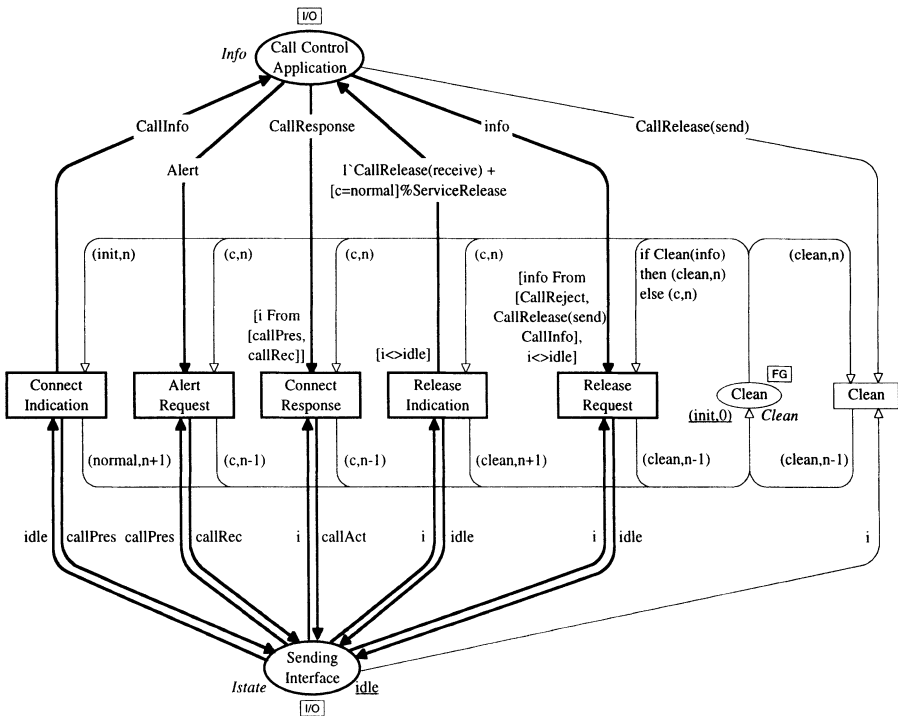


Fig. 8.6. CPN page for *Send Primitives*

The net elements with thick lines are the most important. They describe how the sending interface behaves when different primitives are performed. The *From* operator (in two of the guards) returns true if the left argument appears in the right argument (which is supposed to be a list). The operator is defined as shown below and is polymorphic. In the guard of *ConnectResponse* it is used for elements of *Istate*. In the guard of *ReleaseRequest* it is used for elements of *Info*.

```

fun From(Elem,First::Rest) =
    if Elem = First then true else From(Elem,Rest)
| From(Elem,[]) = false;
infix From;
    
```

The net elements with thin lines are used to *Clean* the CPN model when a is released. When transition *ReleaseRequest* occurs it uses *Clean(info)* to determine whether cleaning should start. If this is the case the token on place *Clean* changes to *(clean,n)* where *n* denotes the number of tokens to be cleaned, i.e., removed from place *CallControlApplication*.

The actual cleaning is done by a transition on page *Clean*, which is shown in Fig. 8.7. It is the subpage of the *Clean* transition in Fig. 8.5. The variable *infoMS* is a multi-set variable. This means that it is bound to a multi-set of *Info* values. The guard *Checks* that the size of *infoMS* is *n* and that all tokens belong to a certain subset of *Info*.

Now let us consider the leftmost transition in Fig. 8.6 in more detail. When transition *ConnectIndication* occurs, the state of the *SendingInterface* changes from *idle* to *callPres*. The *CallControlApplication* receives a *CallInfo* token – representing the information necessary to setup a call. The *Cstate* in the *Clean* token changes from *init* to *normal*, while the *Count* is incremented by one (since the *CallInfo* token is of a kind which has to be cleaned). The remaining transitions work in a similar way.

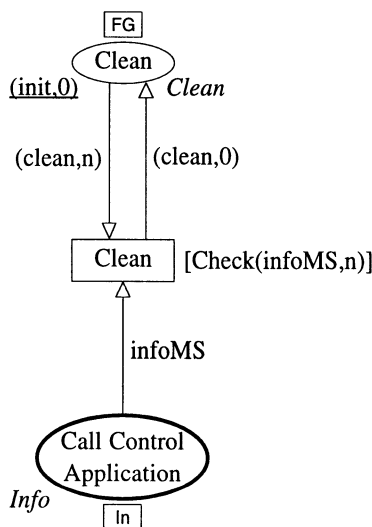


Fig. 8.7. CPN page for *Cleaning*

The *init* state of *Clean* is never restored and hence our model can only process one call. This is sufficient because calls never interact, except for the competition for resources. In practice, the gateway distinguishes between the different calls by using a call reference (in a similar way as described in Chap. 9).

Once the CPN model of the gateway specification was completed, it was validated by means of simulation. Interactive simulation is very useful in the early design stages, since it allows the designer to test the basic ideas behind the design. However, the model is too complex to be validated by simulation alone, and hence we also used occurrence graph analysis.

The occurrence graph is small. It contains only 365 nodes and 860 arcs. We first showed that there were no undesired dead markings and that the gateway was able to establish a call in the expected way. To perform these tests we used the standard occurrence graph queries and we also wrote a few simple system-dependent queries.

When errors were found, they were rather straightforward to correct. Here it turned out to be useful that the CPN tools allow the user to load the marking of interesting occurrence graph nodes into the simulator. In this way it is easy to investigate a marking and its enabled transitions. As an example, we can find out how a dead marking was reached.

Once the service specification was completed, it was time to consider refinement of the gateway specification. This involves designing an architecture that will provide the functionality necessary to fulfil the specification. The proposed functional architecture is shown in Fig. 8.8, which is actually the most abstract page in our refined CPN model. All transitions are substitution transitions. We see that the *Call Control Application* has been segmented into three functional blocks. *Sending Call Control* and *Receiving Call Control* are responsible for call setup and clearing within their respective interfaces. This is achieved by utilising the signalling protocols of the individual networks to transfer the necessary information. *Gateway Call Control* is responsible for handling the interworking between the other two call controls. It interacts with the other processes within the *Gateway Call Control Application* and with the *Service Handler*.

Place *Gateway Call Control IO* (in the centre of Fig. 8.8) represents the communication mechanism within the gateway. It contains *MIP* messages, which is a shorthand for Medium Independent Protocol messages. There is no queuing, and hence MIP message can be consumed, by the relevant process, in any order and at any time. We could model a FIFO queue for each process, but this is considered to be an implementation issue to be added at a later refinement stage.

Place *Node Location* represents a process that provides routing information. The three *Resource Management* places represent processes that are responsible for the monitoring of computing resources, call identifiers, and bandwidth. These processes are queried during call establishment to check whether there are adequate resources for the call. This is especially complex for the *Gateway Resource Management* process, as it must cope with issues related to radio propagation which require complex ionospheric prediction algorithms. However, such details are not included in our CPN model.

As mentioned before, our CPN model only handles a single call. This is sufficient, since calls do not interact with each other, except for the competition for resources. This competition occurs within the three *Resource Management* processes and within the *Service Handler*. In a more general model, there would be a *Sending Call Control* process, a *Gateway Call Control* process, and a *Receiving Call Control* process per call, while the *Resource Management* processes would be shared by all calls. In the event of two call entities requesting a scarce resource, the responsible resource manager will make a choice and allocate the resource to one entity. We model this by introducing a non-deterministic choice to determine the availability of the resources required to establish a call.

Next let us consider the colour set declarations of the refined CPN model (shown on the next page). Now there are four different kind of interfaces, and each of these has its own set of states (in addition to the general interface states represented by *Istate*). The nomenclature of *Sstate*, *Rstate*, and *Gstate* are similar to the one used in Q.931, but modified to suit the gateway. Idle states are represented by *S0*, *R0* and *G0*. Active states are *S10*, *R10*, and *G10*. Release states after an established call have numbers 11 and 12 in them. All other states are associated with the call establishment procedures, both successful and unsuccessful. *Pstate* represents the states of the processes in *Node Location*, the three *Resource Management* places, and the *Service Handler Interface*.

The *Count* colour set is the same as in our first model, and it is used in a similar way, i.e., for bookkeeping related to token cleaning. The *Addr* colour set represents addresses. It denotes the source and destination interface of a message.

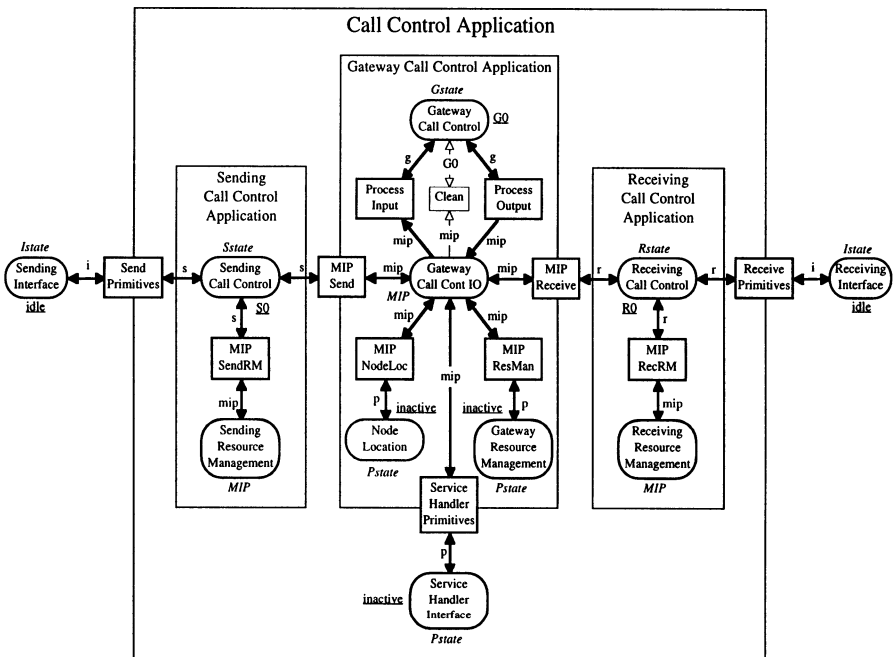


Fig. 8.8. Most abstract CPN view of the refined *Call Control Application*

Finally, *MIP* describes all the different kinds of MIP messages. The message structure was defined using the ASN.1 language, which is a standard used in telecommunications. The messages have fields associated with them, in the same way as a real protocol message contains information fields. Most messages have an *Addr* field to identify the source and destination. Others have a *Reply* or a *Result* field. The latter identifies a source, a destination, and a *Response*.

```

color Interface = with send | receive | gateway | process;
color Istate =with idle | callInc | callPres | callRec | callDel | callAct;
color Sstate = with S0|S1|S2|S3|S4|S5|S6|S7|S7a|S10|
                S11|S12|S14|S15;
color Rstate = with R0|R0a|R1|R2|R3|R4|R5|R6|R6a|R7|
                R8|R10|R11|R12;
color Gstate = with G0|G1|G2|G3|G3a|G4|G4a|G4b|G5|G5a|G6|
                G7|G8|G8a|G9|G10|G11|G12|G13|G14|G15;
color Pstate = with inactive | processing | active | halted;
color Count = int with 0..10;
color Addr = product Interface * Interface;
color Reply = with ok | nok;
color Response = with success | failure | redirect;
color Result = product Interface * Interface * Response;
color MIP =union CallSetup : Addr + CallConnect : Addr +
                ReleaseCall : Addr + Redirection : Addr + Alert : Addr +
                AddrCheckReq : Addr + AddrCheckResp : Result +
                ResQuery : Addr + ResResp : Result + ResReleaseReq : Addr +
                Configure : Addr + ConfigResp : Result + ServRelease : Addr +
                StopServ : Addr + NetStatusEnq + NetStatus : Reply +
                NetLoadEnq + NetLoad : Reply + CRAllocate +
                CRReply : Reply + ResourceRel;

```

Figure 8.9 shows the subpage for *Send Primitives*. It models the interaction between the *Sending Interface* and the *Sending Call Control*. The lower part is identical to Fig. 8.3 and the lower part of Fig. 8.6. However, the additional conditions, imposed by the interaction with the *Sending Call Control*, may change the behaviour of the interface. It is part of the subsequent validation to investigate this and, if necessary, modify the behaviour of the *Sending Call Control* in such a way that the interface specification in Fig. 8.2 is met. Place *One* guarantees that only one call is made. The reason for this has already been explained.

Figure 8.10 shows the subpage for *MIP Send*. It models the sending and receiving of MIP messages to and from the *Gateway Call Control IO*.

When the refined gateway specification was completed, it was validated by means of simulation. When this was done, we wanted to compare the behaviour

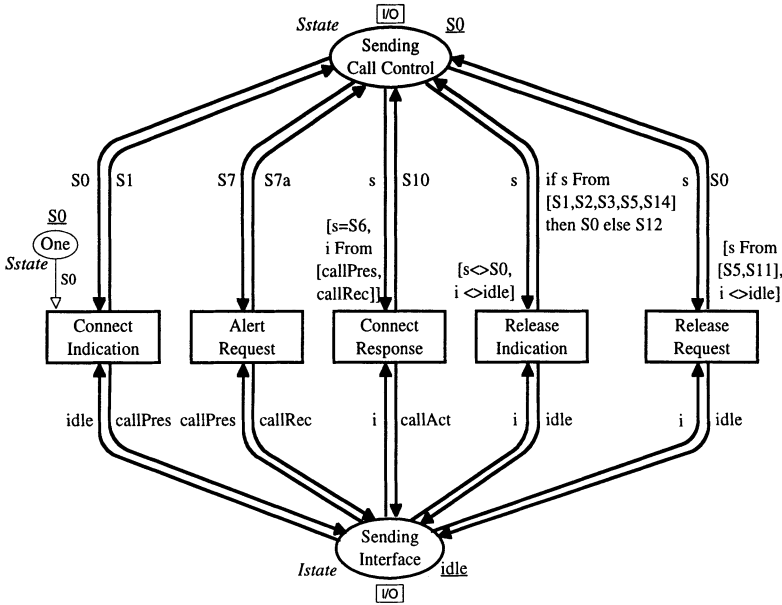


Fig. 8.9. Refined CPN page for *Send Primitives*

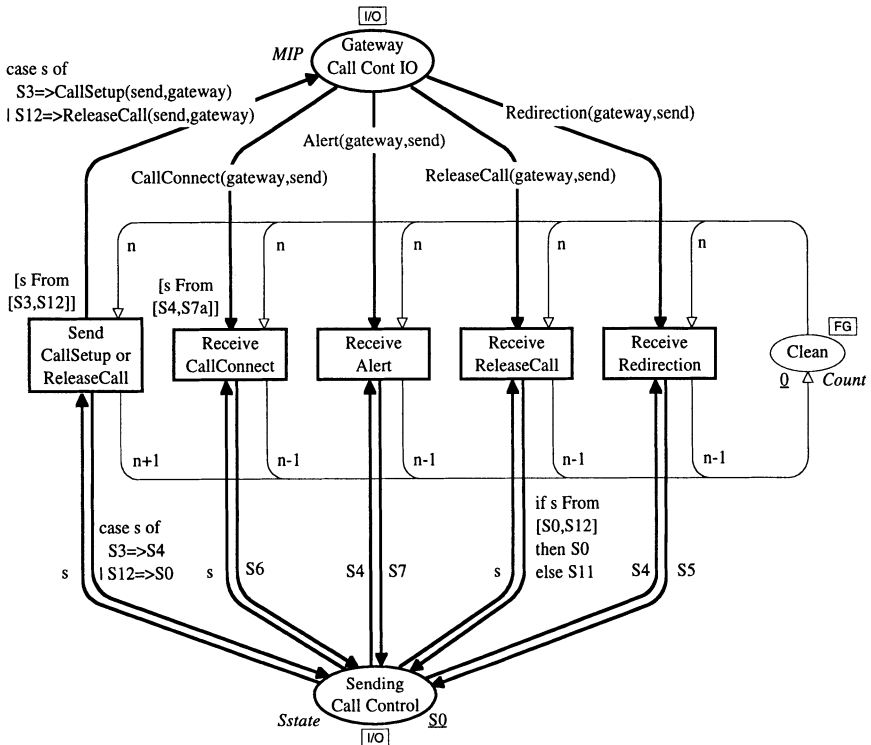


Fig. 8.10. Refined CPN page for *MIP Send*

of the refined specification with the original specification. The two CPN models of the gateway are at different levels of abstraction. However, the interfaces to the *CallControlApplication* are common, and hence we can compare the languages which the interface primitives determine.

To perform the language comparison, we used three different tools. The occurrence graphs were constructed in the CPN tools. Then they were transferred to the Protean tool [7], in which they were reduced by ignoring the cleaning transitions. Finally, the reduced occurrence graphs were moved to the Ara tool [53], which is capable of making an automatic comparison of the languages. In this way it was shown, after a number of iterations, that the refined specification has the same language as the original specification.

One reason for using language comparison is that it suits the “black box” approach taken by the gateway designers. Moreover, the necessary tools were available. Language equivalence does not preserve deadlock properties. However, this is not a big problem, since deadlocks can be easily found and removed by means of the occurrence graph queries in the CPN tools.

8.3 Conclusions for Communications Gateway Project

The use of CP-nets and the CPN tools within the specification stage of the gateway has been successful. The fact that the first attempts to refine the specification did not meet the original gateway specification shows how easy it is to make mistakes when specifying the behaviour of systems.

The readability of our specifications is up to the individual to decide, but as long as efforts are made to present the CP-nets in a simple and readable form, the CPN language should be able to gain wide acceptance.

It would be useful to implement language reduction/comparison algorithms directly in the CPN tools. This would remove the necessity of shifting between different tools and hence decrease the turnaround time. However, we will probably always have a variety of tools, offering different kinds of analytic capabilities. Hence, it would also be useful to develop a standard transfer format for CP-nets and for occurrence graphs.

During the course of modelling, many different modelling options were explored ranging from representing primitives as tokens to having more complex MIP message structures. The models that were eventually obtained are the result of many hours of trial and error to produce a readable and understandable specification. At the end, we decided to represent as much of the behaviour as possible in the net structure, and this meant that we use only a limited number of ML functions, which are all quite simple.

After several iterations of editing, simulation, and occurrence graph analysis, the two CPN models were proven to have the same languages, i.e., the same sequences of interface primitives. During this work, we removed many minor errors and we also located three more fundamental modelling errors.

The first modelling error was caused by an inconsistency between the original specification and the refined specification. In the original specification, the service handler is issued configuration primitives before the receiving interface is informed of an impending call setup. In contrast to this, the first versions of the refined specification informed the receiving interface before configuring the service handler. This inconsistency was due to the fact that in the early design stages the preferred option was not known. Both approaches successfully establish a connection between the involved interfaces, and the difference was not found until the systematic language comparison was performed.

The second modelling error was of a similar kind. It was caused by having different release procedures in the two models. Call release is the most complex part of providing the gateway service, because it can be requested in any state (even while a release process is in progress). In this part of the model we found several errors. One example is that the sending interface was released without releasing the receiving interface. Another is that the node location process was left active after the call release. One complicated issue was the arresting of a call setup that propagates through the gateway while a release primitive occurs at the sending interface. To achieve this, call release procedures had to inform all interfaces of a release, even if they had not been set up as yet. This disabled the receiving interface so that no further call setup actions could occur.

The third modelling error was more subtle. When the original specification was created, the main aim was to produce a simple service specification that met the requirements. At first glance, the model seemed to operate correctly. However, when the refined specification was completed and the language comparison began, it became apparent that there was no way that the refined model could be built to meet the original specification. The modelling processes were blamed at first, but the real reason soon appeared. In the original specification, the *Call Control Application* was directly connected to all interfaces. This allows state changes at one interface to be instantly observed at the other interfaces. However, in the refined model, the *Gateway Call Control Application* is isolated from the *Sending Interface* by the *Sending Call Control Application* and from the *Receiving Interface* by the *Receiving Call Control Application*. Hence instant information sharing is no longer possible and this implies that the original specification cannot be met. To remedy the problem a new transition was added to the *Receiving Interface* of the original service specification. It represents a call rejection within the *Call Control Application* independent of all interfaces.

The errors described above are unlikely to have been found without a systematic verification. We located a number of errors in the refined specification, but sometimes it also turned out that the original specification had to be modified. The errors were not due to the use of CP-nets, but are inherent to the development methodology that we applied. It is hard to know the implications of decisions made at early stages of the design process, but nevertheless it is necessary to delay many kinds of implementation decisions as long as possible. The best way to solve this type of problem is to integrate the validation process with the design process. In this way it is possible to validate the most basic parts of a de-

sign before too much complexity is added. Errors can be removed and valuable insight is gained – to be used in the remaining part of the design. When a refined specification has been proven to meet the original specification, additions can be made to the refined specification. In this way, the design and specification process proceeds iteratively, yielding a sequence of more and more detailed specifications. We may add new processes, for example, and then check whether they interfere with the basic call setup and clearing functions.

Chapter 9

BRI Protocol in ISDN Networks

This chapter describes a project accomplished by *Peter Huber and Valerio O. Pinci, Meta Software Corporation, Cambridge MA, USA, in cooperation with a large telecommunications company.* The chapter is based upon the material presented in [32]. A brief presentation of the project can be found in Sect. 7.1 of Vol. 1. The project was conducted in 1990.

With a large telephone company an experiment was made to point out how current work practices in the development of protocol software may be improved. By means of CP-nets and the CPN tools, we built a formal, executable specification of a standard network protocol, which substituted for a less formal approach using SDL diagrams (augmented by extensive, informal textual descriptions).

The protocol considered defines the procedures required for establishing, maintaining, and clearing phone connections at the Integrated Services Digital Network (ISDN) Basic Rate Interface (BRI). We discuss aspects of the modelling process and present the applied strategies for constructing and testing the actual CPN model. We also sketch how the CPN model was extended to handle the supplementary hold service modelled in Chap. 6.

The project shows that the current work practices in the telecommunications area may be improved by building graphical, directly executable CPN models, which can be validated by means of simulation and verified by means of occurrence graphs. The final CPN model was presented to engineers at the participating telecommunications company. This was done by making manual simulations of typical occurrence sequences followed by automatic simulations with animated application-oriented graphical feedback. According to the engineers, who all had wide experience with modelling and analysis of telephone systems, the CPN model provided the most detailed executable model they had seen for this kind of protocol.

Section 9.1 contains an introduction to the BRI protocol. We also describe the basics of the SDL language which is widely used in the telecommunications area. Section 9.2 presents the CPN model of the BRI protocol. Finally, Sect. 9.3 presents a number of findings and conclusions for the project.

9.1 Introduction to BRI Protocol

In these years there is a large amount of interest in the ISDN area. Most telecommunication equipment manufacturers are designing and implementing products for use at the so-called user-endpoints. Such products must comply with the standard put forward in the specific ISDN protocols. In an ever-more competitive market, it has become important to use new, efficient approaches for the design, development, and production of new, high-quality, sophisticated telephone products. Hence there is great interest in ways of reducing the turnaround time for new products, e.g., by improving the design methods.

The telecommunications company with which we undertook this project was in particular interested in new methods to define, simulate, and verify system features during the traditional requirement and specifications development phases. The aim of the project was to make, in a short time frame, a feasibility study to test and demonstrate how to develop and simulate a model of an ISDN protocol using CP-nets.

Together with the telecommunications company we singled out a reasonable subset of the ISDN protocol: the basic voice services of the **Basic Rate Interface (BRI)**. The BRI protocol deals with the user/network interface for establishing, maintaining, and clearing voice connections (i.e., regular telephone calls). Other ISDN protocols deal with data connections. Although the basic

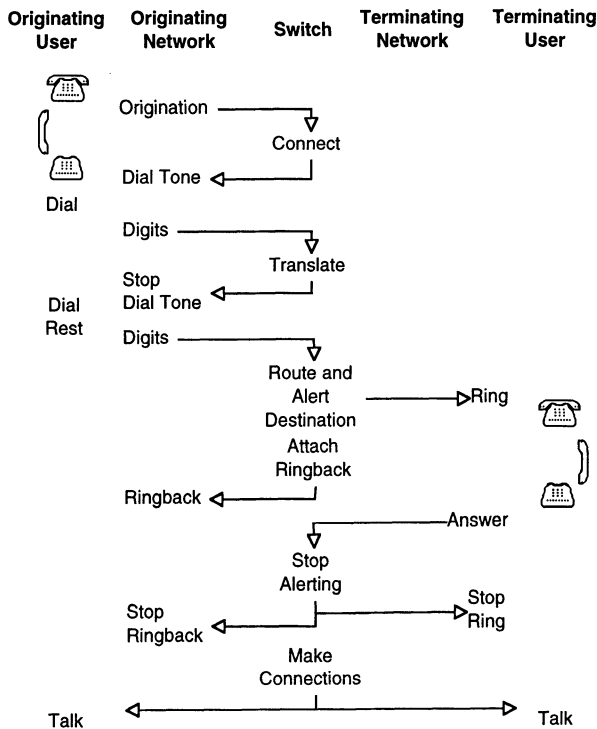


Fig. 9.1. Typical message sequence for a call set-up

voice protocol in itself seemed fairly extensive, it was pointed out to us – by the practitioners – that the protocol is rather elementary and would not necessarily give a full proof of our case. However, it gave us a sufficiently complex specification to start with. Later, we also modelled the hold service, i.e., the facilities for putting a call on hold. It is one of the supplementary services considered in Chap. 6.

The chosen protocol supports the procedures for establishing regular telephone connections. Figure 9.1 depicts, in a deliberately fuzzy way, a typical message sequence for a call set-up. The ISDN BRI specification governs only the message exchange between one telephone (user) and its network at the network layer. Hence for the depicted scenario, the protocol is applied in two locations: (a) between the originating phone and its network and (b) between the terminating phone and its network. In particular the protocol does not cover the switching network between the originating and terminating sides.

As basis for our modelling work, we were given the existing specification [2]. It is made by means of **SDL** (Specification and Definition Language) [16], which is widely used in the telecommunications area. The specification has two parts. The first part contains a number of SDL diagrams and their supplementary text. The second part describes the data contents of protocol messages.

The set of SDL diagrams focuses on the procedures in the telephone switching system for the logical call processing control flow and related actions. We were given 16 SDL diagrams for the user side (originating and terminating side combined) and 12 SDL diagrams for the network side (originating and terminating side combined). Figures 9.2 and 9.3 show two typical SDL diagrams. The small typographical differences are not intentional but probably due to different authors. Below we explain each of the SDL diagrams in detail together with the necessary SDL terminology. For a more detailed description of SDL see [16].

The SDL page in Fig. 9.2 represents a functional block, a state-machine-like submodel, of the user side **process**. A submodel consists of **SDL states** and **SDL transitions**. The top node of Fig. 9.2 represents a state (inscribed with its

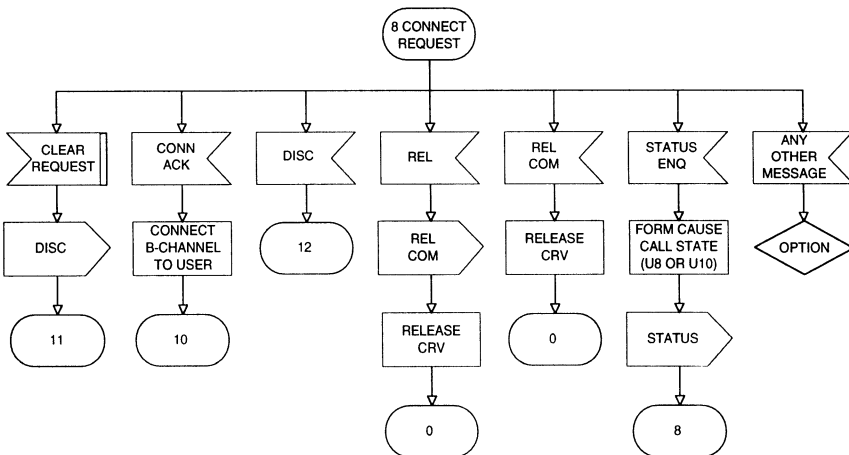


Fig. 9.2. Typical SDL page for the user side

number and name), whereas the seven nodes in the second row are SDL transitions. A process in a given state waits for a trigger for one of the SDL transitions. The processes in an SDL model communicate through infinite FIFO queues: each process has its own queue of incoming messages not yet processed. The messages are often referred to as **signals**, and their kind is distinguished by an abbreviated, capitalised name. The name of the signal recognised by a given SDL transition node is written inside the node. Such an SDL transition node is often referred to as an **input symbol**. In Fig. 9.2, the leftmost input symbol (with the double vertical lines) indicates handling of an internal request signal from the user, i.e., from the next layer in the user protocol. The shape of the six rightmost input symbols indicate handling of messages from the network side.

Below each input symbol there is a vertical **string** of SDL symbols. It contains a number of SDL transitions and ends with an SDL state. The transitions represent actions which will be executed when the first message in the queue matches the input symbol in question. When such a match occurs, the signal is removed from the queue, the actions are executed, and the process changes to the state represented by the SDL state at the end of the string. In a string, several different symbols may be used. The shape of SDL symbols are summarised in Fig. 9.4.

In Fig. 9.2, the leftmost vertical string tells us that upon receipt of a *Clear Request* signal from the user side, the process will send a *Disconnect* message to the network side – represented by an **output symbol**. The process will change

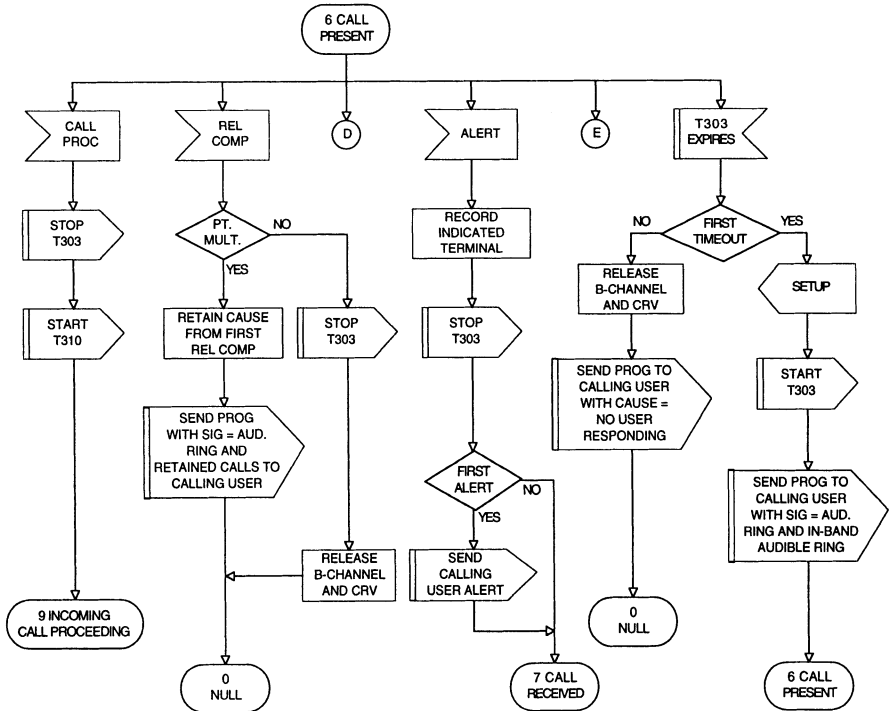


Fig. 9.3. Typical SDL page for the network side

from state 8 to state 11. Like input symbols, output symbols carry the signal name inside them. The second vertical string shows that upon receipt of a *Connect Acknowledgement* signal, from the network side, a specified action is performed and the process changes to state 10. The rectangular **action symbol** contain a short text describing the action. A typical action is to calculate parameters for output signals or to control B-channels and call references. Both of these are used to identify a call between a network and a user.

It is an established practice to organise SDL diagrams with a state node at the top and all possible state changes fanning out below it. The diagram in Fig. 9.2 has to the very right an input symbol indicating that the result of any other signal from the network side is left open for the implementation. Normally SDL diagrams use the convention that signals not modelled explicitly are simply consumed without causing a state change. For our pilot project, we have not described such signals at all, but they could easily have been modelled as well. The diamond-shaped node is an if-then-else **decision symbol**.

The SDL diagram in Fig. 9.3 represents the actions that can take place when the network side is in state 6. The network-side diagram is very similar to the user-side diagram discussed above. The input and output symbols are the “complements” of the user-side symbols – indicating that this is the other end of the interface. Again, the symbols with double bars indicate internal signals.

The leftmost string in Fig. 9.3 is triggered by a *Call Proceeding* signal from the user side. This results in stopping timer *T303*, starting timer *T310*, and changing from state 6 to state 9. The rightmost string waits for the internal action indicating a time-out of timer *T303*. If it is the first time-out, a *Set-up* message is sent to the user side, timer *T303* is started, an internal request *PROG* is sent to the user, and state 6 is entered. If it is the second time-out, an internal action is performed, an internal request *PROG* is sent to the user, and state 0 is entered. The two small circles, D and E, indicate off-page connections. This con-

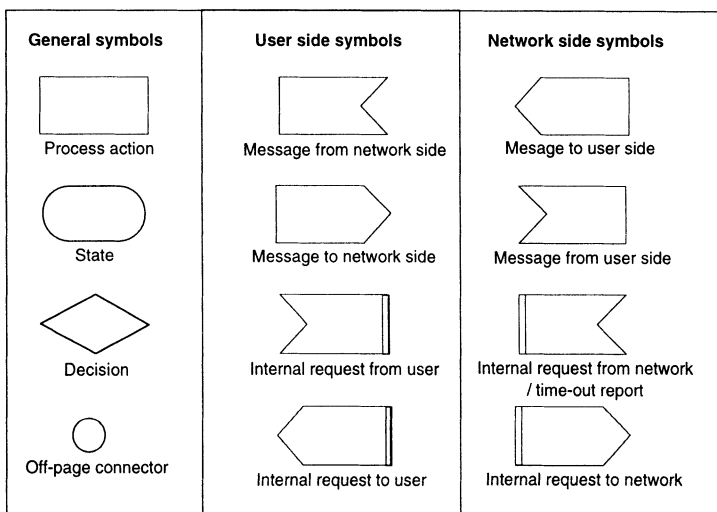


Fig. 9.4. SDL symbols (from [16])

struct is mainly used due to space limitations and does not imply the existence of a well-defined submodel concept as such.

Together with the SDL diagrams, [2] contains 25 pages of prose, which describe what is already represented in the SDL diagrams together with more detailed explanation and listing of special cases and exceptions. As examples of supplementary or redundant text, we have descriptions like: “The user initiates call establishment by transferring a *Setup* message across the user-network interface” or “If the channel the user has requested is appropriate and available, the network shall send the user a *CallProceeding* message with that channel identification specified exclusive”. The second part of [2] describes the bit contents of message data fields (25 pages) and holds a listing of the data contents of the individual messages exchanged between the user side and the network side (40 pages).

We also received the specification of four supplementary voice services [3], including the hold service. This material was in the same form as for the basic voice services discussed above (12 SDL diagrams for the user side and 20 pages of supplementary text).

9.2 CPN Model of BRI Protocol

The CPN model of the ISDN BRI specification was built and debugged in only 16 days. This includes meetings with people from the telephone company. The work was done by a CPN expert – who was a novice in the telecommunication domain. The supplementary hold service was modelled in one extra day. As we shall see below, the CPN model is fairly extensive, with 43 pages organised in four hierarchical levels.

The graphical layout of the CPN model was done in such a way that there is a direct graphical correspondence between the SDL specification and the CPN model. Each SDL page was represented by a CPN page. Each SDL state node was mapped into a CPN place while each vertical string of the SDL diagram was mapped into a single CPN transition followed by a CPN place representing the new state. Instead, we could have represented a string as a sequence of CPN places and transitions, but we preferred the more compact representation. To make the CPN model more readable for SDL people, each CPN transition were given the same shape as the input symbol of the corresponding SDL string. The strategy of mapping one string into one CPN transition cannot be taken if the order of several inputs and outputs is important. For this reason, one of the SDL strings on the network side had to be modelled by two CPN transitions with an intermediate CPN place. When this was discussed with the implementers, we learnt that this “two-stage” behaviour only had been realised during the implementation. There it was handled by introducing new substates – but the SDL specification was never updated.

Figure 9.5 shows the CPN page corresponding to the user-side SDL diagram in Fig. 9.2. The six CPN transitions can easily be identified with the corresponding strings of the SDL diagram. Notice how the shape of the CPN transi-

tions matches the SDL input signals. The rightmost input symbol from Fig. 9.2 has been left out, since it was also left unspecified in [2]. The user state places have been given the number of the state prefixed by “U”, e.g., *U8*. The architecture of the system has been represented directly by means of three port places. The first of these, *InternalUserReq*, represents internal user messages while the two others, *NetworkToUser* and *UserToNetwork*, represents messages from the network to the user and vice versa. In this way we achieved an explicit representation of the message exchange in the system. All arcs surrounding *U8* have identical arc expressions. Hence only one of these is shown. A similar convention is used for the arcs surrounding *NetworkToUser* (and in Figs. 9.6 and 9.15).

Figure 9.6 shows the CPN page corresponding to the network side SDL diagram in Fig. 9.3. This page has similar characteristics as the CPN page in Fig. 9.5. Network state places are named with the state number prefixed by “N”. Besides *NetworkToUser* and *UserToNetwork*, we have now two other port places called *NetFromNet* and *NetToNet*. Additional CPN places represent data local to the network side: the state of timers *T303* and *T310* and the state of the B-channels, *NBC*. The latter are administered by the network side. The two off-page connectors in Fig. 9.4 are modelled by the two substitution transitions “D” and “E”. The subpages of these substitution transitions are the CPN pages that correspond to the SDL diagrams to which D and E lead.

As indicated by Figs. 9.5 and 9.6, each user/network page has a set of places that represent user/network states. The fusion set construct came in handy to “glue together” all places representing a given state. In this way, we ended up with 12 global fusion sets for the user states and 12 for the network states.

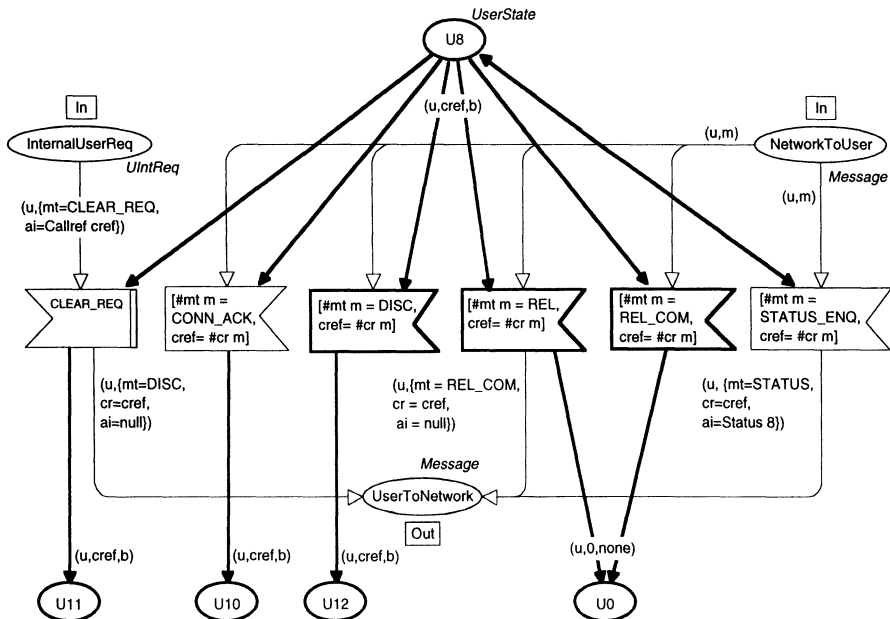


Fig. 9.5. CPN page corresponding to SDL diagram in Fig. 9.2

Analogously, global fusion sets were used to unify places representing timers and the state of B-channels.

For this model, we have hidden the fusion tags and adapted the simple convention that places with the same name are either in a fusion set or related as ports and sockets. The latter case can be distinguished from the former by the fact that ports always have either an In, an Out, or an I/O tag. We have also hidden the HS tags used to identify substitution transitions. Instead we have shown substitution transitions with a thicker border line. The relationship between substitution transitions and subpages can be seen from the names of the substitution transitions and from the page hierarchy in Fig. 9.16.

Figure 9.7 shows the most abstract view of the CPN model. It tells us that the system consists of a *Users* side and a *Networks* side connected via two communication channels *UserToNetwork* and *NetworkToUser*. At this abstract level only the very essence of the BRI protocol is captured: the exchange of messages between users and networks. The arcs of Fig. 9.7 do not carry any inscriptions, but due to the port assignments, tokens representing messages will appear on the places *UserToNetwork* and *NetworkToUser* during simulations – when transitions occur in the subpages of *Users* and *Networks*.

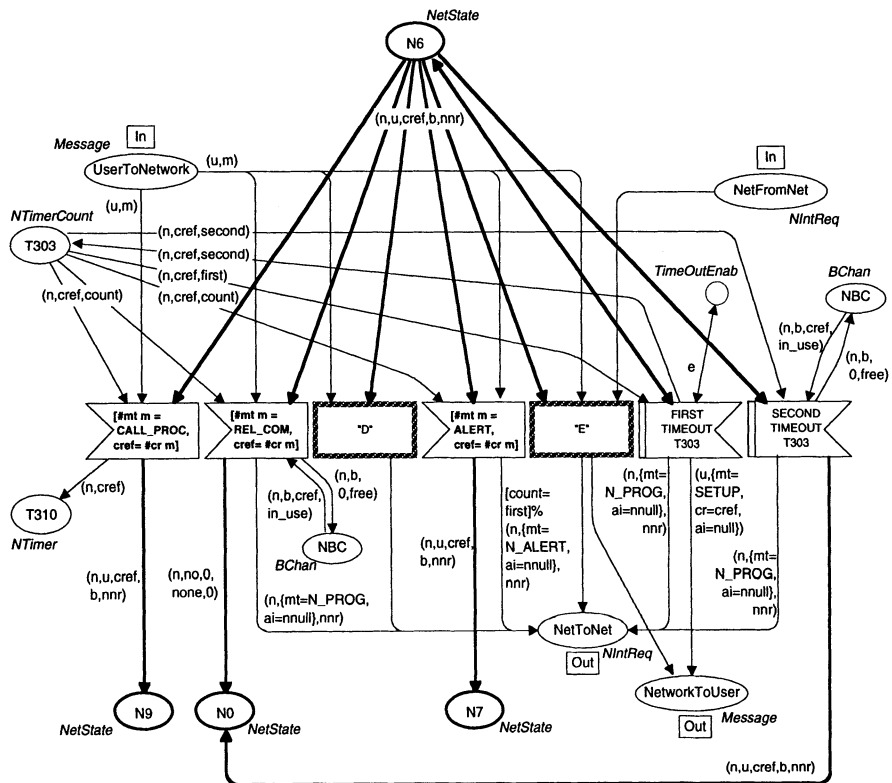


Fig. 9.6. CPN page corresponding to SDL diagram in Fig. 9.3

The subpage for the *Users* transition is shown in Fig. 9.8. The page provides an overview of the user side and interconnects the CPN pages for the individual user states. The SDL specification did not contain any overview diagrams like Figs. 9.7 and 9.8. The substitution transition *U8* in Fig. 9.8 has the CPN page in Fig. 9.5 as subpage. The subpage shows all the possible state changes that can happen from user state 8. Analogously, each of the other 11 substitution transitions in Fig. 9.8 has a subpage that describes the actions that can take place in the corresponding user state. These pages have a structure that is very similar to Fig. 9.5, but the details are of course slightly different. The substitution transition *UREQ* in the upper left corner of Fig. 9.8 will be discussed later.

In Fig. 9.9 we have listed some of the colour sets declared for the user side of the model. Although the protocol only deals with one user and one network, we declare a set of *Users* containing six different telephones. The purpose will become clear when we discuss the modelling of the environment. *Call References* are used to identify the individual connections. The highest bit tells whether the

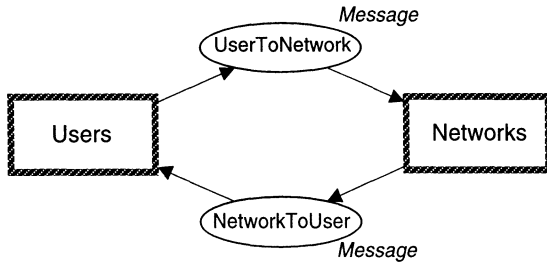


Fig. 9.7. Most abstract CPN view of BRI protocol

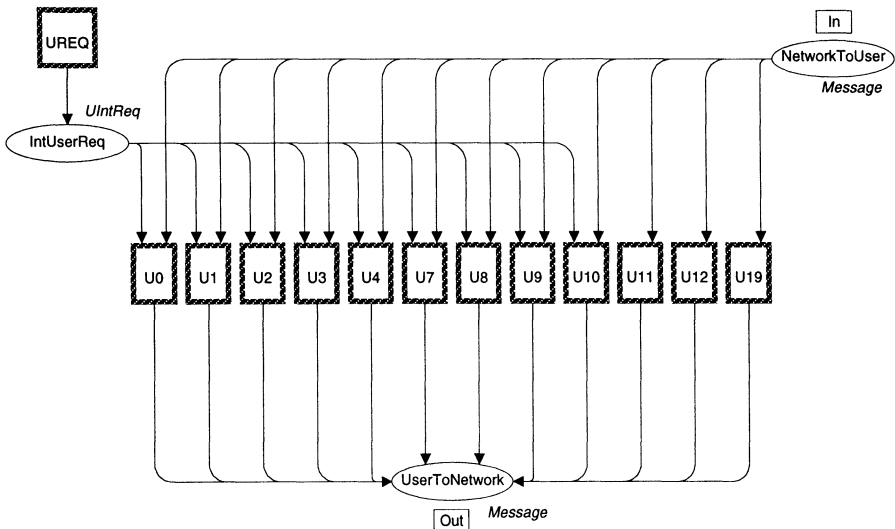


Fig. 9.8. User side overview page

call originates from the user side or from the network side. *B-channels* are the physical channels on which the connections are established. They carry the names *b1* and *b2*. The state of a user is described by a triple which identifies the *User*, *CallReference*, and *B-channel*.

The colour set *MessageType* enumerates the kind of messages exchanged between the user and network. The two next colour sets declare the data contents for *Status* and *Setup* messages, respectively. The union colour set *MData* specifies – like a variant record in programming languages – the tags and corresponding types for the data part of the different kinds of messages. For example, if the tag is *Setup*, the data contents is of type *Setupdata*. The majority of messages uses the *MData* colour *null* to indicate that they carry no additional data.

We declare a *MessageRecord* to consists of three labelled fields containing a *CallReference*, a *MessageType*, and an *MData* value, respectively. Finally, we declare a *Message* to be a pair where the first component is a *User* while the second is a *MessageRecord*.

With these colour set declarations in mind, it becomes rather straightforward to interpret the behaviour of the individual CPN transitions. To illustrate this, we take a look at the three rightmost transitions in Fig. 9.5. During our discussion it may help to review the corresponding SDL diagram in Fig. 9.2. The fourth transition of Fig. 9.5 is executed when a *RELease* message is received from the network. The execution produces a *RELease COMPLETED* message to the

```

color User = with u11 | u12 | u21 | u22 | u31 | u32 | no;
color CallRef = int with 0..255;          (* 1..127 for user *)
                                           (* 128..255 for net *)

color BChanName = with b1 | b2 | none;
color UserState = product User * CallRef * BChanName;
color MessageType = with ALERT | CALL_PROC | CONN | CONN_ACK |
                      DISC | INFO | PROG | REL | REL_COM | SETUP |
                      SETUP_ACK | STATUS | STATUS_ENQ;

color Statusdata = int;
color Setupdata = product BChanName * User;
color MData = union  Status:Statusdata +
                    Callproc:BChanName +
                    Setup:Setupdata +
                    Setuppack:BChanName +
                    null;

color MessageRec = record  cr: CallRef *
                          mt: MessageType *
                          ai: MData;

color Message = product User * MessageRec;
var u: User;          var b: BChanName;
var cref: CallRef;   var m: MessageRec;

```

Fig. 9.9. Excerpts of the user-side CPN declarations

network and the state of the user process changes from *U8* to *U0*. The guard inside the transition checks the *Message Type* and also checks that the *Call Reference* of the incoming message matches the one in the incoming user state token (the *#mt* and *#cr* operators extract the *mt*-field and the *cr*-field of the message record *m*, respectively). From the thick output arc we can see that the process releases its attachment to a *Call Reference* and a *B-channel*. The next transition shows a state change from *U8* to *U0* upon receipt of a *RELease COMPLETED* message from the network. As before, the *Call Reference* and the *B-channel* are released. The rightmost transition describes the handling of a *STATUS ENQUIRY* message from the network. This does not lead to any state change. The only action is to produce a *Status* message to the network telling that the present user state is *U8* (by means of the *ai*-field).

We did not model the FIFO nature of the message queues. However, this is easily done. A straightforward solution is to change the colour set *Message* to be a product of a *User* and a list of *Message Records*.

Figure 9.10 shows some of the colour set declarations for the network side. They are analogous to those of the user side. First we declare three networks. The next colour sets specify non-counting and counting timers (*NTimer* and *NTimerCount*) and B-channel states (*BChanState* and *BChan*). For modelling the inter-network communication we have declared identifiers (*NetNetRef*). The use of these identifiers will be explained below. Similar to the *UserState*, *NetState* keeps track of the network state. This is done by means of a 5-tuple. Finally, the last four colour sets describe the inter-network messages. This is done in a way similar to the messages in Fig. 9.9.

```

color Net = with n1 | n2 | n3;
color NTimer = product Net * CallRef;
color Count = with first | second;
color NTimerCount = product Net * CallRef * Count;
color BChanState = with free | reserv | in_use;
color BChan = product Net * BChanName *
                CallRef * BChanState;
color NetNetRef = int;
color NetState = product Net * User * CallRef *
                NetNetRef * BChanName;
color NIntReqType = with N_SETUP | N_CONN | N_CONN_ACK |
                N_DISC | N_PROG | N_ALERT;
color NIRData = union NSetup:User + nnull;
color NIntReqRec = record mt : NIntReqType * ai : NIRData;
color NIntReq = product Net * NIntReqRec * NetNetRef;

var n,nrec,nsend : Net;   var nnr : NetNetRef;
var nm : NIntReqRec;

```

Fig. 9.10. Excerpts of the network-side CPN declarations

As examples, let us now study the two outermost transitions in Fig. 9.6. As before, it may also be helpful to review the corresponding SDL diagram, i.e., Fig. 9.3. The leftmost transition awaits a *CALL PROceeding* message from the user side. When such a message is received the timer *T303* is stopped (regardless of the value of *count*). The timer *T310* is started and the network state changes from *N6* to *N9*. The rightmost transition models a time-out. It waits for a *T303* timer token with value *second* (indicating that it is the second time-out of the timer). When such a token is available the transition occurs. The corresponding B-channel is released, a message is sent to the other network, and the network state changes to *N0*.

The modelled protocol part only specifies the interface between a single user and a single network. However, we added mock-ups of the environment to achieve a simulation scenario in which several users and networks could participate – as already provided for in the colour set declarations of Figs. 9.9 and 9.10. Two extra pages were added to the diagram: a subnet for generation of user requests (Fig. 9.11) and a subnet for inter-network routing (Fig. 9.12). These subnets provide a simple model of the environment in which the protocol works.

The three transitions in Fig. 9.11 all work in a similar way. Each of them generates tokens representing request from the upper layer in the user protocol. The transitions do not have any input places or guards. Hence there is no restriction on the binding of the variables *u*, *cref*, and *called*. Additional places could have been added to elaborate the functioning of this page.

In an interactive simulation, the three transitions can be executed whenever a certain request is needed. The values of the variables are bound either manually or by means of a random number generator. However, for testing and automatic simulation, we wanted to control these transitions so that a reasonable number of requests are generated. Hence, we added code segments to control the token generation on this page, based on input files. With the file approach it was easy to force the simulation into certain calling patterns of particular interest. An alternative to the files would be to have places with lists of the desired requests.

In order to give a simple model of the inter-network communication, we have declared a set of special identifiers *NetNetRef* in Fig. 9.10. When a call is initiated, at the originating network, two tokens are added to the fusion place *NNR* in

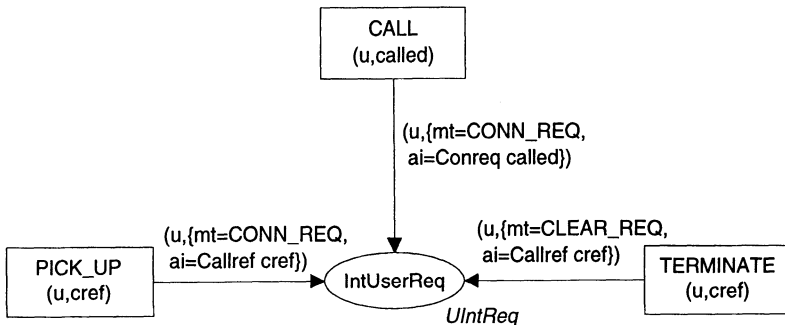


Fig. 9.11. CPN page for generation of user requests

Fig. 9.12. The two tokens represent the inter-network link. As an example, if $u32$ is calling $u12$, then a new *NetNetReference* is generated, say 77, and two tokens with colours $(n3,77,n1)$ and $(n1,77,n3)$ are put in *NNR*. During the simulation, the originating network will put messages for the terminating network in the place *NetToNet*. This will enable the *Routing* transition, which will move the message from place *NetToNet* to place *NetFromNet* and change the first component from n_{send} to n_{rec} (based on the information in *NNR*). The communication in the other direction is supported in a similar way using the other token in *NNR*.

With the constructed CPN model, we made a series of simulation runs. Input data was read from files and the progress of the simulation was observed by watching the token flow and the occurring transitions. As part of the project we also experimented with customised animation. By means of a few rather simple code segments we constructed a simple graphic display of the calling states for the six telephones and the three networks. A snapshot from a simulation run is shown in Fig. 9.13. The snapshot shows that $u11$ has established a local call to $u12$ (thick arrow), $u22$ is trying to call $u11$ but is receiving a busy signal, while $u32$ is in the process of making a connection to $u21$.

This kind of animation can now be obtained by the animation utility described in Sect. 1.3. However, that utility did not exist when our project took place, and hence we needed to do some amount of ML coding. It is straightforward to augment the display to show more technical details like B-channels and call references in use, the state and length of communication queues, and state of timers. Such additions should be designed together with the protocol experts – reflecting their understanding of the simulation model.

During the modelling process it was important for us to apply certain structured model building and testing strategies, similar to structured programming and testing techniques. We want to highlight the following points about our modelling. They made it possible to finish the rather large model in approximately two weeks.

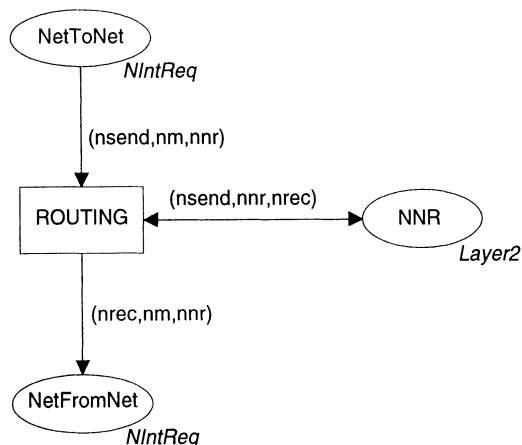


Fig. 9.12. CPN page for network-to-network communication

- The whole user side was laid out and equipped with textual inscriptions before the network side was modelled. For intermediate simulation a one-page mock-up of the network side was made, to respond to user messages for the basic, normal case message pattern.
- The first version of the user side only handled a small set of messages types, and messages were sent with no data contents. Based on that experience the colour sets were elaborated and finalised.
- When the first user page had reached a graphically pleasing state, it was easy to copy the graphics to get a skeleton for the other user pages.
- Even with the full model developed it was still fruitful to be able to test sub-models without having to isolate these in a separate model. This was done using the simulator facilities described in *selection of subdiagrams* in Sect. 6.2 of Vol. 1.
- For a given call several behavioural patterns exist. In order to test, in a structured way, we added enabling places to some of the transitions. As an example, the place right above the transition named *FirstTimeOut T303* (in Fig. 9.6) was used, during simulation, to enable/disable the possibility for time-out by adding/removing an e-token on the fly.

As mentioned at the end of Sect. 9.1, we also received material covering the supplementary hold service [3]. Protocol engineers consider this part to be considerably more challenging than the basic BRI protocol.

A call can be put on hold when it is in one of three user states: *U3*, *U4*, or *U10*. The hold state machine has four states, which are substates of the three regular user states. The SDL specification of the hold service contains $12 = 3 * 4$ additional user side SDL diagrams together with a significant amount of text. The diagrams for the network side were not given, but for this project we im-

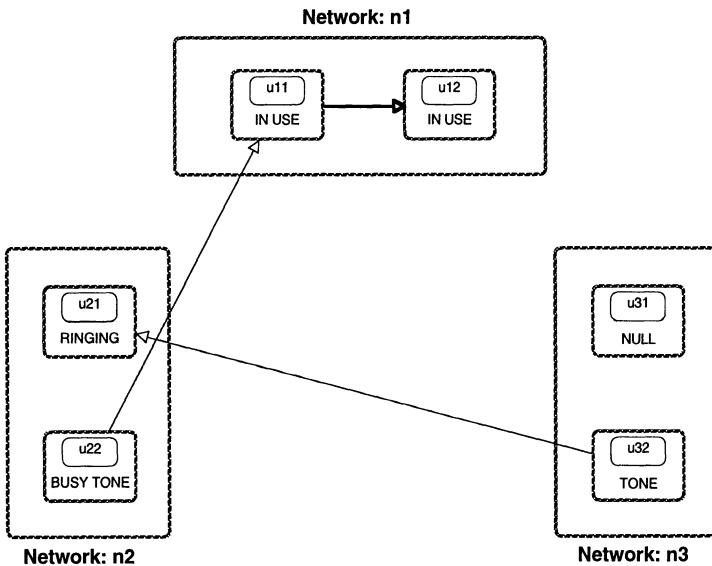


Fig. 9.13. Simple animation updated by transition code segments

plemented CPN transitions, which mirrored the functionality of the user side. The 12 SDL diagrams may not seem too bad, but remember that the hold service is just one of several supplementary services (e.g., call forwarding and conference call). This means that the SDL description of the full BRI protocol becomes rather large and hence harder and harder to work with.

We took the modelling of the hold service as a challenge. We could have made a straightforward mapping of the SDL diagrams into the same number of CPN pages. Instead we wanted to investigate how compactly it could be done using the abstraction mechanisms of CP-nets.

As shown in Fig. 9.14, we augmented the declarations slightly. A colour set for the hold substates was declared (*Substate*), and the *UserState* tuple (from Fig. 9.9) was given a fourth element to record the hold substate. Six additional message types were added to *MessageType* (from Fig. 9.9). In our CPN model all

```

color Substate = with ZERO | (* no substate *)
                  H30 | (* hold requested *)
                  H33 | (* reconnect requested *)
                  H36; (* call on hold *)
color UserState = product ..... * Substate;
color MessageType = with ..... HOLD | HOLD_ACK | HOLD_REJ |
                      RECONN | RECONN_ACK | RECONN_REJ;
var ss : Substate;
    
```

Fig. 9.14. Augmented user side declarations to handle the hold service

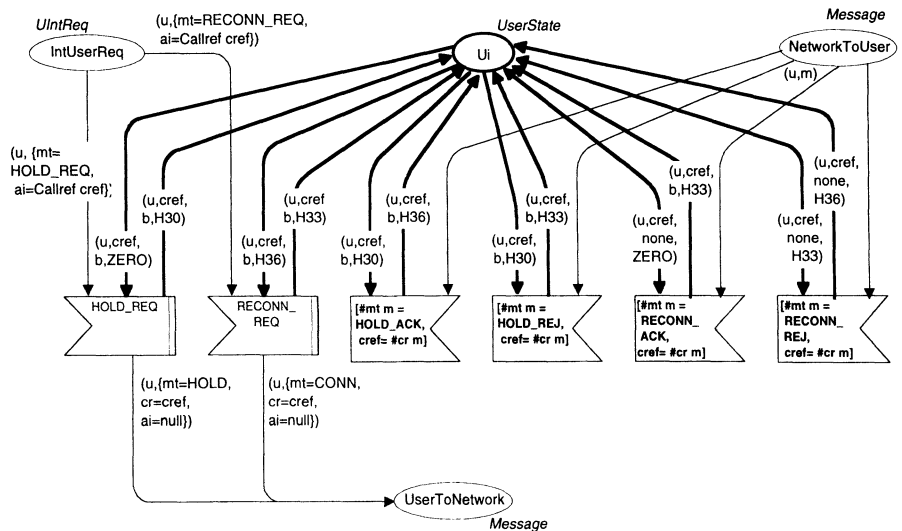


Fig. 9.15. CPN page to handle the hold substates on the user side

arc expressions for arcs to or from *UserState* places were changed from (\dots, \dots, \dots) to $(\dots, \dots, \dots, ss)$. This indicates that the signal handling of the basic BRI protocol does not change the hold substate component of the state token. Finally, two new message types for internal users request were added (not shown).

All the substate handling on the user side was put on a single new CPN page, shown in Fig. 9.15. The two first transitions handle the two new internal user requests and send corresponding messages to the network. Note that the user process remains in state U_i , but the substate value is changed to $H30$ and $H33$, respectively. The names for the hold substates are taken from [3]. Their intuitive interpretation is given in the comment following the declaration of *Substate* in Fig. 9.14. The four last transitions wait for hold related messages from the network and change the substate value accordingly.

The page hierarchy for the final CPN model is shown in Fig. 9.16. The new page, *U_Hold#45*, was included in the existing CPN model, by making the page a subpage of each of the three pages describing a user state in which a call on hold can be activated (i.e., the pages *Outgoing#15*, *Call_Del#16*, and *Active#7*). The port place U_i was related to the socket places $U3$, $U4$, and $U10$, respectively. The changes at the network side were very similar and again it was sufficient to add a single page, *N_Hold#44*.

By reviewing the other supplementary services described in [3], we recognised that all of them can be modelled in a similar, compact fashion. The modelling of the hold service highlights the value of reusable modules. Such a standard

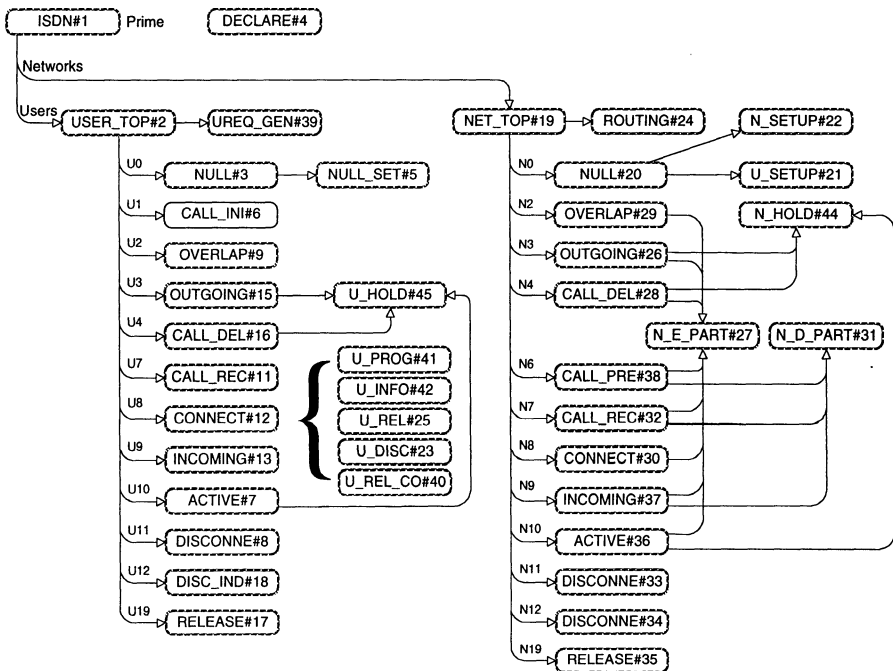


Fig. 9.16. Page hierarchy for BRI protocol

abstraction construct was not used in the SDL diagrams. We chose to consider the four hold states as substates of the ordinary user states. Hence, we represented the substates by a component of the *User* colour set instead of using new state places.

Inspired by the use of reusable modules for the hold service, we reviewed the entire BRI model to see whether there were other parts in which we could reuse common subnets. We now observed that several of the signals are handled exactly the same way in many different user states. For instance, the transition modelling the *RELease* signal in Fig. 9.5 has counterparts with identical inscriptions at most of the other user pages. Hence, we turned all of these transitions into substitution transitions (and removed their arc expressions and guards). All the *RELease* transitions use the same subpage which contains a single transition (identical to the *RELease* transition in Fig. 9.4). This change made it explicit that all these user states handle the release operation in the same way. Moreover, it becomes much easier to maintain the description of the release operation, since only one page has to be updated when changes are made. A similar modification was made for four other signals on the user side. This gives us the five pages *U_Prog#41...U_Rel_Co#40* next to the large bracket in Fig. 9.16. The bracket indicates that the five pages are subpages of most of the pages in *Null#3...Release#17*.

On the network side, the subpages for the D and E substitution transitions (in Fig. 9.6) could be reused in a well defined way on other state pages (see *N_D_Part#31* and *N_E_Part#27* in Fig. 9.16). Again, this way of structuring the model was not employed in the SDL diagrams we were given.

The page *Connect#12* was shown in Fig. 9.5, *Call_Pre#38* in Fig. 9.6, *ISDN#1* in Fig. 9.7, *User_Top#2* in Fig. 9.8, *Ureq_Gen#39* in Fig. 9.11, *Routing#24* in Fig. 9.12, and *U_Hold#45* in Fig. 9.15.

9.3 Conclusions for BRI Protocol Project

In this section, we compare the original SDL based specification from [2] and [3] with the CPN model. We also briefly discuss how the use of SDL and CPN may be integrated. Note that our discussion is based upon the concrete SDL specification that we received in 1990. Since then SDL has been improved in several different ways. This means that some of the shortcomings mentioned below no longer exist.

- The CPN model is *directly executable* and it is possible to instrument it and supply simulation data in a flexible way. It is not necessary to convert the CPN model to any other representation in order to simulate it.
- The use of *hierarchies* in CPN is much more elaborate than in the SDL specification. It seemed that the SDL designers primarily had used new pages when there was no more room on a page. In CP-nets there are major accomplishments obtained through a consistent use of hierarchies:

- unification of user and network states through place fusion is formally defined.
- readability is improved by adding pages with more abstract views of the system (e.g., Figs. 9.7 and 9.8). These pages are also useful during simulation to show behavioural abstractions and for fast browsing.
- substitution transitions can be applied to reuse behaviour that is shared by several components.
- the page hierarchy gives a good overview of the structure of the specification and can be used for fast browsing.
- The *elaborate graphics* supported by the CPN tools makes the model more readable. Places and arcs representing states and state changes are drawn with thick lines, while those representing messages are drawn with thinner lines. Most of the graphical symbols from the original SDL diagrams are retained.
- The use of *structured colour sets* (e.g., products, records, and unions) makes it possible for the CPN model to capture the detailed contents of messages without describing implementation details such as the bit-layout.
- The CPN model constitutes a *single compact model* embodying all the necessary detail such as behaviour, data contents of messages, and system architecture. In this way it is easier to achieve consistency.
- The *syntax check* of the CPN model eliminates many errors and inconsistencies.
- The *maintenance* of the final specification is more feasible and controlled due to its compactness. Much redundancy has been eliminated by the reuse of pages.
- The CPN model is *formal* and it can be *validated* by means of simulation and *verified* by means of occurrence graphs and place invariants. This means that errors and other shortcomings in the specification can be found before implementation.

The use of CPN models does not necessarily imply that the protocol designers should use CP-nets directly for specification of new and existing protocols. It might be a better idea to integrate SDL and CP-nets in such a way that an SDL specification can be translated automatically into a CPN model – which then can be completed with CPN inscriptions capturing those aspects of the behaviour for which SDL is not well suited. Another possibility is to add the net inscriptions directly to the SDL diagrams, before the automatic translation into a CPN model. An integration of SDL and CPN can be done similarly to the existing integration between SADT diagrams (also called IDEF diagrams) and CP-nets. For more details, see Sect. 14.1.

Chapter 10

VLSI Chip

This chapter describes a project accomplished by *Robert M. Shapiro, Meta Software Corporation, Cambridge MA, USA, in cooperation with a manufacturer of supercomputers*. The chapter is based upon the material presented in [52]. A brief presentation of the project can be found in Sect. 7.2 of Vol. 1. The project was conducted in 1990.

We describe how CP-nets and the CPN tools were used to specify and analyse the behavioural aspects of a hardware design at the register transfer level. We constructed a CPN model of a VLSI chip used in the most recent supercomputer developed by the company.

The chip manufacturer was interested in speeding up the design phase of the production cycle for new hardware. The designers specify a new chip by drawing a set of block diagrams, which each contains a set of nodes called blocks and the connections between them. Each block represents a functional unit with a specified input/output behaviour. A complex block may be described by means of a separate block diagram, which is related to the block in a way which is analogous to the relation between a substitution transition and its subpage in a CPN model. When the designers have finished a new chip, the block diagrams are translated, by a manual process, into a simulation program written in a special-purpose dialect of C. The simulation program is then executed on a large number of test data, typically 10 000–20 000, and the output is analysed to detect any malfunctions.

The company wanted to eliminate the programming step, which was both prone to error and time-consuming. It was hoped that the block diagrams drawn by the design engineer could serve as the basis for automatic generation of a CPN model. To achieve this, the graphics of the block diagramming technique would have to be formalised and some inscriptions added to provide a complete behavioural description.

Section 10.1 contains an introduction to VLSI design and validation. Section 10.2 presents the CPN model of the VLSI chip. Finally, Sect. 10.3 presents a number of findings and conclusions for the project.

10.1 Introduction to VLSI Chip

The purpose of the project was to investigate whether the use of CP-nets is able to speed up the design and validation of new VLSI chips at the register transfer level. We present the CPN model of an actual digital filter chip from a super-computer and discuss how this model was used to validate the logic of the chip design.

The basic idea was to replace the manual translation, from the block diagrams into the C program, with an automatic translation into a CPN model. It is important to understand that it is *not* the intention to stop using block diagrams. The designers will still specify the VLSI chip by means of block diagrams, and they will follow the simulation of the CPN model by watching the block diagrams. To support the new strategy three things are needed:

- The existing editor for the block diagrams must be modified, so that it gets a fixed syntax with a well-defined semantics.
- It must be possible to translate a set of block diagrams into a CPN model.
- The CPN simulator must be powerful enough to handle the rather complex VLSI designs, and efficient enough to make it possible to check the large number of test data.

Our project only dealt with the last two issues, which were considered to be the most difficult. It was shown that the block diagrams could be translated into CP-nets. This was done manually, but the translation process is rather straightforward, and we see no problems in implementing an automatic translation. The CP-net obtained only contains 15 pages, but during an execution there are almost 150 page instances. This is because many subpages are used several times. This is the case, e.g., for a subpage representing a 16-bit adder. The CP-net was simulated using the CPN simulator. When maximal graphical feedback was used, the simulation was slow – due to the many graphical objects which had to be updated in each step. However, when a more selective feedback was used, the speed became more reasonable.

The ultimate objective was to develop a design approach whereby the graphical description, with suitable inscriptions, would suffice both as a design medium and as a complete, directly executable specification of the behavioural properties.

To test the new approach, the chip manufacturer provided a register transfer level design of the VLSI chip. The design included a set of block diagrams for the chip, supplemented by a number of pages with textual descriptions. Additionally, we were given a C program simulating the behaviour of the design.

10.2 CPN Model of VLSI Chip

The CPN model of the VLSI chip has the page hierarchy shown in Fig. 10.1. From this it can be seen that we model a *VLSI System* with three identical *Filter Chips*. Each *Filter Chip* has six different parts: *Stage 1..Stage 6*. The first of these contains two, rather complex, subparts which are described by pages *Filter Multiplier Cell* and *Weight File Register*. The remaining pages describe simple functional units: an *Or Gate* (with two, three, or four inputs), a 16-bit *Adder*, and a 12-bit *Limiter*.

Page *VLSI System* is shown in Fig. 10.2. It models three *Filter Chips*, of which the upper one is connected to a test environment that supplies random input values and clock pulses. Once the behaviour of a single chip has been validated, the test environment can be extended to include all three chips (or any other desired number of chips). The test environment can easily be modified to read test data from a text file and record the results on another file.

Page *Filter Chip* is shown in Fig. 10.3, from which it can be seen that the chip has a pipelined structure. Stage n receives clock pulses from stage $n+1$ and sends clock pulses to stage $n-1$. This is done via the places where the input/output arcs carry a small c . When a clock pulse is received, data is transferred to the next stage. This is done via the places with colour sets *I16*, *I12*, and *Bool* (representing 16-bit integers, 12-bit integers, and booleans). To work correctly, each stage must transfer its current data to the following stage before it receives the next set of data from the preceding stage. Hence, clock pulses travel backwards, while data travel forwards. The chip processes six sets of input data concurrently. The data in *Stage 6* are the oldest, while those in *Stage 1* are the youngest.

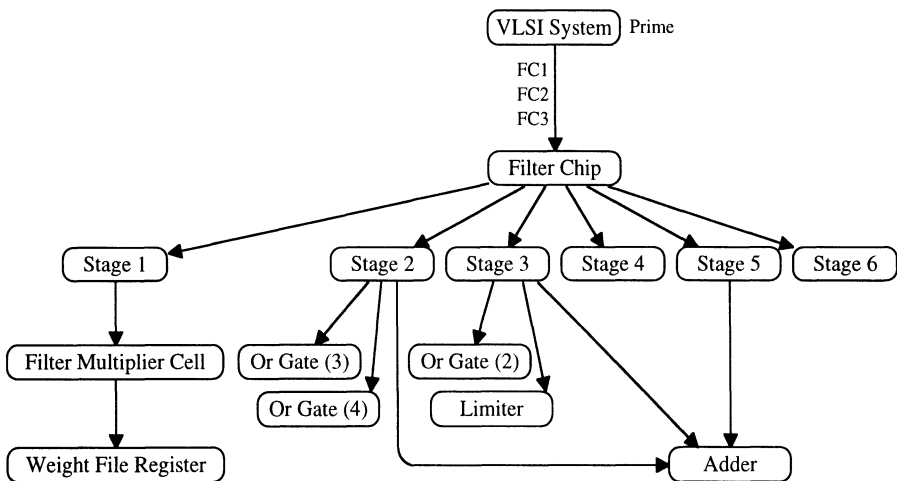


Fig. 10.1. Page hierarchy for VLSI chip

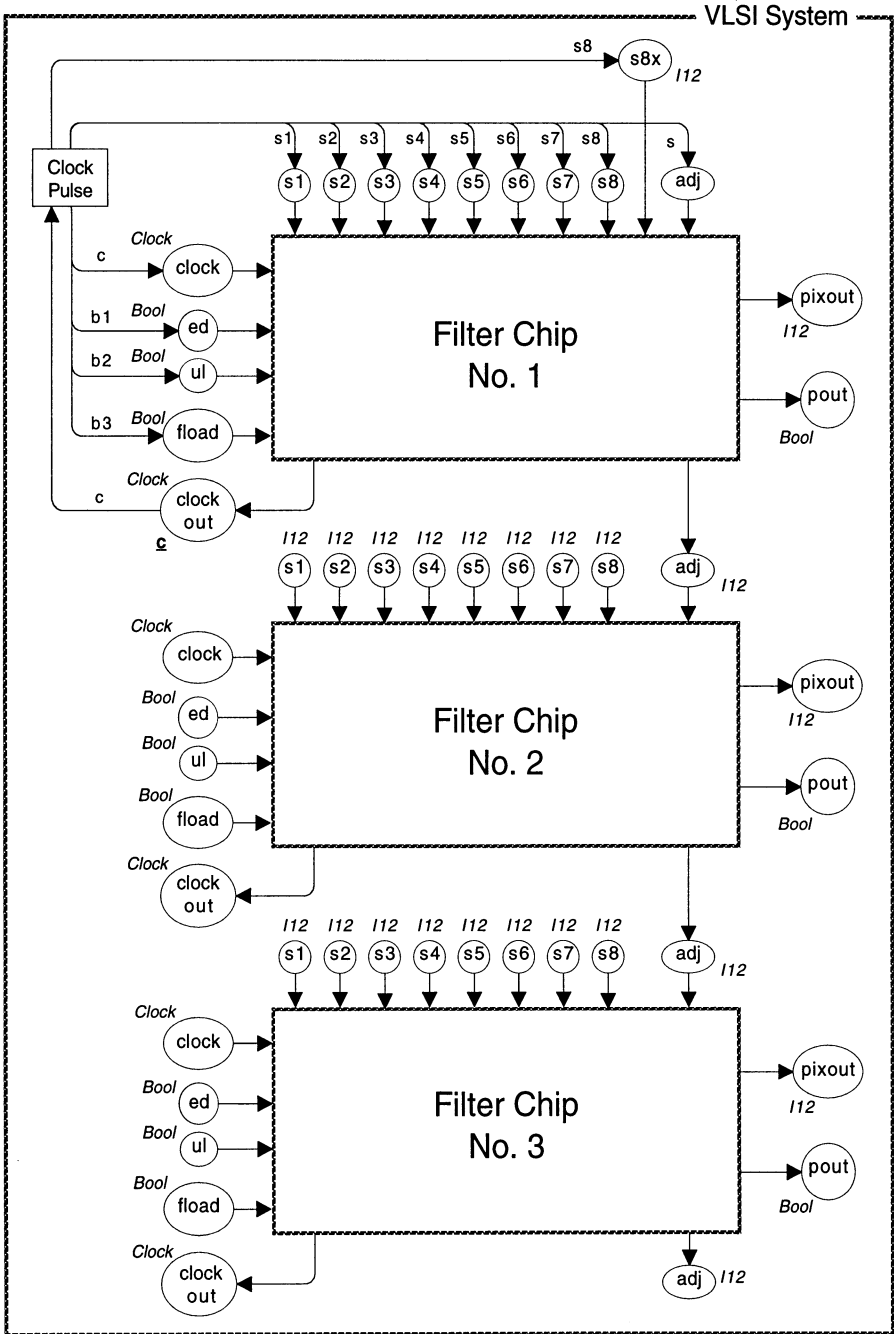


Fig. 10.2. CPN page for VLSISystem

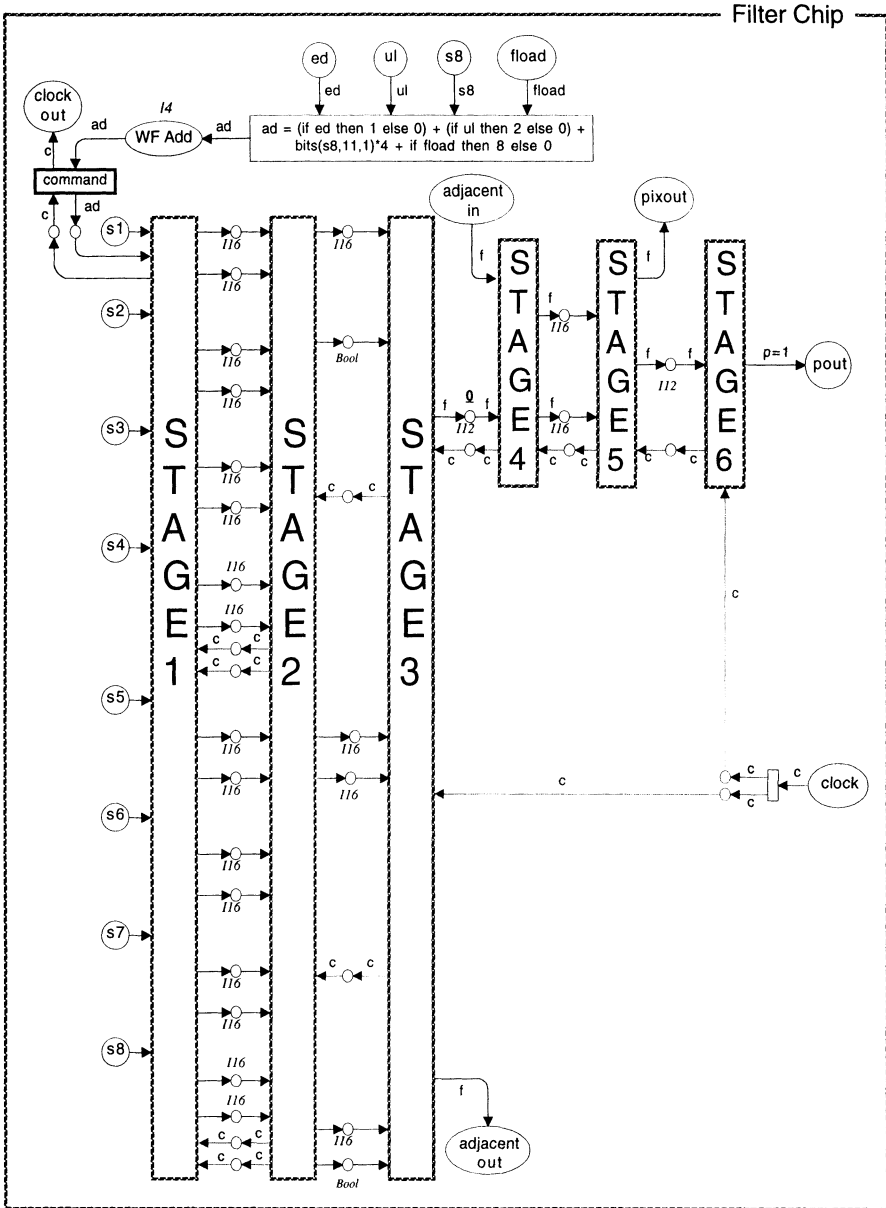


Fig. 10.3. CPN page for Filter Chip

The pages for the six stages are shown in Figs. 10.4–10.6. The eight transitions in the centre of *Stage 1* are substitution transitions, and they all refer to page *FilterMultiplierCell*. This represents the fact that the same functional unit is used eight times on the chip.

In *Stage 2* the four transitions *SUM1L*, *SUM1R*, *SUM2L*, and *SUM2R* represent registers. These registers establish the border between *Stage 2* and *Stage 3*, and it can be seen that they only occur when they receive a clock signal from *Stage 3*, via the two *c*-transitions in the rightmost part of *Stage 2*. When a register is clocked the values present on the register's inputs are transferred to the register's outputs. These new outputs then induce changes that propagate through the circuit. The propagation is blocked when arriving at the input sides of other registers. In this way the registers partition the circuit into a consecutive set of stages. All the remaining transitions in *Stage 2* are substitution transitions. *OR3* and *OR4* represent an *OrGate* (with three or four input signals) while + represents a 16-bit *Adder*. Altogether, each instance of *Stage 2* has 16 subinstances (four *OrGates* and twelve *Adders*).

The last four stages are much simpler. As before, transitions with a thick border line represent registers. The two *LIM* transitions in *Stage 3* are substitution transitions representing a 12-bit *Limiter*. The *Limit* transition in *Stage 5* is an ordinary transition, which performs a slightly different limiting operation. This kind of limiting operation is only used once, and hence it is specified directly in *Stage 5* instead of using a separate page.

The twenty places between substitution transitions *Stage 1* and *Stage 2* (in Fig. 10.3) are socket places. Each of these is related, via port assignments, to two port places (in Fig. 10.4) – one in the right-hand side of *Stage 1* and the other in left-hand side of *Stage 2*. The layout is made in such a way that the two related port places always are horizontally aligned. Moreover, the port places appear in the same vertical position as the corresponding socket place. A similar layout is made for the ports and sockets in all other pairs of consecutive stages (and this is the reason why *Stage 3* has a large empty space in the middle). For the remaining port/socket places we indicate the port assignment by using the same names for ports and their sockets. To improve readability, we have not shown the input/output tags and the port assignments, and we have also omitted the colour sets of all ports.

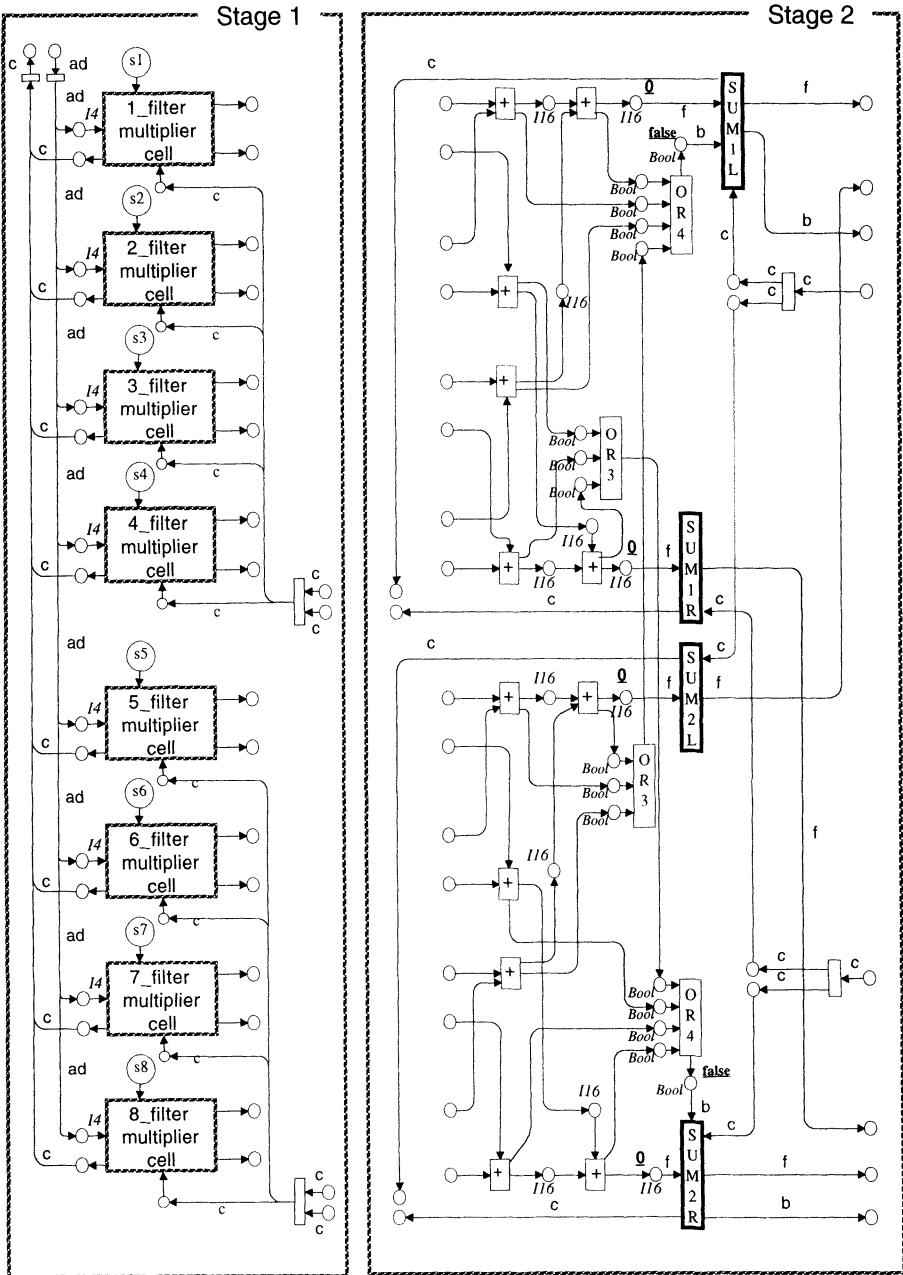


Fig. 10.4. CPN pages for Stage 1 and Stage 2

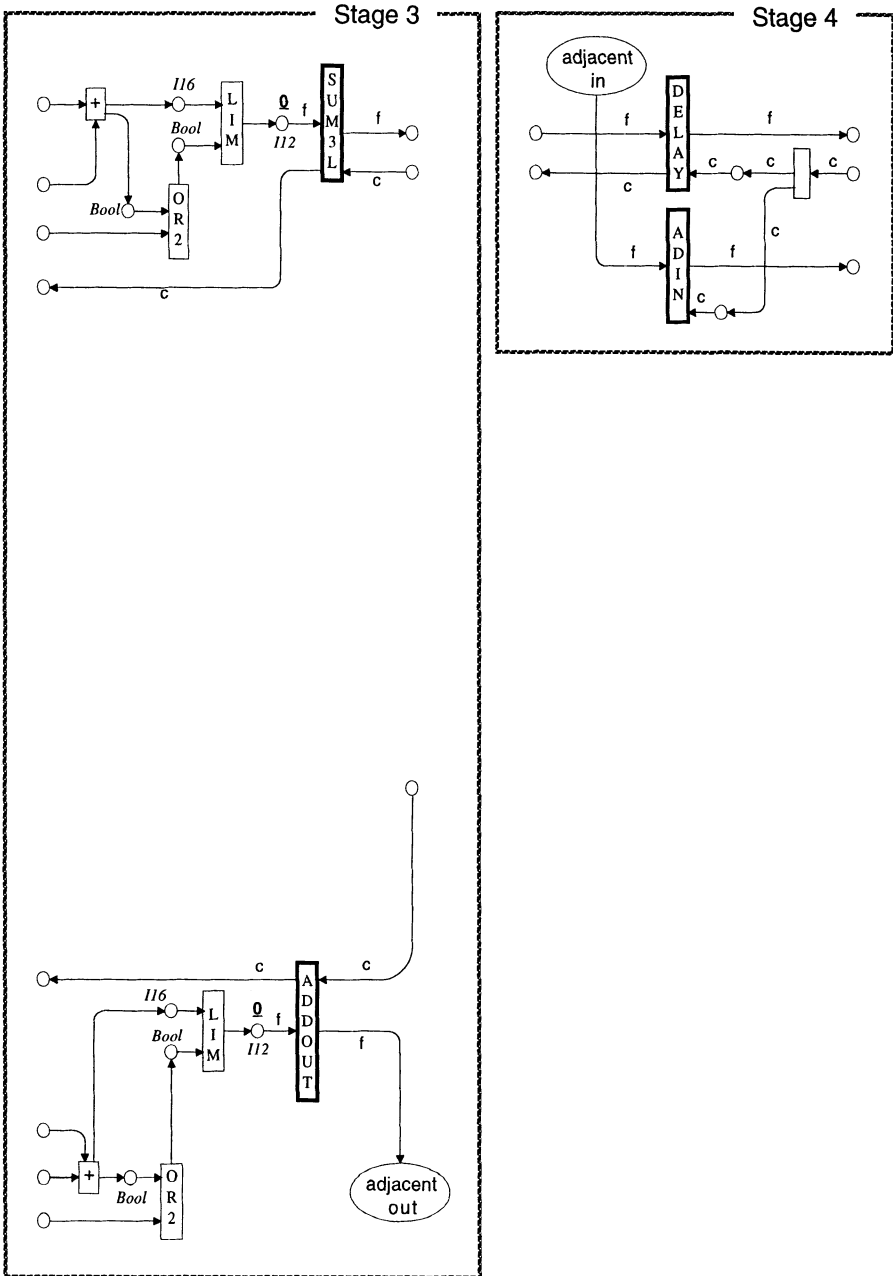


Fig. 10.5. CPN pages for Stage 3 and Stage 4

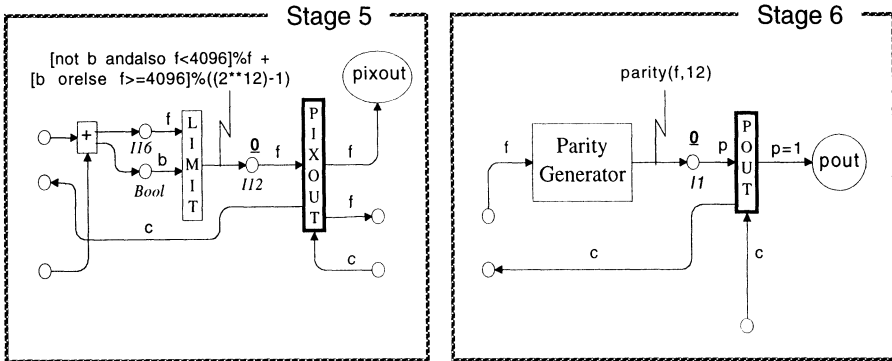


Fig. 10.6. CPN pages for Stage 5 and Stage 6

The remaining pages are shown in Figs. 10.7–10.8. We only show one of the *OrGates*. The others are similar, but they have a different number of arguments.

The declarations of colour sets and functions are quite simple. They look as follows:

```

infix **;
fun ((x:int)**0) = 1
  | (x**y) = x*(x**(y-1));
fun bits(i:int, s:int, n:int) = (1 mod (2**(s+1))) div (2**(s-n+1));
color Clock = with c;
color Bool = bool;
color I1 = int with 0 .. 1;
color I3 = int with 0 .. 7;
color I4 = int with 0 .. 15;
color I8 = int with 0 .. 2**8 - 1;
color I12 = int with 0 .. 2**12 - 1;
color I16 = int with 0 .. 2**16 - 1;
color I18 = int with 0 .. 2**18 - 1;
color Wfactors = product I3 * I8 * I1;

```

The infix function $x ** y$ returns x^y . The function $bits(i,s,n)$ allows an integer i to be viewed as a binary number. It extracts n contiguous bits, starting at high bit position s , and returns the result as an integer.

Here we do not give an explanation of how the VLSI chip works. Such an explanation can be found in [52].

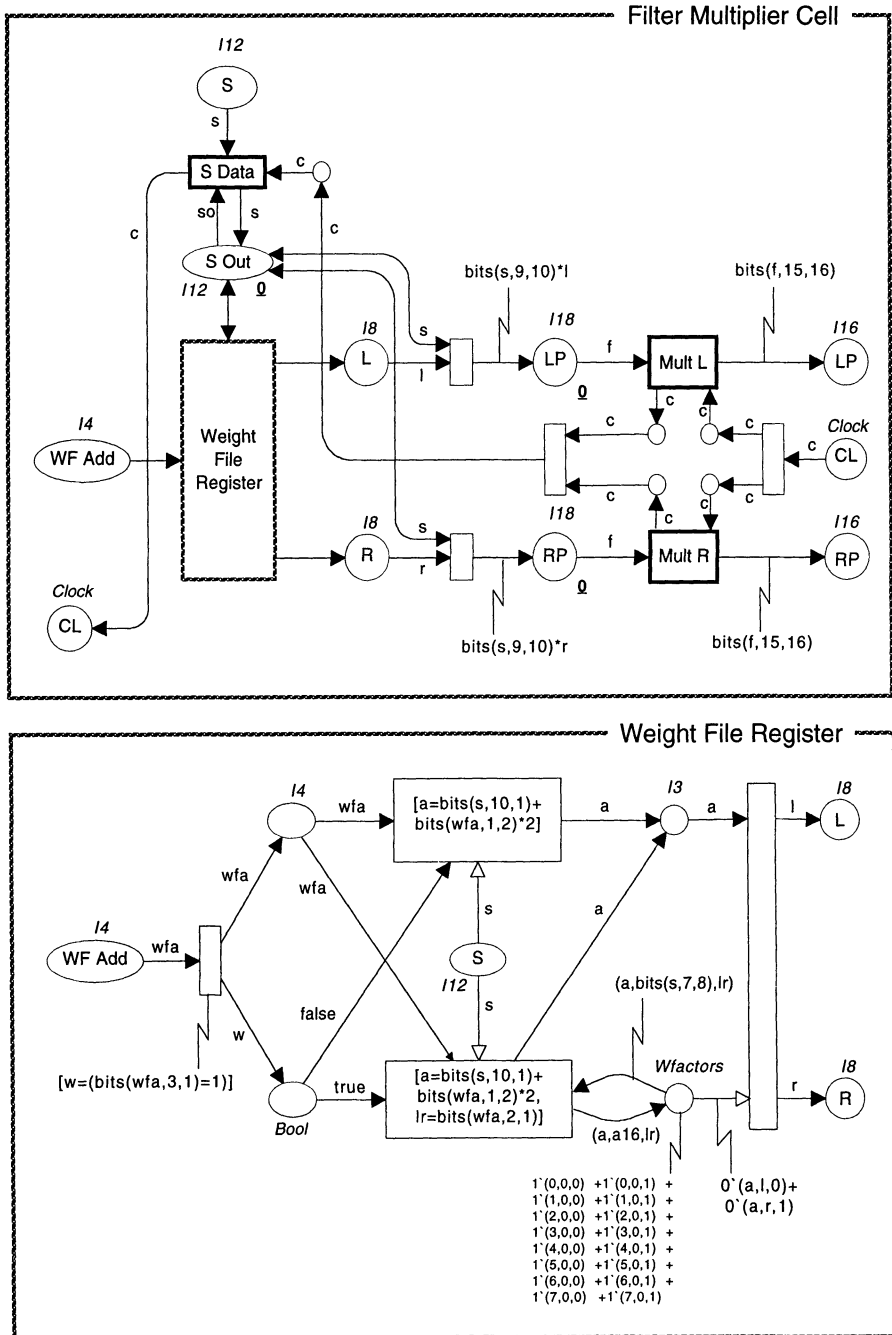


Fig. 10.7. CPN pages for two complex functional units used in Stage 1

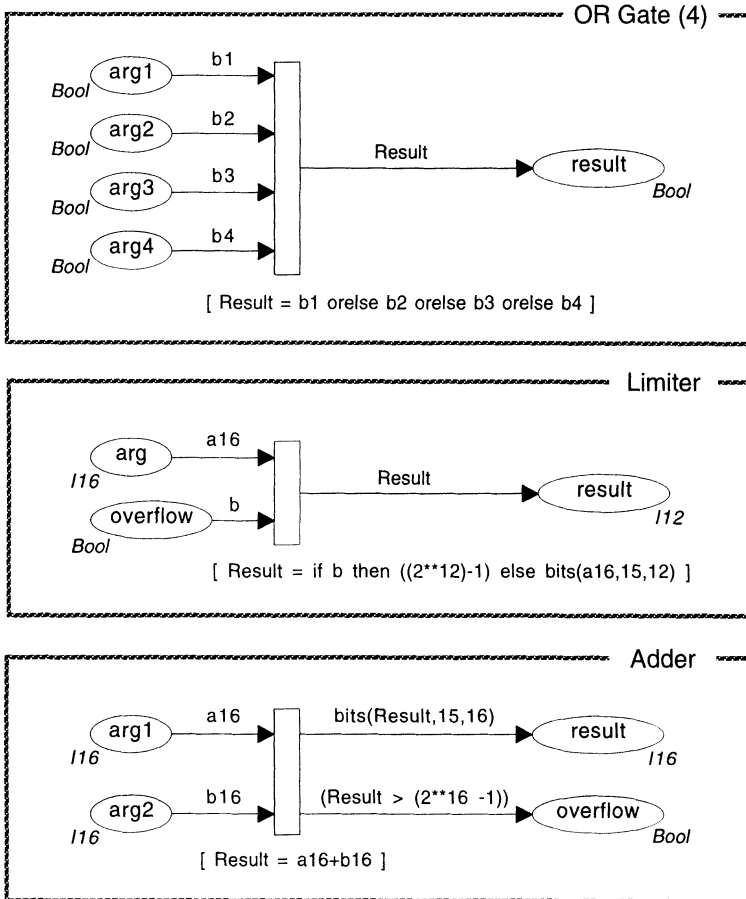


Fig. 10.8. CPN pages for three simple functional units

10.3 Conclusions for VLSI Chip Project

Now let us compare the new design and validation strategy with the old. First of all, it is easier to translate the block diagrams into a CP-net than it is to translate them into a C program. According to the participating company, the latter often takes several man-months, while the construction of the CP-net only took a few man-weeks. The translation is also more transparent, in the sense that it is much easier to recognise those parts of the CP-net which model a given block than it is to find the corresponding parts in the C program. This is due to the fact that each page of the CP-net has almost the same graphical layout as the corresponding block diagram, which means that it is relatively easy to change the CP-net to reflect any changes in the chip design. According to the chip manufacturer, it is

a major problem to maintain the C program. Moreover, as stated above, we believe that it will be easy to automate the translation from block diagrams to CP-nets. This means that the modified CP-net can be obtained without any manual work at all.

Secondly, when the new strategy is fully implemented, the designers will be able to make simulations during the design process. This means that the knowledge and understanding which is acquired during the simulation can be used to improve the design itself, in a much more direct way than in the old strategy, where the entire validation is performed after the design has been finished. Working bottom-up, we validated each low-level page by itself (e.g., the *Limitter*, *Weight File Register*, and *Filter Multiplier Cell*). To do this, values were manually inserted at the input ports. We used interactive simulations to investigate typical occurrence sequences, and we also made a number of automatic simulations. Working top-down, we validated some of the highest level pages (e.g., the pages for the six stages), without including all their subpages. In this way it was possible to validate some of the crucial parts of the design before all the detailed functional units had been specified.

Thirdly, the validation techniques of the old strategy concentrate on logic correctness, i.e., the functionality of the VLSI chip. Very little concern (and no tests) seems to be devoted to those design decisions which deal with timing issues, e.g., the division into stages and the determination of an adequate clock rate. This is surprising, because the timing issues are crucial for the correct behaviour and the effectiveness of the chip. Too-fast clocking implies malfunctioning while too-slow clocking implies unnecessary loss of speed. By means of timed CP-nets, it is rather straightforward to validate both logical correctness and the timing issues by means of a single CPN model. However, our project was carried out before timed CP-nets were developed and implemented, and thus the project did not involve a validation of the time issues.

Finally, occurrence graphs can be used to investigate both logical correctness and the timing issues. Due to the state explosion problem, it may not be possible to construct a full occurrence graph for the VLSI system. However, then we can use partial occurrence graphs or restrict the investigation to selected parts of the VLSI system, e.g., the individual stages. However, our project was completed before the occurrence graph tool was implemented, and thus the project did not use occurrence graphs.

The only real drawback of the CPN approach was the speed of computation. It turned out that the execution of the C program was much faster than the execution of the CPN model. The CPN simulator at that time was simply unable to make the usual number of test runs. However, the project was carried out immediately after the first version of the CPN simulator had been released. Based on the experience with this and other large models we have dramatically improved the speed of the CPN simulator. This means that it no longer is a problem to deal with the necessary amounts of test data.

Chapter 11

Arbiter Cascade

This chapter describes a project accomplished by *Hartmann Genrich, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, Germany* and *Robert M. Shapiro, Meta Software Corporation, Cambridge MA, USA*. The chapter is based upon the material presented in [29]. The project was conducted in 1992.

We describe how CP-nets and the CPN tools were used to model a cascade of arbiters. The main motivation for our work is the observation that CPN models perfectly meet circuit designers' need to visualise and experiment during the development of their designs. Moreover, designers can use occurrence graphs to validate the correctness of the design.

We develop two different types of CPN hardware models. Functional models specify what a component has to do, but not how it is done. To constitute a sound specification a functional model must exhibit certain properties. It must be deadlock free and it must react to input stimuli in the expected way. The validation of functional models is done by means of simulation and occurrence graph analysis.

The other type of CPN models is called circuit models. They are implementation oriented and use gate-like building blocks, such as and-gates, or-gates, and c-elements. A circuit model demonstrates the feasibility of a design. It can be validated by means of simulation and occurrence graph analysis. We also compare the behaviour of the circuit model to the behaviour of the functional model. This is done by means of occurrence graphs.

Section 11.1 contains an introduction to the arbiter cascade. Section 11.2 presents the two CPN models of the cascade. We also discuss the different kinds of validation which we performed. Finally, Sect. 11.3 presents a number of findings and conclusions for the project.

11.1 Introduction to Arbiter Cascade

The asynchronous access of a group of users (e.g., processors) to a single resource (e.g., a bus) can be regulated by a cascade of arbiters. Figure 11.1 shows a cascade of depth two. It contains three arbiters and serves four users. The bus itself is not shown.

Each arbiter serves two users at its input side and acts as a single user at its output side. By using a cascade it is possible to serve any number of users. A cascade of depth $d \geq 1$ contains $2^d - 1$ arbiters and serves 2^d users. The task of each arbiter is to reduce the behaviour of two independent users to the behaviour of a single one.

User i makes a **request** by setting the request line R_i high. Then it waits for the arbiter to decide whether it is safe to **accept** the request. If this is the case, the arbiter sets the acknowledgement line A_i high. Otherwise, it **rejects** the request by setting the not-acknowledgement line N_i high. Before the arbiter can make a positive answer, it must communicate with the rest of the cascade by issuing a request and waiting for this to be acknowledged. The cascade works in such a way that it is guaranteed that at most one user has an accepted request (i.e., simultaneously a high request line and a high acknowledgement line).

Eventually, the user **releases** the accepted request, which means that the bus can be used by other users. The release is done by setting the request line low. Now, the user must wait for its acknowledgement line to go low before making a new request.

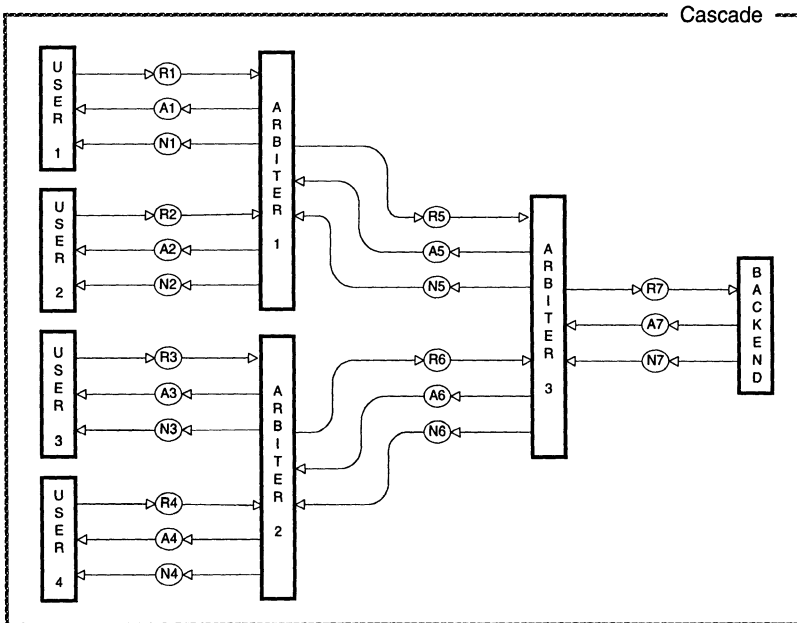


Fig. 11.1. Arbiter cascade

11.2 CPN Model of Arbiter Cascade

Starting with an incomplete description of the desired functionality of our arbiter, we first constructed a functional model providing a complete, unambiguous, and executable specification of our problem. The functional model has four CPN pages. The *Cascade* page provides the most abstract view of the system. It has already been shown in Fig. 11.1. All the transitions on the *Cascade* page are substitution transitions. They use three different subpages, which are shown in Figs. 11.2 and 11.3. The three pages represent the *Arbiters*, the *Users*, and the *Back End*. The latter is used to terminate the cascade. The declarations are very simple:

```

color Control = with c;
color Signal = bool with (L,H);    (* Low and High *)
color Status = with Idle | u1 | u2;
color Name = subset Status with {u1, u2};
color SignalxName = product Signal * Name;
color Answer = with AC | NK;
    
```

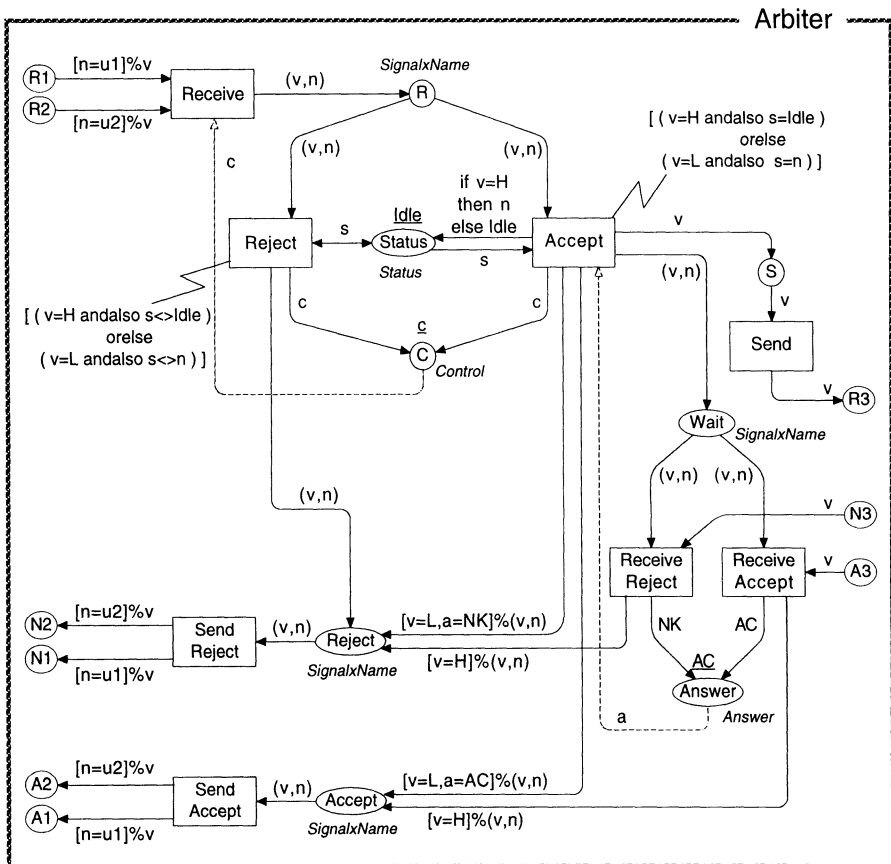


Fig. 11.2. CPN page for Arbiter

```

var v: Signal;
var s: Status;
var n: Name;
var a: Answer;
    
```

In Figs. 11.1–11.3 we have hidden all those colour set inscriptions that are equal to *Signal*. Ports are always positioned along the borders of the pages and they usually have the same name as their corresponding sockets (except that the latter may be augmented with a digit). To improve readability, we have not shown the input/output tags and the port assignments.

Now let us consider the *Arbiter* page in more detail. Incoming **requests** arrive at the input ports *R1/R2*. When one of these receives a high-token, transition *Receive* occurs (the transition is also enabled when a low-token arrives; this situation will be discussed later). The %-operator in the input arcs is a shorthand for an if-then-else construction. When the left-hand expression evaluates to true, the value is the value of the right-hand expression. Otherwise the value is the empty multi-set. The variable *n* is either bound to *u1* or to *u2*. In the first case a token is removed from *R1*. In the second it is removed from *R2*. When *Receive* occurs, it produces a token at place *R*. This token is a pair (H, ui) where *ui* is the requesting user.

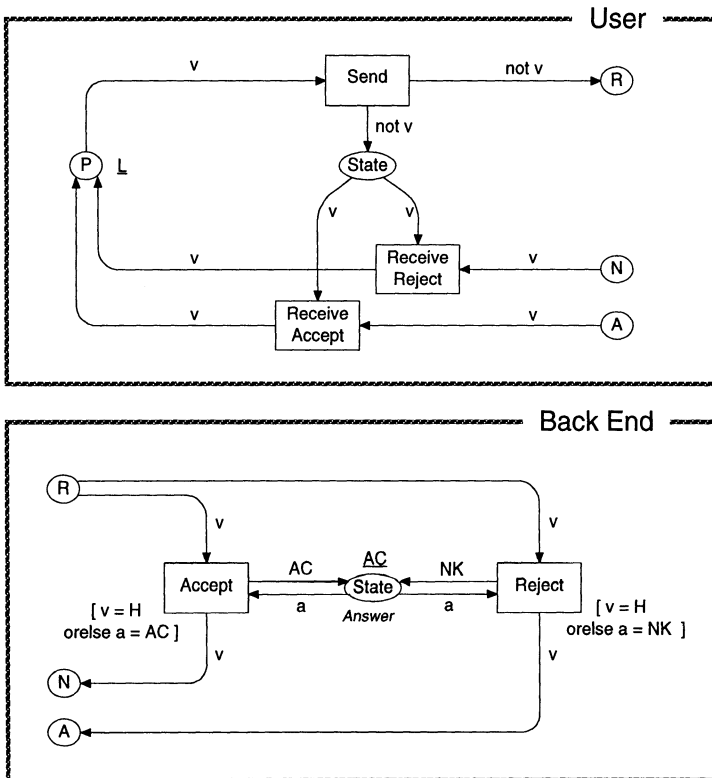


Fig. 11.3. CPN pages for *Users* and *Back End*

Then the arbiter either *Accepts* or *Rejects* the request. The choice depends upon the *Status* of the arbiter. This place has always exactly one token. When neither of the two users has an accepted request, the token value is *Idle*. Otherwise the token value identifies the user with the accepted request. If $Status \neq Idle$ the incoming request is *Rejected* and the user is notified via transition *SendReject* (which produces a high-token on $N1/N2$). If the request is *Accepted*, additional investigations must be performed, because another arbiter in the cascade may have a user with an accepted request. To perform this investigation the arbiter *Sends* a request to its parent in the arbiter cascade. If a positive answer is obtained from the parent, the user is notified via transition *SendAccept* (which produces a high-token on $A1/A2$). If a negative answer is obtained, the user is notified via transition *SendReject* (which produces a high-token on $N1/N2$).

Eventually the user makes a **release**. This is done by positioning a low-token on $R1/R2$. Transition *Receive* occurs as before, but now it produces a token of the form (L,ui) . This means that *Accept* and *Reject* works in a slightly different way than before. If the *Status* of the arbiter is different from the identity of the user, there is nothing to release, and the user is informed about this via *SendReject* (which produces a low-token on $N1/N2$). Otherwise, the *Status* is set to *Idle* and the user is informed via *SendAccept* (which produces a low-token on $A1/A2$). It is also necessary to inform the parent arbiter about the release. This is done by transition *Send*. The arbiter does not need to wait for an answer from the parent before it informs the user. This is because it can predict the answer to be received from the parent, by looking at place *Answer*.

When the functional model was finished, it was validated by means of simulation and occurrence graphs. In addition to verifying standard dynamic properties, such as liveness and boundedness, we showed that the CPN model had the expected behaviour, i.e., that it responded to user requests in the way which we expected. We also localised the appearance of true non-determinism (i.e., conflict situations, as opposed to the interleaving of concurrent events).

When the functional model had been validated, it was used to develop a circuit model. This model is, in many respects, similar to the CPN model presented for the VLSI chip in Chap. 10. However, there is an important difference, which can be seen from page *OrGate* shown in the upper part of Fig. 11.4. In the arbiter model the gate-transitions do not remove tokens from their input places. This means that the places always contain a token (which is either *L* or *H*). It is not the tokens but the changes of their colours that flow through the net. To avoid that the transitions occur all the time, without any effect, we add a guard. In this way, a transition only occurs when it alters at least one output.

One of the most crucial elements in the circuit model is the *Mutual Exclusion Element*, which can be seen in the middle part of Fig. 11.4. It chooses randomly between two high-tokens (on *arg1* and *arg2*), passing on only one of the requests and blocking the other until it is withdrawn, i.e., until the corresponding input goes low.

Altogether, the circuit model contains 18 pages. The most abstract CPN view of the arbiter circuit is presented in the lower part of Fig. 11.4. It uses three

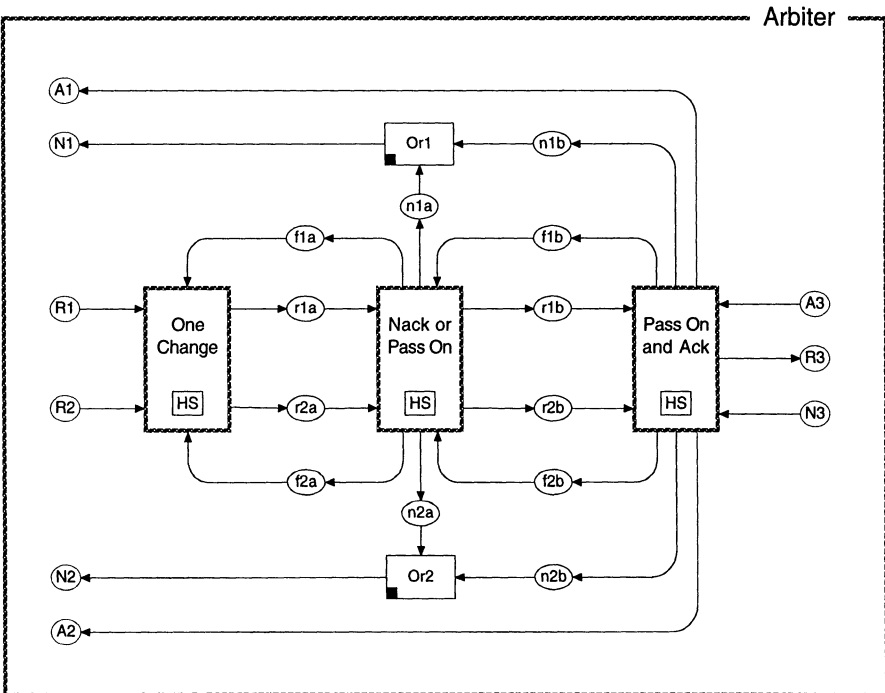
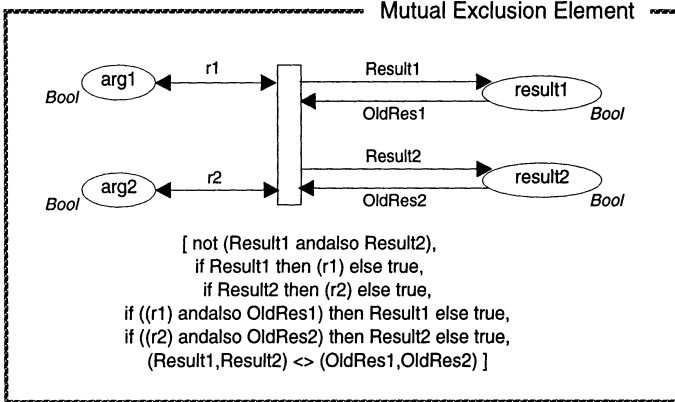
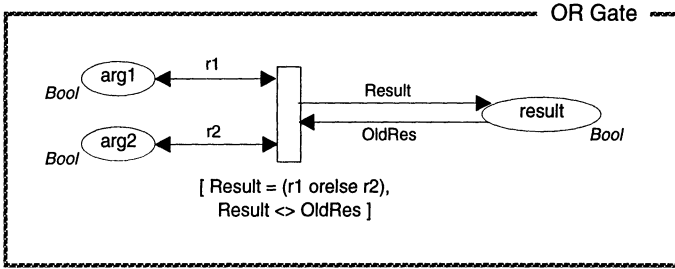


Fig. 11.4. CPN pages for *Or Gate*, *Mutual Exclusion Element* and *Arbiter*

complex subpages, which are shown in Figs. 11.5–11.6. All transitions with a small black dot in one of the corners are substitution transitions that represent simple gates (e.g., an or-gate, and-gate, half-adder, c-element, or clocked flip-flop). Each of these subpages has a single transition and works in a similar way as the *OrGate* in Fig. 11.4. The transitions with two black dots have subpages that use two such simple gates. The declarations are extremely simple. There are only two colour sets:

color Control = with c;
 color Bool = bool with (L,H);

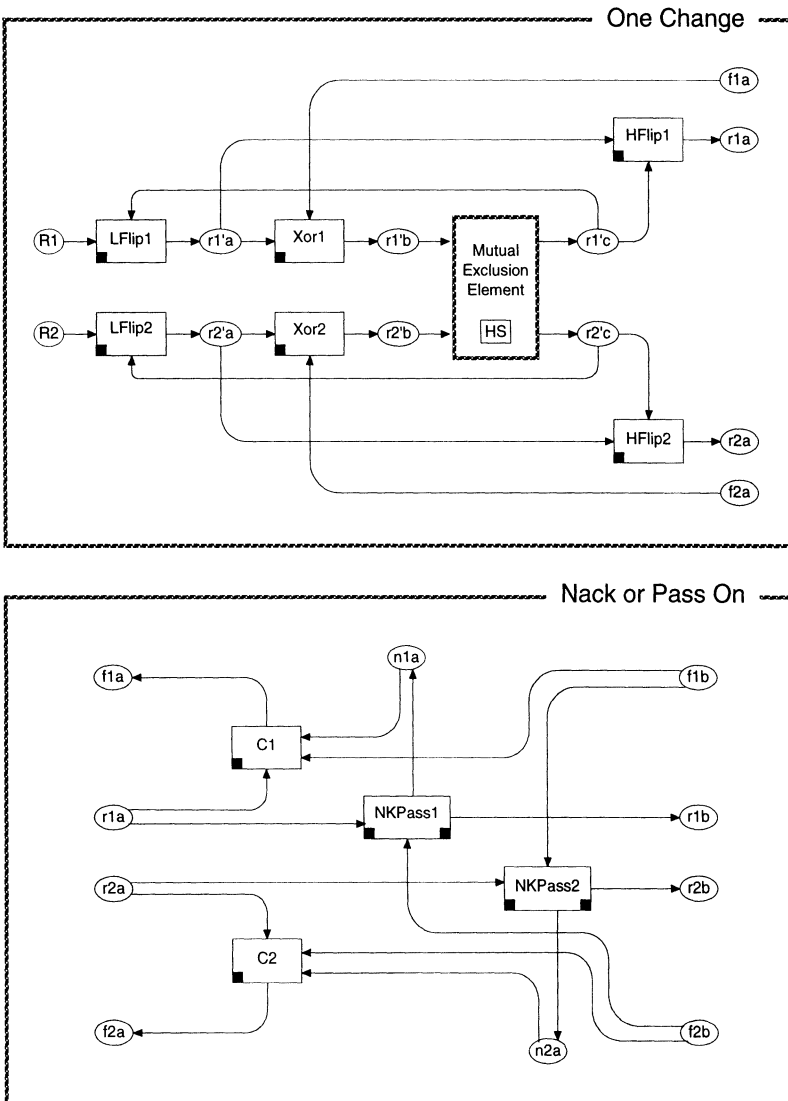


Fig. 11.5. CPN pages for *One Change* and *Nack or Pass On*

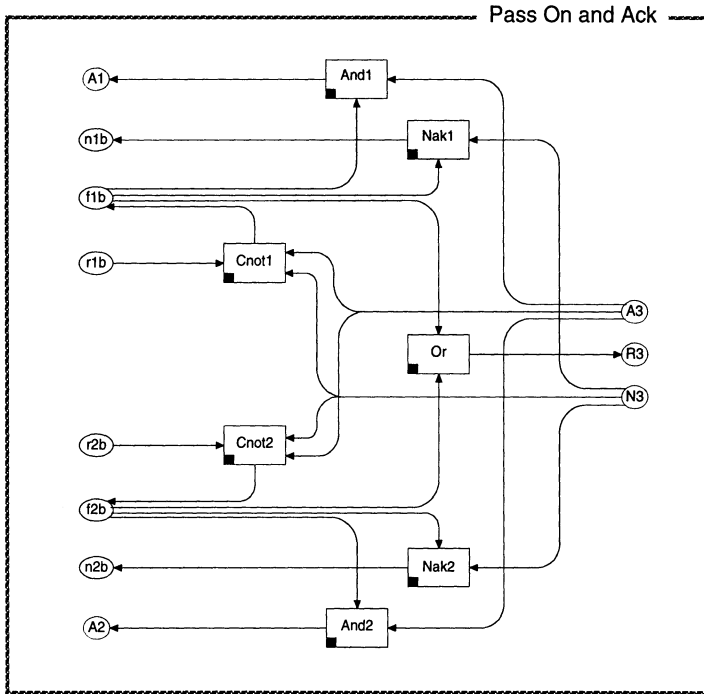


Fig. 11.6. CPN page for *Pass On and Ack*

11.3 Conclusions for Arbiter Cascade Project

We have shown that CP-nets are adequate for the representation of hardware designs, both at the functional level and the circuit level.

The functional and circuit models presented in this chapter are so complex that they cannot, in practice, be validated by manual methods. We have found that the use of simulation is very important during the construction and initial debugging of the two kinds of models. In this way, hardware designers are able to experiment with their designs, trying out new ideas and obtaining valuable knowledge to be used in the rest of the design process.

However, simulation is inappropriate for discovering some kinds of malfunctioning, e.g., those caused by critical races. The problem is that these kinds of error appear in such unexpected situations and so seldom that they are unlikely to be found by simulations. Hence, we also used occurrence graphs. They allow us to investigate all possible occurrence sequences in a systematic way.

We also used occurrence graphs to compare the behaviour of our two CPN models. We selected the places on the *Cascade* page (in Fig. 11.1) to be the interesting ones, because they describe the protocol by which the *Arbiters* communicate with the *Users*, the *Back End*, and the other *Arbiters*. For the circuit model

these interface places always have a token (of value L or H). Hence, it was straightforward to map each marking into an **abstract marking**, simply by ignoring all the other places. For the functional model the situation is slightly more complicated, since we do not always have a token on the interface places. When no token is present, we map the place into the value that the last token had. For this to be well defined it is necessary that all the possible last tokens (in the occurrence graph) have the same value, but this can easily be checked.

Having defined abstract markings for both the circuit model and the functional model, we compared the sequences of abstract markings generated by the paths in the two occurrence graphs. For each path in one of the O-graphs, we show that there exists a path in the other O-graph, such that the two paths have exactly the same sequence of abstract markings (when multiple, consecutive appearances of the same abstract marking are removed). In this way we have shown that the two CPN models have the same observable behaviour with respect to the selected set of interface places. The comparison was done on the fly, i.e., while the occurrence graph of the circuit model was being constructed. This is important, because small implementation errors may blow up the size of the O-graph and hence make it impossible to construct. More details about the comparison can be found in [29]. There it is also discussed how to investigate the delay-sensitivity of our design.

In an earlier paper [28], we constructed a functional model and a circuit model for a simpler version of the arbiter. There we combined the use of occurrence graphs with induction. We used occurrence graphs to prove that a single arbiter had some specified properties. Then we used induction to prove that this was also the case for a cascade of arbitrary depth.

Chapter 12

Document Storage System

This chapter describes a project accomplished by *Gert Scheschonk and Michael Timpe, C.I.T. Communication and Information Technology, Berlin, Germany, in cooperation with Bull AG*. The chapter is based upon the material presented in [50] and [51]. The project was conducted in 1993.

We discuss the modelling and simulation of a document storage system using CP-nets and the CPN tools. The modelled system is part of a large system which is capable of storing 20–30 million documents corresponding to a storage capacity of 10 000–15 000 Gbytes. A total of more than a thousand users request an average of 10 000 documents per hour. The documents are up to 75 years old. Hence they are scanned and stored as image files.

The main goal of our project was to develop a CPN model of the critical parts of the system. The model was used to identify bottlenecks and to determine a hardware configuration providing the desired capacity and response time. For the storage media we considered different types and capacities. Moreover, we considered different kinds of juke boxes.

An essential aim of our simulation was to investigate whether it would be possible to fulfil the required response time. When a set of documents is requested, the first document must be delivered within some specified amount of time, while subsequent documents must be available in the correct sequence and within the specified time interval. The documents are randomly distributed over the servers. This means that a request often will involve several servers.

Section 12.1 contains an introduction to the document storage system. Section 12.2 presents the CPN model of the document storage system. Section 12.3 discusses how the CPN model was simulated to determine a suitable configuration providing the desired response time. Finally, Sect. 12.4 presents a number of findings and conclusions for the project.

12.1 Introduction to Document Storage System

The documents in the storage system are divided into four priority classes, known as service levels:

- SL1 handles 80% of all requests. The documents in this class must be available without any noticeable delay.
- SL2 handles 10% of all requests. The documents must be delivered within 5 minutes.
- SL3 handles 9% of all requests. The documents must be delivered within 10 minutes.
- SL4 handles the remaining 1% of all requests. No response time is specified. This level is also used for backup of documents in the other service levels.

Figure 12.1 shows the different kinds of hardware which are used for the four service levels. The documents of SL1 are stored on optical disks which are directly accessible from the users' local workstations. The documents from the other service levels are stored in a global storage system. SL2 uses juke boxes of

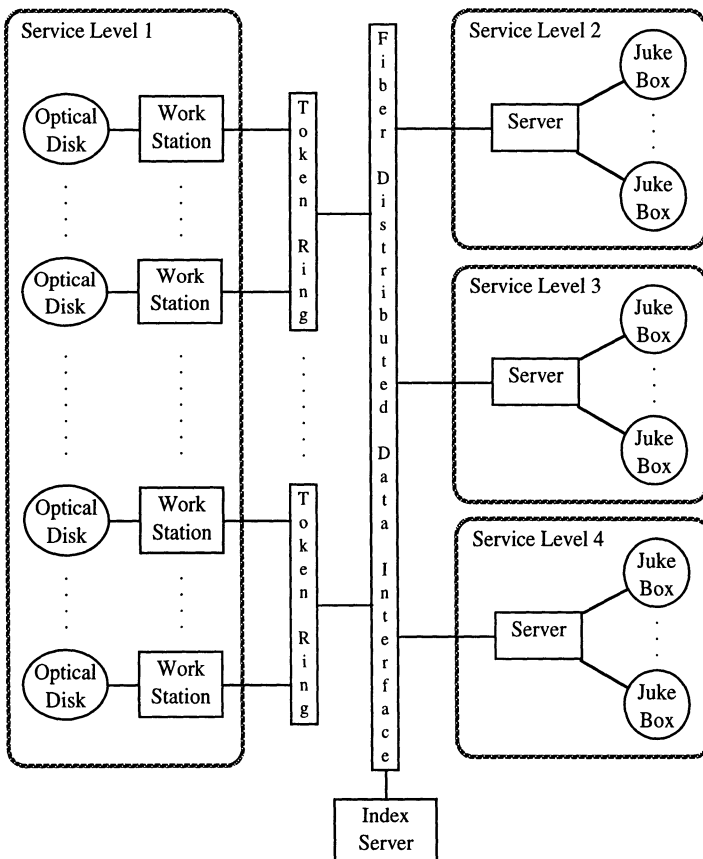


Fig. 12.1. Global architecture of document storage system

optical disks, while SL3 and SL4 use juke boxes of WORM CDs (Write Once, Read Many times). The documents are randomly distributed over the media. An index server keeps information about the location of all documents.

The users have access to the system via workstations which are connected to the storage system by means of a number of token rings and an FDDI backbone ring (Fiber Distributed Data Interface). Each server handles a number of juke boxes. The system covers three different physical locations. Site 1 has the complete equipment shown in Fig. 12.1. Site 2 only has SL1–SL3, while Site 3 has SL1–SL2. To obtain a document, a workstation first makes a request for location at the index server (which is shared by all three locations). Then it makes a request for delivery at the corresponding document server.

The CPN model focuses on SL2 and SL3. There are 660 concurrent users. Each hour they request 6 000 documents from SL2 and 4 600 documents from SL3. Requests are divided into shorts requests (which involve 1–3 documents) and long requests (which involve 4–12 documents). The requests are assumed to be uniformly distributed over time (with a little bit of randomisation). This distribution was chosen by Bull.

To minimise the use of man-power and to be able to meet the tight project deadline, the CPN model only includes those parts of the system which are expected to be time-critical. No performance problems are expected at the index server. Hence it is not modelled. SL1 requests do not involve the servers and SL4 requests are few and with no specified response time. Hence we do not model these levels.

The purpose of our simulation is to evaluate the performance of the juke boxes positioned at the servers. It should be determined whether they are able to deliver the requested documents within the specified response time. Another important question is to determine an appropriate configuration of the storage system, i.e., the number of juke boxes in each server and the number of disks, disk drives, and robotics per juke box. The robotics are the mechanical devices that mount the different disks on the available disk drives. Usually, there are many more disks than disk drives. The mechanical process of mounting a disk on a disk drive by means of a robotic is very time-consuming. Hence the CPN model reflects this part in great detail.

12.2 CPN Model of Document Storage System

In this section we describe the project organisation and the CPN model which we constructed.

The project group contained two persons from C.I.T. and two persons from Bull. The former had a detailed knowledge of CP-nets and the CPN tools, while the latter had no prior exposure to Petri nets. Hence the project was preceded by a one-week training course, introducing the basics of Petri-net modelling and the CPN tools. After this training, the Bull project members were able to read and check the CPN models and to simulate different configurations on their own.

The project began with a two-day meeting in which the critical parts of the system were identified. The entire project lasted five weeks. Three weeks were used to create the CPN model, one week to test the model, and one week to simulate a number of configurations. Altogether the project used approximately six man-months. This includes the time used by other Bull personnel to provide information to the project group and to evaluate the results.

Figure 12.2 shows the most abstract view of the CPN model. It consists of an *initialisation* part, a *processing* part and a *reporting* part. The initialisation part reads the parameters of the system from a text file. Moreover, it generates tokens representing all system components (such as disks, disk drives, and robotics) and tokens representing all requests (and their subrequests). As mentioned above, a request involves 1–12 documents, and thus it is modelled as a pair, where one component is a *request time*, while the other is a *list of subrequests*. Each subrequest is represented by a record with the following fields:

```
{ ID = (Req,SubReq),
  TargetDisk = ((SL,JB),Disk),
  Priority = .....,
  Delays = [t1, t2, ...] }.
```

The *ID* field identifies the subrequest by specifying the ordinal number of the request and the ordinal number of the subrequest. The latter is an integer between one and twelve. *Target Disk* identifies the disk where the document is located. This involves a service level, a juke box number, and a disk number. *Priority* is used to favour the early subrequests (of each request) – since they should be finished before the later ones. The initial priority is specified by a system parameter. Finally, *Delays* is used to record the time delays caused by the

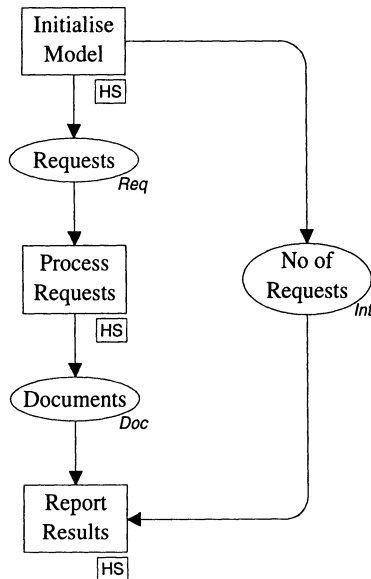


Fig. 12.2. Most abstract CPN view of document storage system

different operations. It is initialised to the empty list, and updated each time the subrequest encounters a delay.

Each disk has its own *disk queue*, which is represented by a triple:

$(((\text{SL}, \text{JB}), \text{Disk}), \text{Priority}, [(\text{sr}_1, \text{t}_1), (\text{sr}_2, \text{t}_2), \dots])$.

The first component identifies the disk. The second gives a priority which will vary over time. The third is a list where each element is a pair denoting a subrequest and its request time.

Next, let us consider the processing part of the CPN model. It represents the actions that are necessary to fulfil the requests. It starts by adding the subrequests to the appropriate disk queues. Then the subrequests are processed. Each juke box works in a loop, where it first selects a disk and then reads the documents specified by the subrequests in the selected disk queue. To select a disk, the disk must be unlocked and a disk drive and a robotic must be available. Moreover, the disk must have the highest priority (among those which can be chosen).

When a disk has been mounted and spun up, all the requested documents are read and temporarily stored in the memory of the server. If the size of the stored documents exceeds the memory size, the server has to swap some documents to virtual memory. This causes a significant delay and hence it is modelled. When all the requested documents have been read, the disk is dismounted and the disk drive becomes available for other subrequests. Then the server transmits the documents to the user workstations (via the FDDI backbone ring and the token rings). Finally, the server clears the documents from its memory.

When a document reaches the requesting workstation, all the recorded delays are passed to the reporting part of the CPN model. The reporting part creates a text file containing the delays gathered throughout the simulation. This file is used for subsequent analysis in a standard spreadsheet/charting program. For each subrequest the report file contains information about:

- request transfer time (from workstation to server),
- waiting time (for a free disk drive and robotic),
- preparation time (mounting, spin-up, reading of other requests in disk queue).
- reading time (from disk),
- stored time (in the memory of the server),
- document transfer time (from server to workstation).

The CPN model is relatively small and simple. The entire model consists of 10 pages with only 41 places and 20 transitions (of which nine are substitution transitions). The construction of the model was complicated by the fact that the CPN simulator, at that time, did not work efficiently when some places had a very large number of tokens. To overcome this problem, groups of subrequests were mapped into a single token, where the colour was a list with an element for each subrequest. Analogously, disk queues were represented by only two tokens – one for SL2 and one for SL3. These modifications made the net inscriptions and the necessary ML functions a bit complex. After our project the algorithms and data structures of the CPN simulator have been totally redesigned, and now it is no longer so important to avoid large numbers of tokens. More details about the CPN model and the necessary modifications can be found in [50] and [51].

12.3 Simulation of Document Storage System

We investigated four different configurations. For each of these, the system parameters were read from a file containing the following information:

- Number of subrequests (for SL2 and SL3).
- Maximal length of short/long requests and the ratio between the two kinds of requests.
- Initial priority for subrequests.
- Size of delays (for token ring transfers, spin-up, spin-down, reading of document, disk-pick, memory swap).
- Configuration of system (number of juke boxes, disks, disk drives, and robotics in SL2/SL3).
- Size of server's memory.
- Length of period (over which the requests are distributed).
- Capacity of each token ring (number of documents that can be concurrently transmitted).

We first investigated a configuration with 11 juke boxes having a total of more than 5 000 disks. For this configuration the results were pretty bad. Only 9% of the SL2 subrequests and 22% of the SL3 subrequests were handled within the specified limits of 5 and 10 minutes, respectively:

Configuration 1	SL2	SL3
Juke boxes	6	5
Disks (per juke box)	540	410
Disk drives (per juke box)	2	2
Robotics (per juke box)	2	2
Capacity of each token ring	4	
Requests inside limit	9%	22%

A more detailed analysis of the recorded delays showed us that the waiting time (for disk drives and robotics) and the document transfer time (over the token ring) were large, while all other delays were insignificant. Moreover, it was seen that the waiting time increased over the simulated time period. At the beginning the delay was reasonably, but gradually it became worse and worse.

To remedy the problems with the waiting time, we tried a configuration that used a different kind of juke box with fewer, much larger disks and only one robotic. This was expected to increase the performance, since less time would be used to shift between different disks. With the new configuration, we obtained the following slightly improved results:

Configuration 2	SL2	SL3
Juke boxes	8	7
Disks (per juke box)	51	37
Disk drives (per juke box)	2	2
Robotics (per juke box)	1	1
Capacity of each token ring	4	
Requests inside limit	10%	30%

The detailed analysis showed that the new configuration had managed to decrease the waiting time (for disk drives and robotics), but simultaneously the document transfer time (over the token ring) had become worse. Hence, we increased the capacity of the token ring from 4 to 7 (while the remaining configuration was unaltered). This led to a dramatic improvement:

Configuration 3	SL2	SL3
Juke boxes	8	7
Disks (per juke box)	51	37
Disk drives (per juke box)	2	2
Robotics (per juke box)	1	1
Capacity of each token ring	7	
Requests inside limit	83%	100%

Finally, we added one more juke box for SL2 (while the remaining configuration was unaltered). This gave us the following results, which were considered to be satisfactory:

Configuration 4	SL2	SL3
Juke boxes	9	7
Disks (per juke box)	51	37
Disk drives (per juke box)	2	2
Robotics (per juke box)	1	1
Capacity of each token ring	7	
Requests inside limit	98%	100%

12.4 Conclusions for Document Storage Project

Our project had two different goals. The prime goal was to investigate the document storage system and propose a configuration that met the specified requirements. A secondary goal was to introduce CP-nets and the CPN tools at the client company in such a way that they could be used in subsequent projects without external assistance. Both goals were achieved. The project group determined a suitable configuration and the client recognised CP-nets as a useful method which could be used to solve other problems arising within the company. After the project the client was able to continue the CPN experiments, e.g., to investigate new configurations.

The main technical problem encountered during our project was the decrease in simulation speed caused by having a CPN model with a very large number of tokens at some places. This problem was overcome by modifying the CPN model – lumping some of the tokens together. With the new CPN simulator this modification is unnecessary and hence it would have been much easier to develop the CPN model, which also would have been more straightforward to understand.

The CPN tools contain charting facilities that can be used to display simulation results. However, with the amount of data involved in our simulations, it turned out to be much more efficient and flexible to let the CPN model write all the raw results to a text file, for subsequent analysis in a standard spreadsheet/charting program.

Chapter 13

Distributed Program Execution

This chapter describes a project accomplished by *Jens B. Jørgensen and Kjeld H. Mortensen, Aarhus University, Denmark*. The chapter is based upon the material presented in [35]. The project was conducted in 1994.

We present a project in which CP-nets and the CPN tools were used to investigate a protocol used in the distributed implementation of an object-oriented programming language. The language allows objects to be distributed on several computers. The protocol is used when an object on one computer invokes an object on another computer. This works in a way which is similar to remote procedure calls.

During the modelling phase a number of simulation runs were performed to test the individual parts of the model (as these were developed) and to investigate the detailed behaviour of the protocol. The modelling and simulation increased the understanding of the protocol, and also caused some changes to the design and implementation of the protocol. Some of these changes were simple bug-fixes, while others were conceptual simplifications or improvements of the performance of the protocol (e.g., by removing some unnecessary critical sections).

At the end of the modelling phase, occurrence graphs and place invariants were used to verify that the protocol has certain desired properties, e.g., that there are no deadlocks, that objects always can continue to do remote object invocations, and that a monitor construction correctly ensures exclusive access to a critical section.

Section 13.1 contains an introduction to distributed program execution and the protocol used to implement remote object invocations. Section 13.2 presents the CPN model of the protocol. Section 13.3 discusses how the model was verified by means of occurrence graphs and place invariants. Finally, Sect. 13.4 presents a number of findings and conclusions for the project.

13.1 Introduction to Distributed Program Execution

The Beta language is an object-oriented language which has been developed at Aarhus University over the last 20 years (in cooperation with a number of other universities and research institutes). An introduction to the Beta language can be found in [37]. Recently, Beta has been extended to allow objects to reside on different computers [8]. Such objects may interact by means of remote object invocations, which in many respects are similar to remote procedure calls (RPC).

Remote object invocation is implemented by means of a protocol contained in an application framework called DistBeta. To understand the protocol, it is important to know the following concepts from DistBeta:

- An **ensemble** is the operating system (on a concrete computer on the network). Each ensemble contains a number of processes, which are called **shells**. A shell may communicate with other shells (in a remote ensemble or in its own ensemble). Moreover, a shell may communicate directly with its ensemble.
- Each shell contains a set of threads (lightweight processes). The number of threads may vary dynamically. However, each shell always has:
 - at least one **user thread** executing the main program of the shell,
 - exactly one **listener thread** taking care of incoming requests from the network.

The DistBeta framework has a class called the RPC handler. This class contains primitives for message passing and for serialisation and unserialisation of parameters (also known as marshalling). The parameters passed in an object invocation may be ordinary data values, objects, or references to objects.

Inside a shell each object has a local identifier. However, each object also has a unique global object identifier, OID, which can be used in the entire system. Each shell keeps two tables which are used to map local identifiers into global identifiers and vice versa. One of the tables is for local objects while the other is for remote objects.

Figure 13.1 shows an object OB_1 (in a shell SH_1 on a host HO_1) which invokes a remote object OB_2 (in a shell SH_2 on a host HO_2). This involves the following sequence of actions:

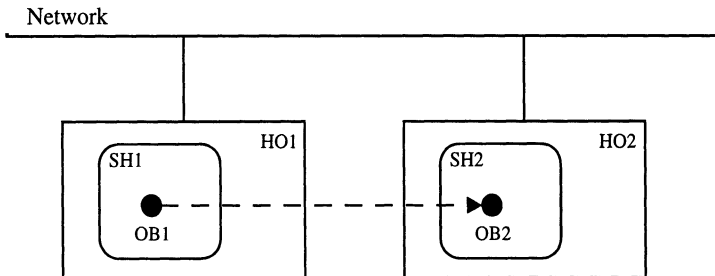


Fig. 13.1. Example of a remote object invocation

- OB_1 determines the OID of OB_2 . The parameters are serialised. OB_1 uses a method from the RPC handler of SH_1 to start the remote object invocation.
- A request message is sent from HO_1 to HO_2 . The message contains the OID of OB_2 and the serialisation of the parameters. OB_1 is blocked.
- The RPC handler in SH_2 receives the incoming request. A worker thread is allocated. The parameters are unserialised. The local identifier of OB_2 is determined from the OID (using a table in SH_2).
- The object OB_2 is invoked with the specified parameters.
- The worker thread receives the result. The result is serialised. Control is returned to the RPC handler in SH_2 . The worker thread is released.
- A reply message is sent from HO_2 to HO_1 . The message contains the serialisation of the result.
- The result is unserialised by OB_1 .

The protocol is rather lengthy and the complexity is increased by the fact that a number of shared resources are used, to which exclusive access is granted by means of monitors and semaphores. Some examples are: allocation/release of the different kinds of threads, use of the OID tables, and the request of a new unused OID (from an ensemble).

13.2 CPN Model of Distributed Program Execution

Our CPN model emphasises the description of the basic flow of control (inside threads and shells), the sharing of resources and the competition for access to critical sections.

Threads are represented by tokens with the colour set *Thread* in which each colour is a pair. The first element (known as the *ThreadInfo*) contains the identity of the thread (together with the identity of the shell and the identity of the ensemble to which the thread belongs). The second element (known as the *Environment*) contains different kinds of information needed to model the execution of the thread. Most places in the CPN model have *Thread* as colour set. The flow of *Thread* tokens describes the progress of the different threads. The presence of a *Thread* token on a place indicates that a thread is in the state represented by the place (e.g., ready to make an OID lookup).

Threads communicate by sending packets over a network. Packets are modelled by the colour set *Packet* in which each colour is a record with three fields – describing the sender, the receiver, and the contents of the packet. The latter describes whether the packet contains a request, a reply or an error message. The actual data values are omitted – we are interested in communication patterns, not in the concrete data values transmitted.

Figure 13.2 shows the page hierarchy of the CPN model. From this it can be seen that the model consists of four parts:

- *Top-level* consists of two pages (*DistBeta* and *ShellLevel*). They provide the most abstract view of the system.
- *Network* consists of a single page (*Network*).

- *Sender* consists of five pages (*Send*, *Get*, *Put*, *Assign OID*, and *RPC call*).
- *Receiver* consists of eight pages (*Receive*, *Listen*, *Execute*, *Generate OID*, *Get*, *Put*, *Assign OID*, and *RPC call*). The last four pages are shared with the *Sender*.

The total system contains 12 pages (with a total of 17 page instances). It has approximately 50 transitions and 70 places. Below we only describe two typical pages. The other pages are comparable with respect to the size of the net structure and the complexity of the net inscriptions.

Page *RPC call* is shown in Fig. 13.3. The net elements with thick lines represent the control flow of the involved threads. By convention all places with thick border lines have *Thread* as colour set. The two remaining places (in the left-hand part) represent the interface to the network. By convention all places with this kind of border line have *PacketBuffer* as colour set. A token on one of these places has as its colour a pair in which the first element identifies a shell SH_i , while the second is a list of *Packets* (representing a queue of packets) being sent from/to SH_i .

When the topmost place in Fig. 13.3 contains a token, the corresponding thread is ready to initiate a remote object invocation, having already finished the necessary preparations, e.g., the serialisation of parameters. When transition *Send* occurs, a new packet is added to the output queue for the corresponding shell. This is modelled by changing a token colour at place *ToNet* from $((ensID, shlID), pckList)$ to $((ensID, shlID), pckList \wedge [pck])$, where $(ensID, shlID)$ identifies the shell (and its ensemble), while $pckList$ is the old contents of the

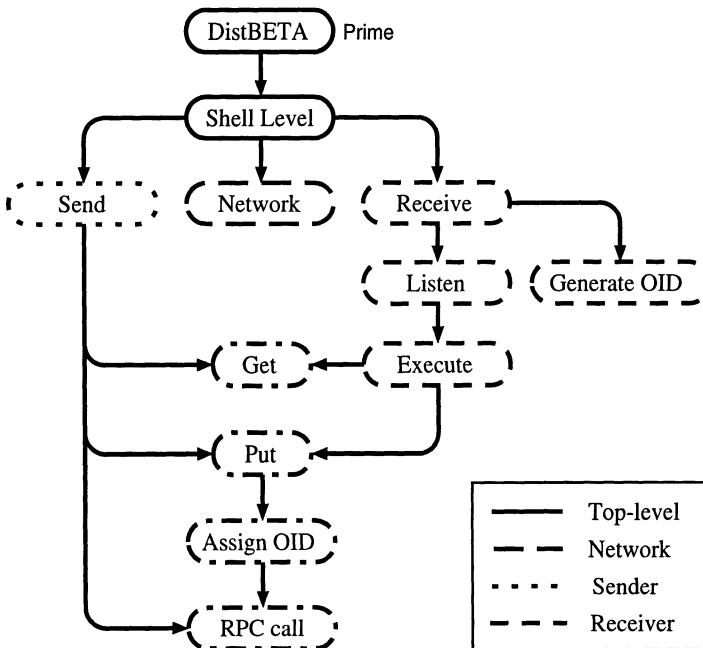


Fig. 13.2. Page hierarchy for distributed program execution

buffer, and *pck* is the new packet. The \wedge operator concatenates two lists. The first two lines of the guard guarantee that the packet is put in the correct buffer, i.e., the one that corresponds to the *ensID* and *shlID* specified by the variable *thrInf* (which is of type *ThreadInfo*). The third line of the guard determines the new packet. This is done by means of a function *NewPack* which as parameter uses the *RPCparam* field in the environment *envr*.

After the *Send* operation the thread becomes *Blocked* until an answer is received via place *FromNet*. When this happens the *Receive* transition occurs. It removes a packet *pck* from the head of the input queue of shell *shl*. The guard checks that the packet is addressed to the *Blocked* thread and also checks that the packet contains either a reply or an error message (i.e., differs from *aRequest*). If it contains a reply, the thread enters the *Success* state. Otherwise the thread either enters the *Begin* state or the *Error* state, depending on the value of the boolean function *Retry* (which uses a field in the environment of the thread).

From Fig. 13.2 it can be seen that page *RPCcall* also is used by the *Receiver* part of the *DistBeta* system. This happens when a receiving object performs an RPC with its ensemble to obtain a new *OID*.

Page *AssignOID* is shown in Fig. 13.4. It models how *OIDs* are obtained – either from the ensemble or from a local cache. For the protocol to work correctly, the *OID* requests must be executed one at a time (for each shell). This is guaranteed by using a monitor, which is represented by the dashed place and the two dashed arcs (in the right-hand side of the figure). Initially, *MonitorFree* contains a token for each shell. When an *OID* request is initiated, the token for the corresponding shell is removed from *MonitorFree*. This prevents that other

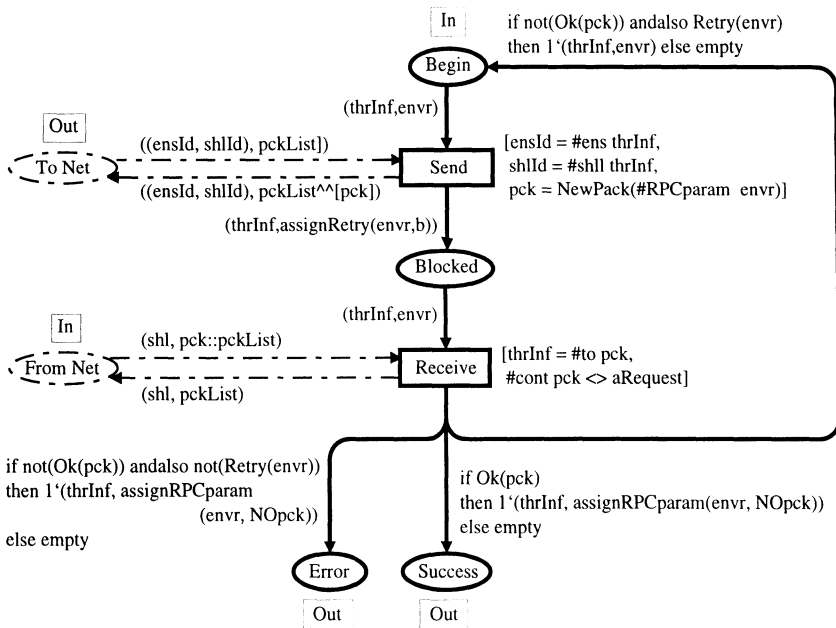


Fig. 13.3. CPN page for *RPC call*

threads (of the shell) enter the monitor. When the request finishes, a token is returned to *MonitorFree*. This means that other threads (of the shell) may enter the monitor. For the input arc to *P3*, we have only shown the first element in the arc expression. The second element is quite complicated and not important for the discussion in this chapter.

From Fig. 13.2 it can be seen that there are two instances of the page *AssignOID*. One page instance is used by the *Sender* part and the other by the *Receiver* part. However, there is only one monitor for each shell. This is modelled by letting *MonitorFree* belong to a global fusion set. Intuitively, this means that the two page instances share the place.

Our CPN model is an abstraction of the real protocol used in the DistBeta system. To be able to use occurrence graph analysis, we kept the model as clear and simple as possible. After discussions with the designer of the DistBeta system, we chose to ignore a number of protocol aspects, of which the most important are:

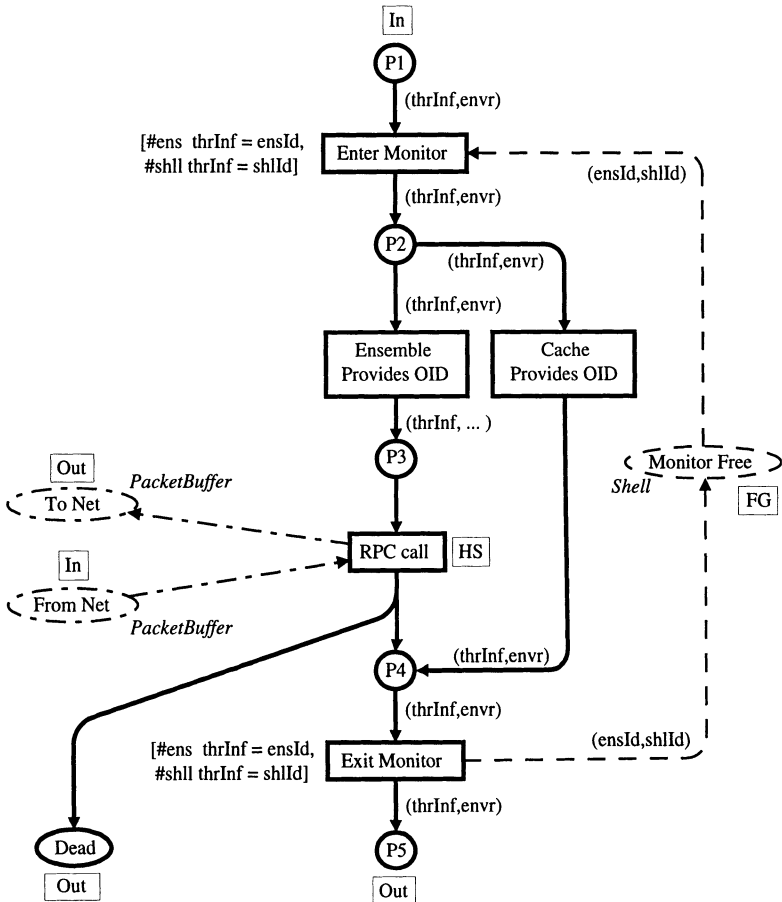


Fig. 13.4. CPN page for *AssignOID*

- We do not allow an object to act, simultaneously, both as a client and as a server. This means that it is impossible to create chains of invocations.
- We do not model that the monitors and semaphores use queues to guarantee that competing threads are served in a FIFO manner.
- We only model communication errors that happen on the sender side.

13.3 Verification of Distributed Program Execution

From the very beginning of our project, we had the intention of verifying the protocol by means of occurrence graphs. Hence, we were careful to construct the CPN model in such a way that the number of reachable states remained manageable. For example, we did not include actual data values in the packets, but only information about the packet type, i.e., whether the packet represents a request, a reply, or an error message. We also tried to model the different operations of the threads with as few transitions as possible, to minimise the state explosion caused by the different interleavings between operation sequences.

We constructed O-graphs for a number of different initial markings, each representing a possible configuration of the protocol system, i.e., a particular collection of ensembles, shells, and threads. Due to the state explosion, we were only able to construct O-graphs for small configurations. Hence, we cannot claim to have made a total verification of the protocol (which actually has an unlimited number of possible configurations). However, for small configurations we did manage to prove that the protocol possesses the following behavioural properties:

- The system has no dead markings (i.e., no deadlocks).
- All reachable markings are home markings (which implies that they all are reachable from each other).
- Each user thread remains active, in the sense that it always has the possibility of making a new request for a remote object invocation and getting a reply back (formulated as liveness of some particular sets of binding elements belonging to transitions *Send* and *Receive* on the page *RPCcall*; the reply may be an error message).

To verify these properties, we actually had to make a small modification of the CPN model. The problem was that an ensemble may fail to provide an OID to a thread, which then becomes *Dead*. To avoid this situation, we simply assume that all OIDs are provided by the local cache. This is achieved by adding the guard *[false]* to transition *Ensemble Provides OID* in Fig. 13.4. With this simple modification the three properties could be verified.

During the O-graph analysis we considered the four configurations shown in Fig. 13.5. Ensembles are drawn as boxes, shells as rounded boxes, and threads as black dots – similar to Fig. 13.1. As an example, the first configuration consists of one ensemble, with one shell and two threads. The O-graphs were constructed on a Sun Sparc 20 with 256 MB physical RAM. For each configuration we give the time used to calculate the O-graph and its size. For the last configuration we

were unable to construct a full O-graph due to lack of memory. Hence, we are only able to provide a lower limit for the computation time and size. To verify the dynamic properties mentioned above, we used a number of standard queries, implementing the O-graph proof rules described in Sect. 1.4 of Vol. 2. The time used to formulate and execute these queries was small compared to the time it took to construct the O-graphs.

Configuration 1 is more likely to lead to a dead marking than configurations 2 and 3 (in the case where we have an incorrect protocol). The reason is that all monitors, resources, and critical sections are local to shells, and hence things become more interesting when several threads belong to the same shell. For configurations 2 and 3, the O-graphs have exactly the same size. This is not surprising. In both cases, the two user threads are independent, because they belong to different shells. Hence, they never have to wait for each other.

It is easy to see that there are a number of other configurations with three user threads (e.g., three ensembles, one shell in each, and one user thread in each shell). However, all these configurations have O-graphs which are larger than the O-graph of configuration 4 (because threads that run in the same shell are forced to wait for each other while threads in different shells can proceed independently of each other). Hence, it should be clear that the available computer and tool support only allow us to construct full O-graphs for configurations 1–3. Here, it is worthwhile to mention that we used a version of the occurrence graph tool which was far from optimal with respect to use of memory.

Above, we have described how O-graphs were used to verify the final version of the CPN model. However, we also used occurrence graphs to debug our model. Occurrence graphs provide a fast and easy way to investigate a complex system. To handle a larger configuration, we can construct partial O-graphs. This allows us to investigate, in a systematic way, a much larger number of occurrence sequences than can be handled by simulation (in a reasonable time). More information about the construction of partial O-graphs can be found in Sect. 1.7 of Vol. 2.





Configuration		Minutes	Nodes	Arcs
1		1.6	5 501	13 725
2		12.7	21 554	54 793
3		10.9	21 554	54 793
4		≥ 87.5	≥ 75 018	≥ 183 827

Fig. 13.5. Size of O-graphs for distributed program execution

To reduce the size of the O-graphs and hence be able to deal with larger configurations, we tried to simplify our CPN model by mapping sequences of neighbouring transitions into a single transition, using a reduction rule formulated and proved in [31]. In this way we were able to remove five transitions and five places from the model. This may seem insignificant, since it removes less than 10% of all nodes in the CPN model, but it implied a 50% reduction of the size and construction time of the O-graphs for configurations 1–3. However, unfortunately we were still unable to construct a full O-graph for configuration 4. More details about the reduction can be found in [35].

It would also have been obvious to try to reduce the size of the occurrence graphs by using permutation symmetries, as described in Chap. 5. However, this was not attempted. At the time where our project was conducted, we did not have adequate computer support for such a task.

Compared to the occurrence graph method, the use of place invariants has the great advantage that we do not have to calculate all reachable markings, because the necessary checks are static and local. Another advantage is that it is possible to conduct proofs that are independent of the chosen configuration and hence valid for all of them. The major disadvantage of the place invariant method is that it is less automatic than occurrence graphs. To formulate and use place invariants, the user needs a fair amount of mathematical skill. Moreover, the use of invariants (to prove dynamic system properties) involves a mathematical proof which often is prone to error and time-consuming.

By means of place invariants we were able to verify the following properties of the protocol system:

- The set of user threads is constant, i.e., no user thread ever disappears and no new user thread is ever created.
- The set of listener threads is constant.
- The set of network input buffers is constant.
- The set of network output buffers is constant.
- The monitor at page *AssignOID* works correctly, i.e., there are never two user threads from the same shell inside the monitor.

From our knowledge of the protocol and the CPN model, it was quite obvious that we would expect these properties to be satisfied. It was straightforward to get the ideas behind the properties and also easy to formulate them in terms of place invariants.

For the first two properties this was done by choosing a set of weights that map all tokens representing user/listener threads into their *ThreadInfo* part, while all other tokens are ignored (i.e., mapped into the empty multi-set). For the next two properties we only used non-zero place weights at those place instances that correspond to input and output buffers (i.e., the places *ToNet* and *FromNet* in Fig. 13.4). For the last property, *MonitorFree* (in Fig. 13.4) got the identify function as weight. The places of the critical section got the weight *ShellID*, which maps each *Thread* token into the identity of the shell (and ensemble) in the *ThreadInfo* part. All other places had the zero-function as weight.

The correctness of the first four invariants could be established, quite automatically, by an early version of the invariant tool described in Sect. 4.4 of Vol. 2. However, for the fifth invariant the tool was unable to prove that the invariant was respected by transition *Receive* on page *RPC call*. This transition was simply too complex to be handled by the present version of the invariant tool. Hence, the transition had to be covered by a manual proof. To increase our confidence in the manual proof, we used the O-graphs to check that the fifth invariant was satisfied for configurations 1–3. We simply made a search where it was checked that each reachable marking fulfilled the invariant.

13.4 Conclusions for Distributed Program Execution Project

In this project we have modelled and verified a protocol for remote object invocation in an object-oriented language.

First we built the CPN model. The discussions with the protocol designer and the actual modelling work improved the understanding of the protocol. As a consequence a number of modifications were made to the protocol. In particular, some unnecessary critical sections were removed.

The model built in this project is too complex to be adequately investigated by simulations alone. Hence, the next step was to verify the CPN model by means of occurrence graphs and place invariants. O-graphs were used to prove dynamic properties such as absence of dead markings and liveness of specific sets of binding elements. Place invariants were used to prove a quite different type of dynamic properties, e.g., that certain sets of threads remain constant, and that a monitor construction correctly ensures exclusive access to a critical section.

During the verification no errors were found and hence this made no direct contribution to the protocol design/implementation. However, the verification increased our confidence in the correctness of the protocol.

Chapter 14

Electronic Funds Transfer System

This chapter describes a project accomplished by *Valerio O. Pinci and Robert M. Shapiro, Meta Software Corporation, Cambridge MA, USA, in cooperation with Marine Midland Bank of New York and Société Générale*. A brief presentation of the project can be found in Sect. 7.4 of Vol. 1. The chapter is based upon the material presented in [42]. The project was conducted in 1989.

We describe how CP-nets and the CPN tools were used to develop a bank software application for supervising electronic funds transfer. For our project we used a new software development methodology. Requirements analysis and system specification were done by means of Structured Analysis and Design Technique (SADT), producing a functional description of the system activities. System design and verification were done by means of CP-nets, creating executable models used for rapid prototyping and validation of behavioural properties. Finally, implementation was done by means of the Standard ML programming language. The three parts of the development process were closely interconnected. The CP-nets were derived from the SADT diagrams, while the final ML code was derived from the CP-nets.

The ideas presented in this chapter have also been used in a number of other projects. Some of these have combined SADT and CPN. They have augmented informal SADT descriptions with additional details and transformed them into CP-nets constituting a more precise and executable model. Examples are the Bank Courier Network in Chap. 15, the Nuclear Waste Management Programme in Chap. 19, and the Radar Control Post in Sect. 7.3 of Vol. 1. Other projects have used the ML code from the CPN simulator as implementation of the system. On an experimental basis, this was done for the Security System in Chap. 1.

Section 14.1 contains an introduction to SADT and explains how SADT diagrams are translated into CPN models. Section 14.2 introduces the electronic funds transfer system and the SADT model of it. Section 14.3 presents the different CPN models of the funds transfer system. The first CPN model was used for rapid prototyping while the last was used for generation of stand-alone ML code constituting the final implementation. Finally, Sect. 14.4 presents a number of findings and conclusions for the project.

14.1 Introduction to SADT

In our project we used three different kinds of descriptions/languages:

- Structured Analysis and Design Technique (SADT) was used in the early project phases to specify the functional decomposition of the system and the data flow dependencies between components.
- CP-nets were used to add data descriptions and behavioural aspects to the SADT specification. The CP-nets were simulated to study the dynamic properties of the new system and to evaluate the impact of different design decisions. This worked as a kind of prototyping.
- Standard ML was used to implement complex algorithms and as the target language for the semi-automatic production of executable code.

SADT is a method for functional description of complex systems. SADT is in widespread use in some European countries and in the United States (where it is known as IDEF). SADT diagrams are in many respects similar to CP-nets, and this means that they consist of a set of pages. In the SADT terminology each page is called a diagram. However, here we shall stick to the CPN convention and use the term diagram for a *set of pages* which constitutes a model. Each SADT page contains a number of rectangular boxes. They are called activities (or functions) and they model actions in a way which is similar to the transitions of a CP-net. The activities are interconnected by arcs, which are called arrows. There are three different kinds of arrows. They represent data/physical flow, control flow, and mechanisms (i.e., availability of resources). SADT has no counterpart to places and this means that the arrows interconnect activities directly with other activities. However, each arrow has a label, and this plays a role similar to the colour sets of CP-nets. The three kinds of arrows are distinguished by their position. They all leave the source node via the right side, but they enter the destination node from the left (data/physical flow), the top (control flow), and the bottom (mechanisms). Finally, each SADT page (except the most abstract page) is a refinement of an activity of its parent page (superpage), and this works in an analogous way to the substitution transitions of CP-nets. A detailed introduction to SADT can be found in [38].

SADT diagrams are often ambiguous. As an example, a branching arrow may mean that the corresponding information/material sometimes is sent in one direction and sometimes in another. However, it may also mean that the information/material is split into two parts, or that it is copied (and sent in both directions). The designers of SADT argue that it is adequate to allow such ambiguities, because they primarily view SADT as a language for description of functionality at an abstract level, without having to worry about the detailed behaviour, which in their opinion is an implementation detail. However, we want to use SADT to specify behaviour and we want to use it to specify executable simulation models. It is then obvious that all such ambiguities must be removed. This means that SADT must be augmented with better facilities to describe the detailed behaviour of activities, e.g., to specify what a branching arrow means.

There are many different ways to describe the behaviour. Several papers on SADT propose to attach a table to each activity. Each line in the table describes a possible set of acceptable input values and specifies the corresponding set of output values. Another, and in our opinion much more attractive possibility, is to describe the input/output relation by a set of arrow expressions and a guard – in exactly the same way that the behaviour of a CP-net transition is described by means of the surrounding net inscriptions. Thus we introduce a new SADT dialect, called SADT_{CPN} (or IDEF_{CPN}). In addition to the added arrow expressions and guards, SADT_{CPN} has a global declaration node (containing the declarations of types, functions, operations, variables, and constants). It is possible to use fusion sets, code segments, and time delays in a way similar to that of CPN models.

Due to the many similarities between SADT_{CPN} and CP-nets, it is straightforward to translate an SADT_{CPN} diagram into a behaviourally equivalent CPN model. This means that the CPN simulator can be used to investigate the behaviour of SADT models. An SADT tool [39] allows the user to construct, syntax-check, and modify SADT_{CPN} diagrams. This tool works in a similar way to that of the CPN editor, and many parts of the two user interfaces are identical. The SADT tool can create a file containing a textual representation of the SADT_{CPN} diagram, and this file can be read into the CPN simulator, where it is interpreted as a CPN model. The translation from SADT_{CPN} to CPN models is thus totally automatic. The user views the simulation results on the corresponding CPN diagram – but this is not a big problem because the two diagrams look almost identical. Figure 7.6 in Vol. 1 shows an SADT_{CPN} page, while Fig. 7.7 shows the corresponding CPN page obtained by the automatic translation.

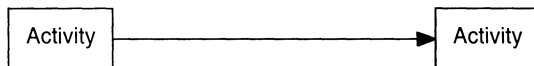
As described above, it is necessary to add behavioural information to an SADT diagram in order to obtain an executable CPN model. The behavioural information can be added before or after the automatic translation. If the behavioural information is added before the translation, i.e., in the SADT tool, we have the following sequence of diagrams:

SADT diagram \longrightarrow SADT_{CPN} diagram \longrightarrow CPN diagram.

If the behavioural information is added after the translation, i.e., in the CPN tool, we get:

SADT diagram \longrightarrow Net structure of CPN diagram \longrightarrow CPN diagram.

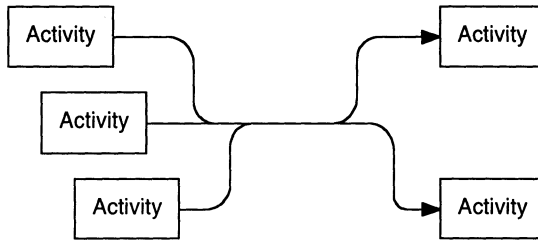
In the electronic funds transfer project we used the latter scheme. The automatic translation transformed the SADT diagram into the page hierarchy, the net structure, and the colour set names of a CP-net. The basic idea behind this translation is to replace each SADT arrow:



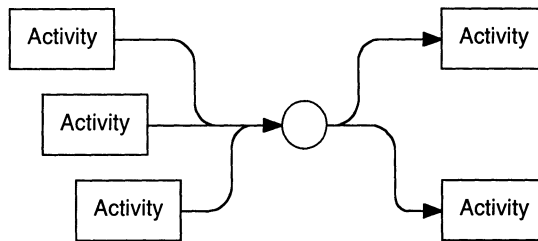
with a CPN place and two CPN arcs:



However, since SADT arrows often fork and join, it is more correct to say that we replace each bundle of SADT arrows:



with a CPN place and a number of CPN arcs:



To be able to do this, it is necessary to require that all arrows in a bundle join before they fork. This implies that the arrows have a common section, where the CPN place is positioned. This demand is not really a restriction. It is satisfied by almost all SADT diagrams, and when it is not, it is usually easy to obtain by a simple local modification of the corresponding SADT page.

Each CPN place gets a colour set which has the same name as the label attached to the corresponding SADT arrow. This means that there are a lot of different colour sets – often one for each place in the CPN model. In practice, many of these colour sets denote types which are structurally equivalent, i.e., have the same elements. It would of course be easy to replace each such group of colour sets by a single colour set. However, this would make the CPN models less comprehensible, because the SADT arrow labels convey important informal information about the purpose of the arrow. An alternative solution would be to map the arrow labels into place names, and then add colour sets manually. Some SADT dialects allow the modeller to create a data dictionary specifying a type for each arrow label. When this is done, it is straightforward to use the data dictionary to obtain the necessary colour set declarations.

The individual SADT pages are related to each other in a very similar way to the one known from CPN models. A complex activity may be refined, i.e., described in more detail at a separate page, typically by dividing it into 3–5 subactivities. The refinement page has the same incoming and outgoing arrows as the activity it refines. These inputs, outputs, controls, and mechanisms are mapped into sockets (of the substitution transition representing the refined activity) and ports (of the subpage representing the refinement). Actually, the SADT hierarchy concepts are slightly more restricted than those of CPN models. In SADT it

is not possible to reuse a page (without copying it). Hence there is only one instance of each page, and the page hierarchy is a tree, while in CPN it may be an acyclic graph. Moreover, SADT has no fusion sets. As mentioned above, fusion sets, code segments, and time delays have been added to the SADT_{CPN} dialect. This has turned out to be very useful, facilitating the modelling of complex systems.

The combination of SADT, CPN, and ML has a number of advantages. Some of the most obvious are the following:

- SADT's loose notation, its simple graphic layout conventions and its strong decomposition primitives make it easy to learn. Moreover, there are a number of textbooks and courses available. In a few weeks designers can become familiar with the SADT language and learn to generate descriptions that are easy to understand and well structured.
- The automatic translation from SADT diagrams into the net structure of CPN models ensures a large degree of consistency between the descriptions used in the early and middle phases of the system development. This is in particular the case when the behavioural information is added to the SADT diagrams before the translation into CP-nets.
- CP-nets allow the designers to prototype/validate different aspects of the system. CPN models can be obtained throughout the entire analysis and specification phase. In this way valuable insight in the problem area is obtained early enough to influence the design. This is in contrast to many other system development methods, where validation is done after the specification has been finished and often by a totally different group of people.
- The use of Standard ML as inscription language for CP-nets makes it easy to describe quite complex data structures and algorithms. In this way it is possible, gradually, to extend the initial system prototype towards a more detailed implementation.
- From the internal code used by the CPN simulator, it is possible to obtain stand-alone ML code, which can either be used for further prototyping or as the final implementation.

14.2 Introduction to Electronic Funds Transfer System

Electronic bank-to-bank funds transfer deals with the electronic movement of funds, i.e., money. In the USA this is done via two dedicated payment networks (Chips and Fedwire). The average daily transfer amounts to trillions of dollars and is constantly increasing. This has created the potential for major financial disruption triggered by the failure of a single banking house. The speed at which a transaction is processed allows banks to execute secondary and tertiary payments based upon expected incoming funds – a potential disaster if the payer fails to deliver the cash. When a computer failure at Mellon Bank required an extension of the business day to correct their systems, the Federal Reserve granted the extension, but failed to give proper notice of the situation. Many banks ended the day in debt since they expected payments which did not arrive that day. Conse-

quently, many banks had to buy settlement funds at the end of the day to cover their debts, driving the cost of overnight borrowing far above the norm and producing significant losses.

Our project was carried out in cooperation with two banks, Société Générale and Marine Midland Bank of New York. The project involved the design and implementation of new software to control the electronic transfer of money between banks. The purpose was to reduce the risks without causing too much delay. Two managers at the banks involved had an idea for a new control strategy, which would allow the relevant staff to use computer support to control the funds transfer. The two managers concretised their idea in terms of a relatively small SADT diagram which contained a rather informal description of the proposed algorithm. The constructed SADT diagram was translated to a CP-net, and more accurate behavioural information was added by an experienced CPN modeller. This was done in close cooperation with the two bank managers, who also participated in the debugging, during which the original algorithm was tested and slightly improved. The additional behavioural information could just as well have been added before the translation, i.e., by means of the SADT tool instead of the CPN editor.

The new application is used for two different purposes: to predict the debt positions as the day evolves and to determine in what sequence the payments should be executed. Since the number of payments that can be executed every hour is limited by the network capacity, the rate of flow of money can be increased by grouping together transactions with similar characteristics, such as the debit and credit parties. Also, the net result of incoming and outgoing payments with similar characteristics may be calculated, thus reducing the amount of money that travels through the networks.

The SADT specification was built by a team of four people, having different job functions at Marine Midland Bank of New York. Figure 14.1 shows the most abstract SADT view of the new software. The system is viewed as a single activity, called *Manage Intraday Debt*. Inputs are bank-to-bank transactions, called *Interday Posts* and *Intraday Posts*, together with a *Start of Day Position* for debt. Outputs are *Scripts* containing the order in which the transactions should be exe-

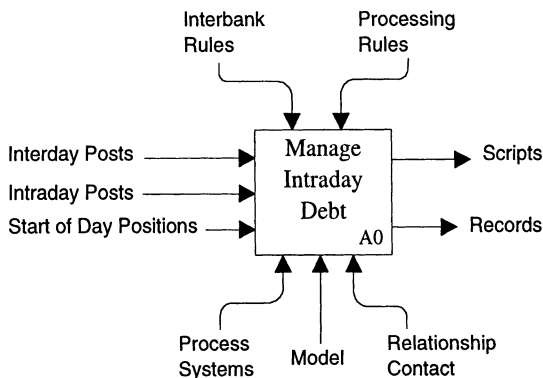


Fig. 14.1. Most abstract SADT page for electronic funds transfer system

cuted and *Records* of the transactions. Inputs are transformed into outputs under the control of *Interbank Rules* and *Processing Rules* using the mechanisms *Process Systems*, *Model*, and *Relationship Contact*. Figure 14.2 shows the refinement of the activity in Fig. 14.1. Here we find the same inputs, outputs, controls, and mechanisms. We also see that the activity has been broken down into three subactivities *Debt Management*, *Source Data Management*, and *Debt Simulation Model* interconnected by a number of arrows. Each of these subactivities are further decomposed on separate pages, leading to an SADT description containing 9 pages and a total of 32 activities.

The *Source Data Management* activity collects and pre-processes the necessary data. Then it splits the transactions into those that are already executed and those that have yet to be executed. The actual and expected positions are determined. Transactions with similar characteristics are grouped together and the net result of incoming and outgoing payments is calculated, making it possible to reduce the actual amount of money transferred via the networks.

The *Debt Simulation Model* activity orders the transactions in such a way that the average debt is minimised. Then the transactions are checked against different kinds of limits (regulating the amount of money a customer may transfer and

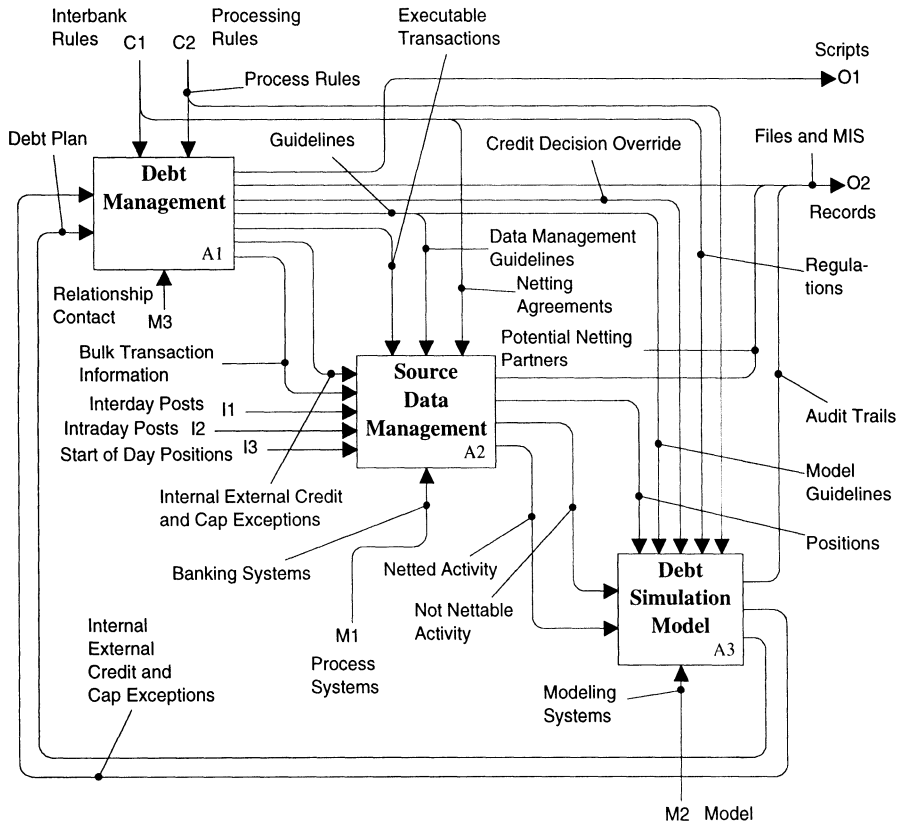


Fig. 14.2. Refinement of the SADT page in Fig. 14.1

the size of ongoing engagement between two banks/countries). When all payments have been checked a debt plan is produced.

The *DebtManagement* activity takes care of the transactions which do not pass the limits. These payments generate exceptions that can be overridden, e.g., due to an expected receivable. If an exception is overridden, the transaction is re-inserted into the list of transfers to be executed. Otherwise it is postponed. The debt plan is reviewed. If acceptable, it is turned into a script for executing transfers. Alternatively, a new debt plan may be produced by modifying some of the parameters which control, e.g., the transaction sequencing algorithm.

A more detailed description of the SADT model and the funds transfer control algorithm can be found in [42].

14.3 CPN Model of Electronic Funds Transfer System

When the SADT model had been constructed, it was transformed into a CPN model, as described in Sect. 14.1. In this process experienced CPN persons worked together with bank staff to find, clarify, and remedy inconsistencies and errors. Some changes to the SADT diagram were made, in particular to the arrow structure. It took five man-weeks to create the SADT diagram, only one man-week to get the first CPN model, and 16 man-weeks to develop this into the final CPN model.

Figure 14.3 shows the most abstract page of the CPN model, while Fig. 14.4 shows its direct subpage. To make Fig. 14.4 readable, at the chosen scale, we have hidden all colour sets. The two CPN pages correspond to the SADT pages in Figs. 14.1 and 14.2, and it can be seen that the graphical layouts of the CPN pages are very similar to the layouts of the SADT pages.

During the project there were several different versions of the CPN model. The first of these was obtained more or less directly from the SADT diagram, and it was rather crude, with simple arc expressions and very simple colour sets. This model was primarily used to describe the flow of data, while the actual data manipulations were ignored. The objective of this phase was to generate a rapid

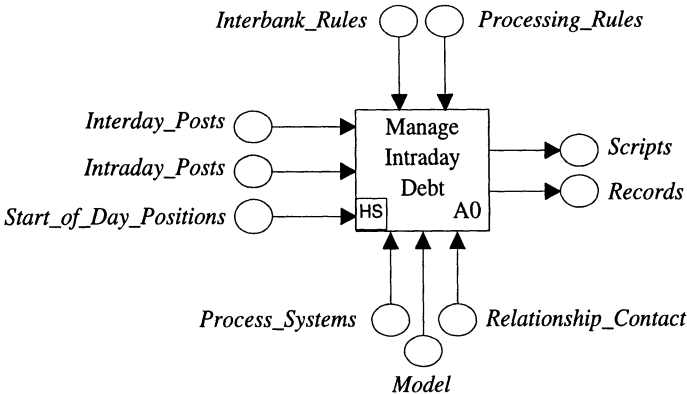


Fig. 14.3. Most abstract CPN view of electronic funds transfer system

prototype of the application in order to validate the correctness of the initial SADT specification. We made this validation as early as possible to remove possible design errors. The validation was done in a meeting with the bank managers. During the meeting, the graphical properties of the simulation tool were used to demonstrate the model behaviour in terms of data flow and token colours. Based on the meeting, the higher management in the bank decided to continue the development process.

Next, the colour sets and arc expressions were made more detailed and a large number of more complex ML functions were declared, e.g., to search, sort, and merge lists of transactions. Figure 14.5 shows some of the colour sets used in the CPN model. They tell us that a transaction is modelled as a record with five fields. The first field contains a text string for unique identification of the transaction. The remaining fields specify the payment method, the amount to be transferred, the debtor account, and the creditor account. The colour set *TransactionList* contains lists of transactions. It is used to declare a large number of structurally equivalent colour sets (of which only three are shown). As discussed in Sect. 14.1, we could have used *TransactionList* as colour set for all the places that contain a list of transactions. Then we could have retained the arrow labels in the place names (instead of the colour set names).

A typical example of a transition is shown in the upper part of Fig. 14.6. It shows how two lists of transactions, *NotNettableActivities* and *NettedActivities*, are merged and sorted into a single list of *SortedActivities*. The merging is done

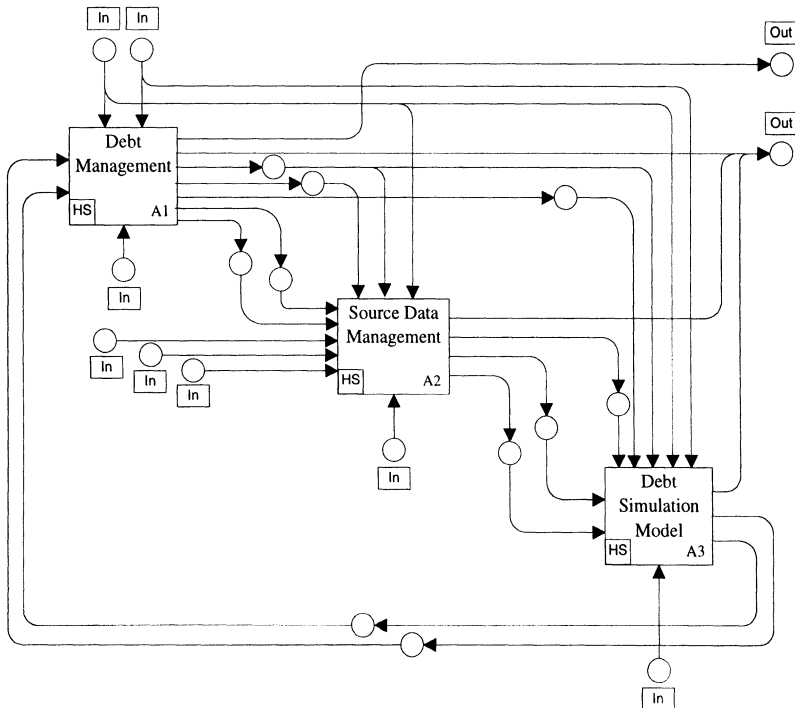


Fig. 14.4. CPN page for the SADT page in Fig. 14.2

by means of the ML operator `^^` which concatenates two lists, while the sorting is done by means of the ML function *Sort*. In the early phases of our prototyping, it was not necessary to define the detailed sorting algorithm. Our focus was on the system architecture and validation of the interaction and data flow. Hence, we declared the *Sort* function in the following, trivial way:

```
fun Sort (translist: TransactionList) = translist;
```

Later the *Sort* function was elaborated to reflect a more realistic sorting of the list. To make it easy to experiment with different sorting algorithms, we then modified the transition as shown in the middle part of Fig. 14.6. The new input/output place contains a token that specifies the desired sorting method. To change to a new method, it is sufficient to change the marking of the new place. This can be done either by means of the *Change Marking* command or when an initial configuration is read from a file.

Later on, we modified the transition so that it got the format shown in the lower part of Fig. 14.6. Each arc expression is now a single variable. All the computation logic is moved to the code segment, which calculates the values of the output tokens from the values of the input tokens. As we shall see below, this format makes it very easy to translate the CPN model into stand-alone ML code.

As soon as the different parts of the more detailed CPN model were finished, they were debugged by means of the CPN simulator. First we used the *Change Marking* and *Bind* commands to test a number of bindings for each individual transition. Then we made a number of manual simulations during which break-points were used to investigate whether markings and enablings were as expected. The focus of the simulation was to test whether the overall logic of the simulation model matched the modellers' expectations, in terms of resource sharing, synchronisation, etc. Hence, we used very simple input data. Transaction lists were short or even empty.

Now we were ready to start the detailed implementation of the new software application. At this stage we made a more detailed description of the data and the algorithms. Several databases were modelled, e.g., representing customer balances, network balances, and agreements on netting opportunities. In the earlier phases we had modelled the logic of the data flow. Now we modelled the detailed

```
color PayMethod = with Book | ChipOut | FedOut | ChipIn | FedIn;
color Transaction = record id: String *
                        pm: PayMethod *
                        am: Real *
                        DebtId: String *
                        CredId: String;
color TransactionList = list Transaction;
color Netted_Activities = TransactionList;
color Not_Nettable_Activities = TransactionList;
color Sorted_Activities = TransactionList;
```

Fig. 14.5. Excerpts of the colour set declarations for electronic funds transfer system

data and the detailed data manipulations. During this phase the amount of ML code grew to nearly 1300 lines, while the size of the net structure was unchanged. The validation of the final model was done by the bank's technical staff. This took one week, in which they performed a large number of extensive simulations. During this phase a serious design error in the SADT model was discovered. To fix the error it was necessary to modify both the SADT model and the CPN model. However, this took only three days, thanks to the flexibility and locality properties of the models.

The main difference between the prototyping phases and the implementation phase was the volume and accuracy of the ML code. In the first part of the project, the graphical interface of the tools was very important, and it was the graphical aspects of SADT and CP-nets that made it possible for the bank managers to make their ideas concrete. However, later it turned out that the graphical interface became less important, while the actual data produced by the simulations became more so. Now the simulation data became far too complex to be

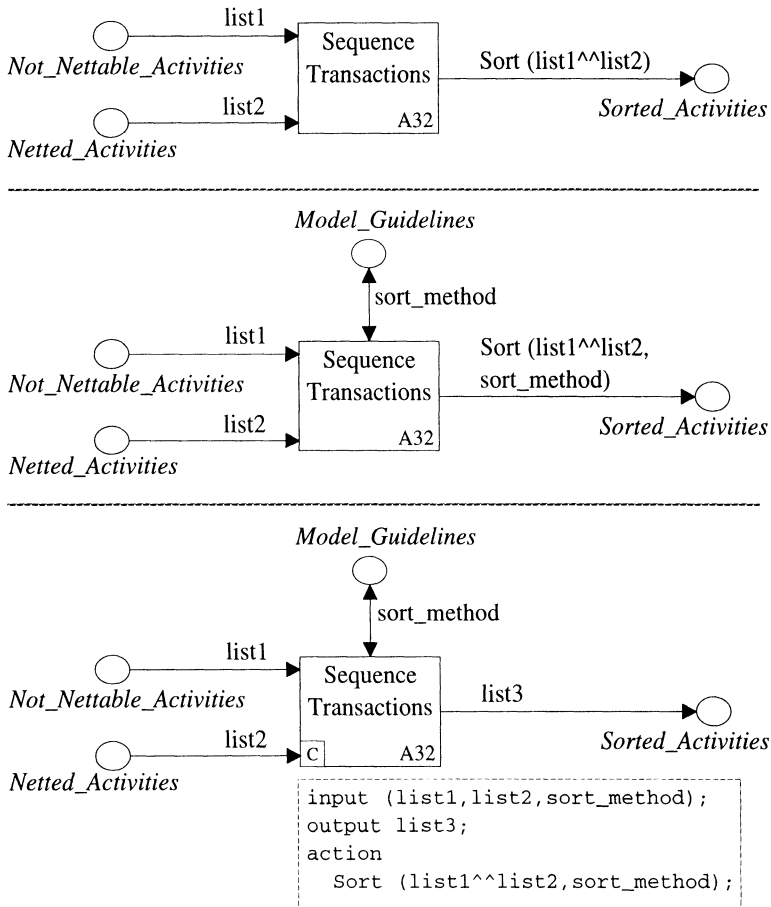


Fig. 14.6. Three different versions of a CPN transition

conveniently inspected as token colours. Instead we used files, which were read and written via code segments.

At the time of our project the CPN simulator was rather new and it did not have all the facilities it has today. A crucial problem was the fact that there was no easy way to perform fast automatic simulations. It was possible to remove the updating of the different parts of the graphics. This made the simulations run somewhat faster. However, the entire simulation process was still controlled by the process in charge of the graphical interface. This process invoked the ML engine to execute one simulation step at a time. This simulator design made automatic simulations much slower than those we make today. To circumvent this problem we transformed the CPN model into stand-alone ML code.

In the beginning we made the transformation manually. This was possible because our CPN model was rather atypical. Each transition is executed exactly once and this happens in a fixed order, which is totally independent of the input data. This makes it possible to remove the entire enabling calculation. Moreover, transitions have the form shown in the lower part of Fig. 14.6. This makes it possible to replace each transition with a simple ML statement. The ML statement for the transition *Sequence Activities* in Fig. 14.6 is shown below. Between *let* and *in* we find the variables from the input part of the code segment. Between *in* and *end* we find the action part of the code segment. The scheme also works when the transition has more than one output place and when the code segment is more complex, e.g., contains side conditions in the form of file operations.

```

val Sorted_Activities =
  let
    val list1 = Not_Nettable_Activities;
    val list2 = Netted_Activities;
    val sort_method = Model_Guidelines;
  in
    Sort(list1^^list2, sort_method)
  end

```

To obtain a stand-alone ML program, we first mapped each transition into an ML statement of the form shown above. Then we concatenated the ML statements, in the order in which the transitions were known to occur.

With the stand-alone ML code we were able to execute a much larger number of transactions. A simulation with nearly 12 000 transactions took less than 10 minutes (on a Sun Sparc work station). Of this time, three minutes were used to read all input data from files, two minutes for the processing of the transactions, and four minutes to produce output files. The latter contain detailed information, e.g., about netting, sequencing, and exceptions. Of the output time, 50 seconds were spent writing the proposed script for the next transfer and writing the updated position database.

The experiments above taught us that ML code could be executed much faster than seen in the early versions of the CPN simulator. Based on this experience the CPN simulator was enhanced so that it became capable of totally automatic generation of stand-alone ML code, which is executed without any communica-

tion with the graphical parts of the CPN simulator. This kind of simulation is today known as automatic simulation (earlier it was called super-automatic simulation). It works for all kinds of CPN models, not just for models with the simple, atypical enabling structure used in the funds transfer project.

14.4 Conclusions for Electronic Funds Transfer Project

In this project we have demonstrated that the complete software life cycle can be supported using an integrated modelling approach, based on SADT, CPN, and Standard ML. This was done by performing a case study during which we developed a software application supporting a new strategy for control of electronic funds transfer. We developed semi-automatic transformations – from SADT to CPN and from CPN to ML. The transformation from CPN to ML is now part of the CPN simulator, where it is totally automatic.

Our approach embodies several advances. Construction of executable models starts in the early specification and design phases. This means that the software developers acquire a detailed knowledge about the system to be built. A large number of errors and inconsistencies are removed at a very early stage. This reduces the cost of system development and maintenance. Changes can be tested quickly and at the desired level of detail. Enhanced applications can be generated within days.

The different phases of system development use descriptions and languages that are closely related to each other. This reduces the time and manpower needed to move from specification to design and from design to final implementation. Their close relationship also ensures more consistency between the different kinds of descriptions.

The new control strategy, proposed by the two bank managers, seemed to be working as expected. The strategy was tested on historical bank data, using the ML code produced by the CPN simulator. However, at this stage the project was discontinued, primarily because some of the key persons left the banks.

Chapter 15

Bank Courier Network

This chapter describes a project accomplished by *Valerio O. Pinci, Meta Software Corporation, Cambridge MA, USA, in cooperation with Shawmut National Cooperation, USA*. The chapter is based upon the material presented in [43]. The project was conducted in 1992.

We describe how CP-nets and the CPN tools were used to model and simulate the truck courier network of Shawmut National Corporation, which is the third largest bank in New England. The trucks feed the processing centres of the bank with checks from more than 300 branches located throughout Massachusetts, Connecticut, and Rhode Island. The objective of the modelling project is to optimise the truck delivery schedules so that checks are delivered just in time to keep the processing centres operating at maximum capacity. By using the CPN model, the bank reduced the number of trucks in the courier network and hence the cost of the overall operation.

In the project we used Structured Analysis and Design Technique (SADT) together with CP-nets. This was done in a similar way as described in Chap. 14. We used the SADT tool to obtain a work flow description of the operations of the truck courier network. Then the SADT diagram was transformed into a timed CP-net. In this way we obtained a performance model that was utilised by analysts at the bank to study the impact of changes in the truck delivery schedules on the utilisation of staff and equipment at the check processing centres. Simulation statistics were presented by means of business charts, providing an easy and straightforward way to compare the performance of different truck and staff schedules.

Section 15.1 contains an introduction to the bank courier network and describes the organisation of the project. Section 15.2 presents the CPN model of the bank courier network, including the input files and output charts. Finally, Sect. 15.3 presents a number of findings and conclusions for the project.

15.1 Introduction to Bank Courier Network

In 1992 the Shawmut National Corporation decided to launch a pilot project to evaluate the usefulness of SADT and CP-nets to obtain performance models of selected parts of their bank operations. Shawmut's interest was triggered by the successful use of this modelling and simulation technique at Canadian Imperial Bank of Commerce (CIBC). In CIBC's case a simulation model was built of the check processing data centre, which happened to be spread over two different floors in the building. The general assumption was that the elevator constituted a serious bottleneck, and hence the choice was considered of either adding a second elevator or moving all the check processing operations to the same floor. Since both of these solutions would require costly investments, the management decided to test the effects of these solutions by means of a simulation model. The results of the analysis were quite surprising. They show that the sorting operations is the main bottleneck, not the elevator. By changing the job allocations of the existing staff and making one more person available for sorting, the bank became able to process an additional 95 000 checks per day, representing a 6.5% capacity increase. Adding one more elevator would have accomplished only a 0.5% improvement of the capacity.

As a pilot project Shawmut decided to model their courier network in which armoured trucks are used to transfer checks, mail, and cash from the branches to the main encoding sites where the checks are processed. The pilot project only involved the 130 branches in Massachusetts. For this area 30 trucks are used. Each day they deliver several hundred thousand checks to the main processing centre in Boston. At peak hours the centre has more than 40 staff members on duty.

The pilot project was a success. The analysis of the simulation results allowed the bank to remove a truck and save \$35 000 a year. In a second phase the rest of the branches are being added to the model to determine whether more trucks can be saved. It is also the plan to use the CPN model to adjust the route schedules, when new branches are opened or existing branches closed.

The pilot project was completed within 12 weeks of work by one modeller/analyst who had no previous experience with SADT or CP-nets but had very extensive knowledge of the operations of the encoding centre. The modeller was assisted by senior staff from Meta Software. This assistance was provided once a week for approximately half a day. The project time was used as follows:

- three weeks to build the SADT model, including one week for review.
- seven weeks to build the CPN model, including one week for review and four weeks for instrumentation (see below).
- two weeks to simulate different scenarios, i.e., new truck and encoding staff schedules.

The instrumentation created the ML code that read truck and staff schedules from files and the ML code to update the business charts used to display the simulation results. One reason for this phase taking a relatively long time is that it required more use of the Standard ML language – with which the Shawmut

modeller had no previous experience. In the reviews different parties were involved, such as staff from Meta Software, from the branch offices, from the encoding centre, and from the courier company.

The purpose of the simulation is to measure the impact of different truck and encoding staff schedules on the performance of the encoding centre. The model does not attempt to optimise the truck routes. This is done by the courier company based on the desired schedule. The encoding equipment imprints the dollar value in machine-readable form on the check with the aid of an operator. The functioning of the encoding centre depends upon the number of checks being delivered to it (i.e., the route schedules) and it also depends upon the available amount of processing capacity (i.e., the encoding staff schedules). The system performs optimally when no more checks arrive than are needed in order to keep all the encoding staff busy. If there are too many checks at the encoding site, money is wasted paying for unnecessary truck deliveries. If there are too few checks money is wasted paying for staff being idle because there are no checks to encode.

15.2 CPN Model of Bank Courier Network

To achieve its purpose, the simulation model must have detailed knowledge of the route schedules, the number of checks waiting for pick-up, staff schedules, and the available encoding staff/equipment. This information is read from files.

A typical route schedule is shown below. It tells us that truck no. 4 is supposed to start in Boston at 7:45, visit branch B483 at 8:20, branch B488 at 9:10, etc. At 11:05 the truck returns to the main check processing centre in Boston. The complete input file contains 31 route schedules. The file was created by the Shawmut analyst, based on the existing schedules provided by the courier company.

Route	Time	Type	Location
4	7:45	Start	Boston
	8:20	Branch	B483
	9:10	Branch	B488

	11:00	Branch	B469
	11:05	To Main	Boston
	12:10	Branch	B483

A typical branch schedule is shown below. It tells us that a truck at 8:20 is expected to fetch 360 checks, no mail, and some ATM items (cash from deposits at the automatic teller machines) at branch B483. The complete input file has approximately 130 branch schedules, each representing an average day of branch activity. The file was created by means of a data base package used at the bank to

track down how many items were found at the branch when the pick-up took place.

Branch	Time	Checks	Mail	ATM
B483	8:20	360	No	Yes
	12:10	620	Yes	Yes

A typical encoding staff schedule is shown below. It tells us that the main check processing centre in Boston has no staff allocated for encoding from 11:00 to 12:00, while two persons are working with pre-encoding. In the pilot project there was only one encoding centre. Later on, two more will be added.

Location	Time	Encoding	Pre-encoding
Boston	11:00	0	2
	12:00	1	2
	13:00	4	3

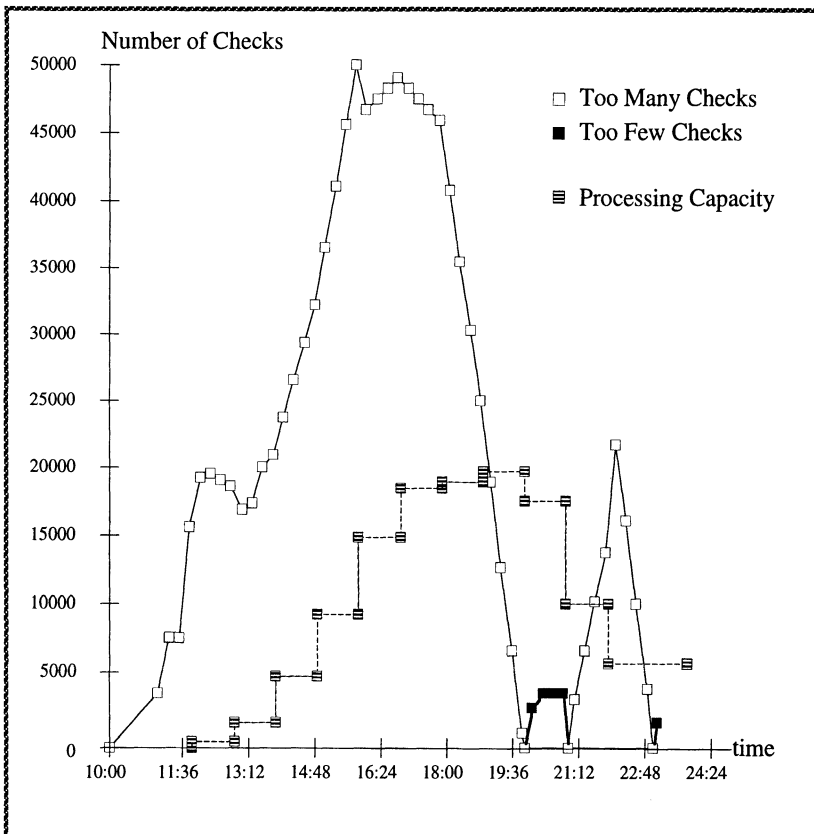


Fig. 15.1. Output chart showing the flow of checks

The simulation produces the graphical output shown in Figs. 15.1 and 15.2. The first graph shows the state of the check processing centre. The dashed curve (with zebra-striped nodes) represents the processing capacity of the encoding centre. The other curves represent the amount of check overflow (white boxes) and check underflow (black boxes). The chart shows that there is no processing capacity available until just before 12:00. In spite of this fact, checks are being delivered at the encoding centre starting shortly after 10:00 and the graph shows a large accumulation of unprocessed checks. Then the processing capacity grows, but not enough to keep up with the stream of incoming checks. Just before 18:00 the picture changes. From this point on the check overflow curve drops dramatically, telling us that there is now enough capacity to process the incoming checks and also to reduce the backlog, which finally is removed just after 19:36. Then there is more than one hour with unused processing capacity before another overflow period starts. The graph suggests that some of the early deliveries can be eliminated without reducing the efficiency of the check processing centre. Also changes in the staff allocation may improve the performance of the operation. For example, the encoding staff could be reduced earlier than 21:00, resulting in less unused capacity (and in later completion).

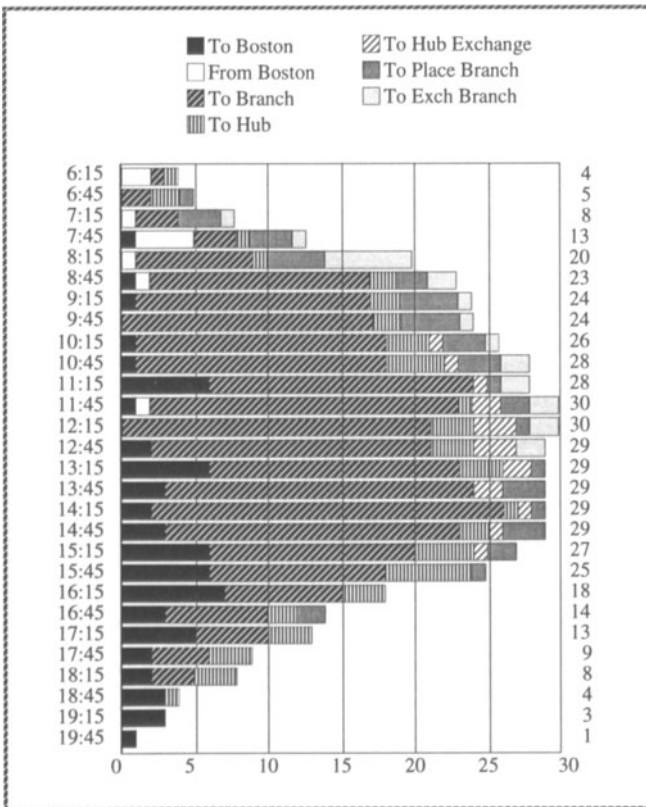


Fig. 15.2. Output chart showing the use of trucks

The second graph, Fig. 15.2, displays information about the use of trucks. At 6:15 there are a total of four trucks. Two have just left Boston, one is headed towards a branch and one towards a hub (i.e., a branch with a check sorting facility). The graph can be used to improve the route schedules. For example, one might wonder why there are trips back to Boston (i.e., to the encoding centre) long before any processing capacity is available. Looking at Fig. 15.1 gives rise to more puzzlement – there are no signs of checks being delivered so early. The explanation is that these early trips are primarily made to accommodate internal mail distribution. By investigating different route schedules, we can evaluate, e.g., how expensive this mail service is.

The first model for the courier network was created in SADT. Then the SADT model was translated into the net structure of a CP-net using the tools and techniques described in Sect. 14.1. Compared to many of the other models presented in this book, the courier network model is small and simple. The most abstract CPN page is shown in Fig. 15.3, while a more detailed view is presented in Fig. 15.4.

From Fig. 15.3 we see that the courier network removes *Sorted Mail* and *Loads* (i.e., *Checks*, *Mail*, and *ATM* items) from the branches (represented by the input places to the left) and delivers the same kinds of items to the encoding centre (represented by the output places to the right). A *Route Schedule* is used to determine in which order the branches are serviced. *Trucks* are used for the physical flow from the branches to the encoding centre. *Sorting Rooms* are used to temporarily store checks before they are taken to the encoding centre. *Trouble Reports* may be generated when for example a truck fails to meet another truck at an exchange place. Under those circumstances an *Adjustment to Schedule* must be requested from the person responsible for the operations at the courier company.

On Fig. 15.4 we find the same inputs, outputs, controls, and mechanisms (i.e., resources). They are input and output ports, but we have hidden the port tags, because they are redundant with the SADT naming conventions (where I ≈ Inputs, O ≈ Outputs, C ≈ Controls, and M ≈ Mechanisms). In Fig. 15.4

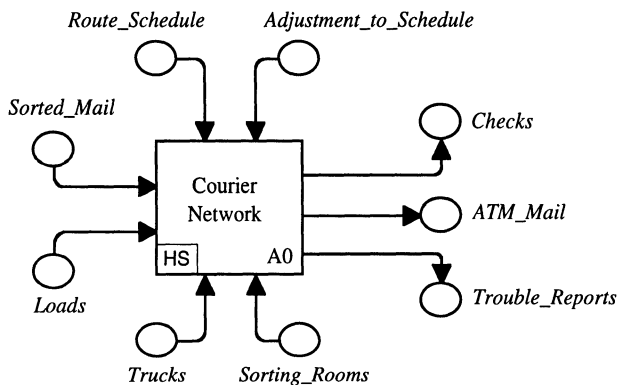


Fig. 15.3. Most abstract CPN view of bank courier network

we also see the main activities of the trucks. The *Route Schedules* determine the order in which these activities take place. Some trucks start by *Driving to Central Mail Room*. Then they *Drive to Hubs* or *Drive to Branches*, and finally they *Drive to Processing Centre*. Some of the places and arcs in Fig. 15.4 are dashed. This indicates that they were removed from the CPN model, although they existed in the original SADT model. Removing places *C2* and *O3* means that the CPN model does not generate *Trouble Reports* and does not wait for *Adjustment to Schedules*. This was not considered essential in order to capture the nature of the courier network operation. Hence, it was omitted in the pilot project. The remaining five dashed places (without names) were used to represent the various items carried by the trucks (i.e., *Checks*, *Mail*, and *ATM items*). However, as we shall see below, this information is much more adequately kept in the colours of the tokens representing the individual trucks.

Now let us consider Fig. 15.5, which shows some of the colour sets used in the CPN model. They are fairly self-explanatory. For efficiency we have chosen to model all trucks by a single token of colour *Trucks* instead of having a large number of *Truck* tokens. Analogously, we model all the possible truck loads (of checks, mail, and ATM items) by a single token of colour *Loads* instead of having a large number of *Load* tokens. With the new simulator, described in Chap. 4, it would no longer be necessary to use lists.

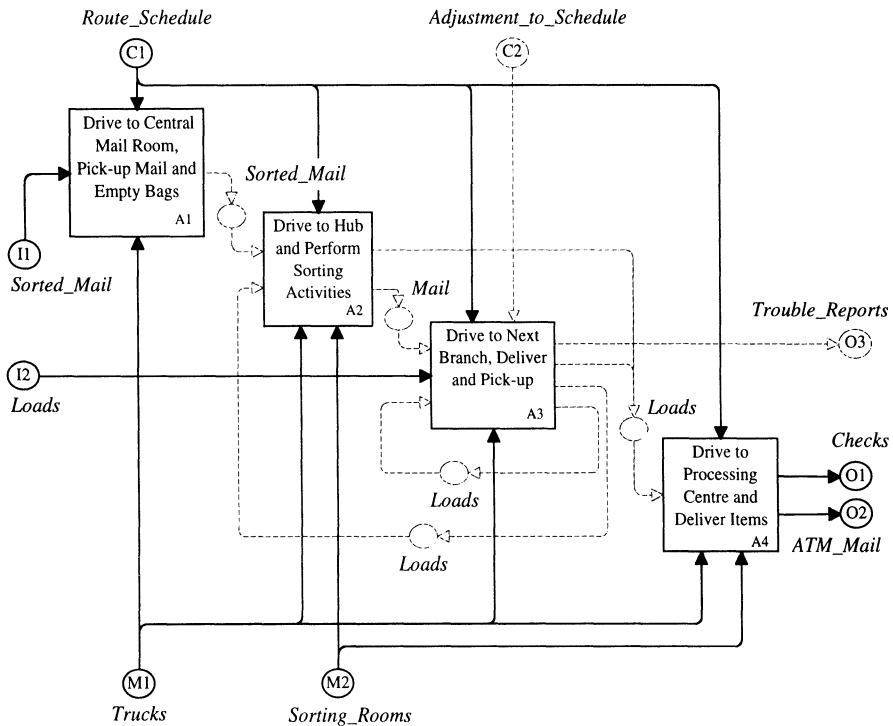


Fig. 15.4. More detailed CPN page for bank courier network

With the colour set declarations in Fig. 15.5, the route schedule from Sect. 15.2 looks as shown below (hours and minutes are converted into minutes). Notice that the time stamp indicates the time for the first stop (which in this case actually is a start). When a stop has been serviced, it is removed from the list and the time stamp of the token is updated to match the time of the next stop.

```
(4, [ {time=465, kind=Start, loc="Boston"},
      {time=500, kind=Branch, loc="B483"},
      {time=550, kind=Branch, loc="B488"}, .....]) @ 465
```

The net structure in Fig. 15.4 is automatically derived from the SADT diagram. The only thing the modeller has to do manually is to delete the dashed places and arcs (which can be done by a single editor operation). Now let us discuss how the net structure is augmented with net inscriptions, i.e., arc expressions, guards, and time expressions.

To illustrate this, we consider transition A3 in Fig. 15.4. The transition has three input arcs (from *I2*, *C1*, and *M1*) and no output arcs. The final version of the transition is shown in Fig. 15.6, and here we notice that each of the three input arcs now has a matching output arc. This is due to the fact that the CPN model is constructed in such a way that most places always contain the same number of tokens. This way of modelling may seem a bit strange for CPN people, but it is quite natural for SADT people who usually consider the inputs of an activity to be persistent material that can be used by several activities without destroying it. Now let us consider the net inscriptions in more detail:

- Place *I2* has a single token of colour set *Loads*. It is a list with an element for each branch. The element tells us how many *Checks* the branch has and whether there is any *Mail* or *ATM*.
- Place *M1* has a single token of colour set *Trucks*. It is a list with an element for each truck. The element tells us the identity of the truck, how many *Checks* the truck has, and whether there are any *Mail* and *ATM*.

```
color Kind = with ToMain | Hub | Branch | FromMain | Start;
color Location = string;
color Stop = record Time : TIME * kind : Kind * loc : Location;
color Route = list Stop;
color TruckId = int;
color Route_Schedule = product TruckId * Route timed;
color Checks = int;
color Mail = bool;
color ATM = bool;
color Truck = product TruckId * Checks * Mail * ATM;
color Trucks = list Truck;
color Load = product Location * Checks * Mail * ATM;
color Loads = list Load;
```

Fig. 15.5. Excerpts of the colour set declarations for bank courier network

- Place *C1* has a token for each operating truck. The token is of colour set *Route_Schedule*. It is a pair where the first element is a *TruckId* while the second element is a list of *Stops*. The colour set is timed. This means that each token carries a time stamp indicating the time for the next stop.

From the arc expression on the input arc from *C1*, we see that the transition is only enabled when *C1* contains a token for which the next stop is of kind *Branch*. When this is the case, the transition occurs at the time indicated by the time stamp of the token. The *rest* of the *Route_Schedule* is returned to *C1* (unless empty) with a time stamp equal to the time of the next stop (which is found by using the record selector *#Time* on the *head* of *rest*). The load of the truck involved and the load of the branch involved are determined by means of the ML functions *TruckLoad* and *BranchLoad* (in the guard) and updated by means of the ML functions *UpdateTrucks* and *UpdateLoads* (in the output arc expressions). These ML functions are straightforward. Each of them searches through a list (specified by the first parameter) and reads/updates the list item that matches a given truck/branch (specified by the second parameter).

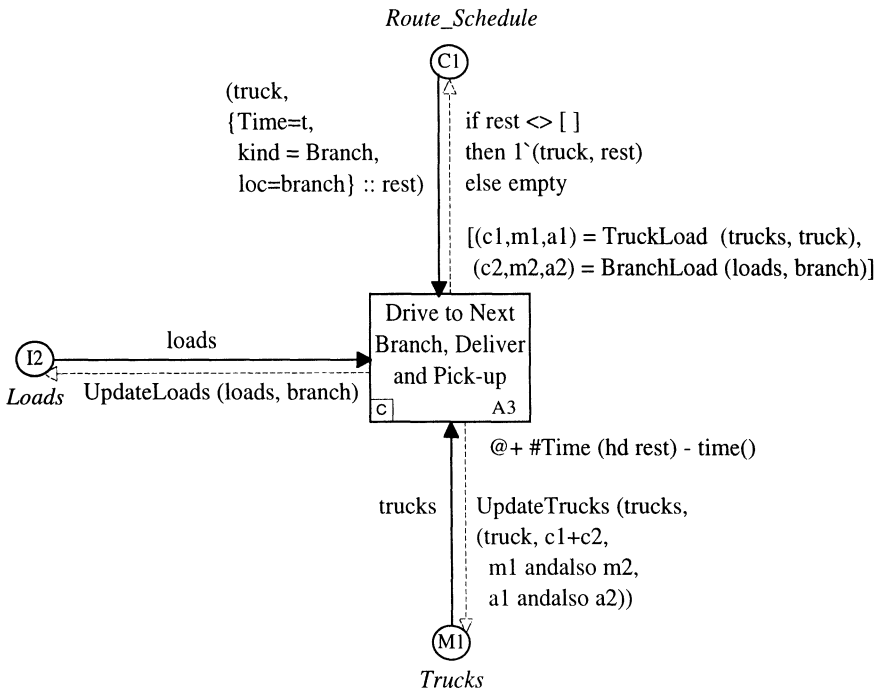


Fig. 15.6. Detailed look at one of the transitions in Fig. 15.4

15.3 Conclusions for Bank Courier Network Project

In this chapter we have presented the successful use of SADT and CP-nets for modelling and simulating a truck courier network servicing a large US bank. As a result of the pilot project the bank has improved the route schedules to obtain a \$35 000 cost reduction per year. Additional savings are expected as the model is extended to cover all branches.

As a result of the successful pilot project, the management of the bank decided to initiate a larger modelling project covering the entire encoding operations. The project is intended to analyse the impact of the new imaging technology and to optimise its use. The project group consists of three persons who work with the construction and use of SADT and CPN models. At the same time, more opportunities for modelling and simulation are being identified in the retail branch network.

We have performed a number of projects for commercial customers, such as Shawmut Corporation and Canadian Imperial Bank of Commerce. Based on these experiences, we believe that the combined use of SADT and CP-nets provides a useful and efficient platform for business reengineering, i.e., optimisation of work flows, courier networks, etc. SADT provides easy-to-do and easy-to-read process descriptions in a graphical language which is already familiar to many system analysts. The SADT models are translated into CPN models and in this way the analysts get access to the simulation power of CP-nets. In our project the net structure was obtained automatically, while the net inscriptions were added manually.

Chapter 16

Network Management System

This chapter describes a project accomplished by *Søren Christensen, Aarhus University, Denmark, and Leif O. Jepsen, RC International A/S, Aarhus, Denmark*. The chapter is based upon the material presented in [13]. The project was conducted in 1990.

We present a project where CP-nets were used for the detailed design and specification of a software module for the network management system of a European wide area network. The module was designed using the CPN tools, in particular the CPN editor and the CPN simulator. Furthermore, place invariant techniques were used to verify properties of the software module.

The resulting CPN model was used as basis for the implementation of the software module. The implementation language was a variant of Pascal called Real Time Pascal. It includes facilities for concurrent processes, mailbox handling, and buffers for communication between processes. The implementation and test phases followed standard development procedures and the module is now a running component of the network nodes.

The model was also used to evaluate the design proposal. An experienced developer without knowledge of CP-nets had less than an hour of informal introduction. After this he was able to understand the CP-net model and to give qualified feedback in the form of proposals for changes to the model.

The use of CP-nets and the CPN tools was a success. The implementation of the module was fast, it was easy to extend it afterwards, and only a few bugs were found in the test phase. The use of CP-nets in the design phase contributed to the development of a better product using fewer resources.

Section 16.1 contains an introduction to the network management system and the project organisation. Section 16.2 presents the CPN model of the network management system. Section 16.3 discusses how we used simulation and place invariants to validate and verify our design model. Finally, Sect. 16.4 presents a number of findings and conclusions for the project.

16.1 Introduction to Network Management System

The project was part of a large software development project carried out by RC International. The aim of the total project was to develop the RcPAX X.25 wide area network to provide the International X.25 Infrastructure Service which spans 19 countries in Europe and connects 20 private and 11 public X.25 networks.

The RcPAX network consists of a number of network nodes handling access traffic to/from users of the network and transit traffic internally in the network. The Network Management System (NMS) enables operators at the Network Management Centre (NMC) to monitor and control all modules of the total network. The NMS is a distributed application which is an integrated part of all network nodes. Figure 16.1 shows a schematic overview of the network.

Each network node has a software module that handles the communication between the NMC and the different software modules local to the network node. At the basic level, the NMS works with three kinds of messages: the NMC can issue a *request* to a specific software module, the request will trigger an *answer* which is sent back to the NMC, and finally information on *events* at the network nodes will be sent to the NMC.

The structure of the individual network nodes is shown in Fig. 16.2. Each node is a single machine with a management board and a number of transputer boards. The latter are used for all the software responsible for access and transit traffic. The local management system represents each software module as a Local Control Probe (LCP). The management board runs the Network Control Probe (NCP) which is responsible for communication to the NMC. The function of the Sub NCP (SNCP) and the LCP adaptor is to connect the LCPs with the NCP across a local bus, called the Link Bus.

CP-nets were used to design the SNCP module. A detailed model of the intended behaviour of the SNCP was made. More rudimentary descriptions of the

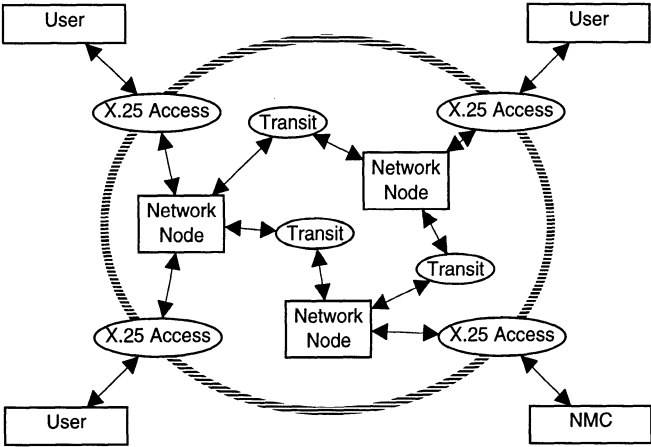


Fig. 16.1. Simplified overview of the network

behaviour of the LCPs and the LCP adaptor were added as an environment for simulating the behaviour of the SNCP module. The complete development of the SNCP module was accomplished in four phases using the following amount of man-weeks:

- Analysis: 4
- Design: 2 + 2
- Implementation: 2
- Testing: 2

The analysis phase was performed without the use of CP-nets. It produced a conventional textual specification of the protocol connecting the LCPs with the SNCP. The set of protocol services includes *connect*, *disconnect*, *send event*, *receive request*, and *send answer*. A similar textual specification of another protocol connecting the SNCP with the LCP adaptor was made in another subproject.

For each service in the two protocols the typical use was illustrated in a message sequence chart like the one shown in Fig. 16.3. It describes how a new LCP informs the NMC about its existence. First the LCP sends a *connect lcp* to the SNCP which sends an *lcp init* to the LCP adaptor. The reply is either an *lcp*

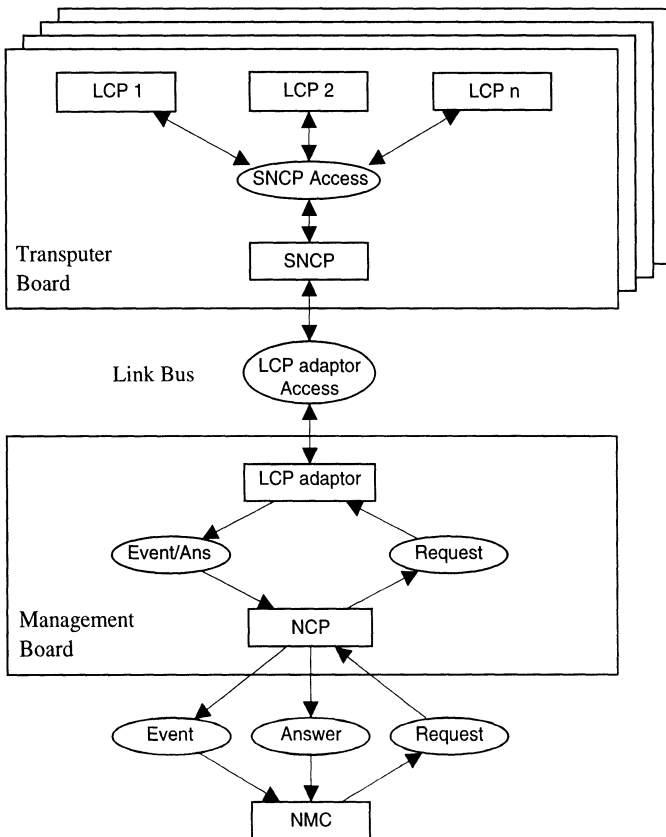


Fig. 16.2. Structure of a single network node

init not done or an *lcp init done*. It is important to notice that a message sequence chart only specifies a single typical sequence of protocol events. If error handling should be documented, it would be necessary to create a significant number of additional message sequence charts. It is also difficult to describe how interleaved event sequences are handled, e.g., the situation where a request for the LCP is received before the *connect lcp* answer is returned.

The task of the next phase was to design the SNCP module according to the specification of the two protocols. Although the functions of the module were well understood, this was a complex task:

- The SNCP should handle the communication with the LCP adaptor on another CPU board (including retransmission of lost messages, acknowledgements, etc.).
- The SNCP should handle all the LCPs and the LCP adaptor in parallel.
- Error situations and their corresponding actions should be identified.

The complexity of the design problem implied a need to work on the control structure and the internal state of the SNCP module – without going into too much implementation detail. This was the original motivation for the use of CP-nets. The CPN editor was used to develop the detailed design of the control structure and the internal state of the SNCP module. Furthermore, a rudimentary description of the surrounding components was added to provide an environment making it possible to evaluate both the internal and external behaviour of the SNCP.

The project was carried out by two persons. One of these, the modeller, was a software developer at RC International. He was responsible for the development of the SNCP module, and he took the initiative to use CP-nets, but had no prior experience of using Petri nets for modelling. The other person was an experienced user of the CPN tools. His primary task was to assist the modeller using the tool and to discuss how CP-nets could be best used to model the system. Learning to use the tool was part of the design phase. We estimate that this took two of the four man-weeks used in this phase.

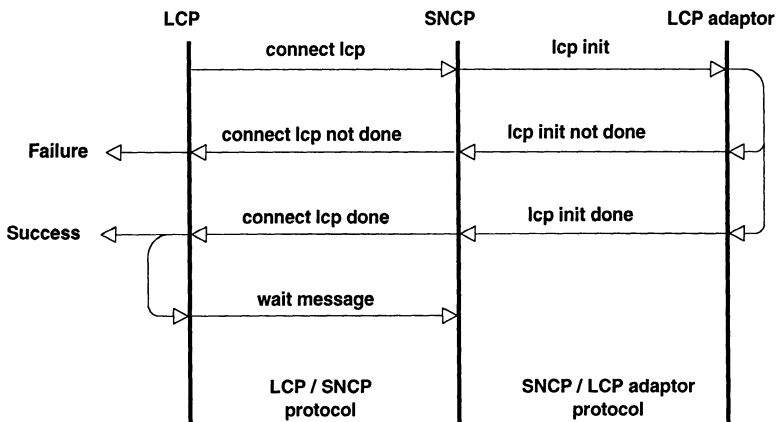


Fig. 16.3. Message sequence chart specifying a typical sequence of messages

16.2 CPN Model of Network Management System

The aim of the modelling phase was to design the control structure and internal status information of the SNCP module. The starting point of the design was the textual protocol specifications made in the analysis phase.

We modelled the system in a top-down manner. The most abstract level of the CPN model is shown in Fig. 16.4. It illustrates the different hardware compo-

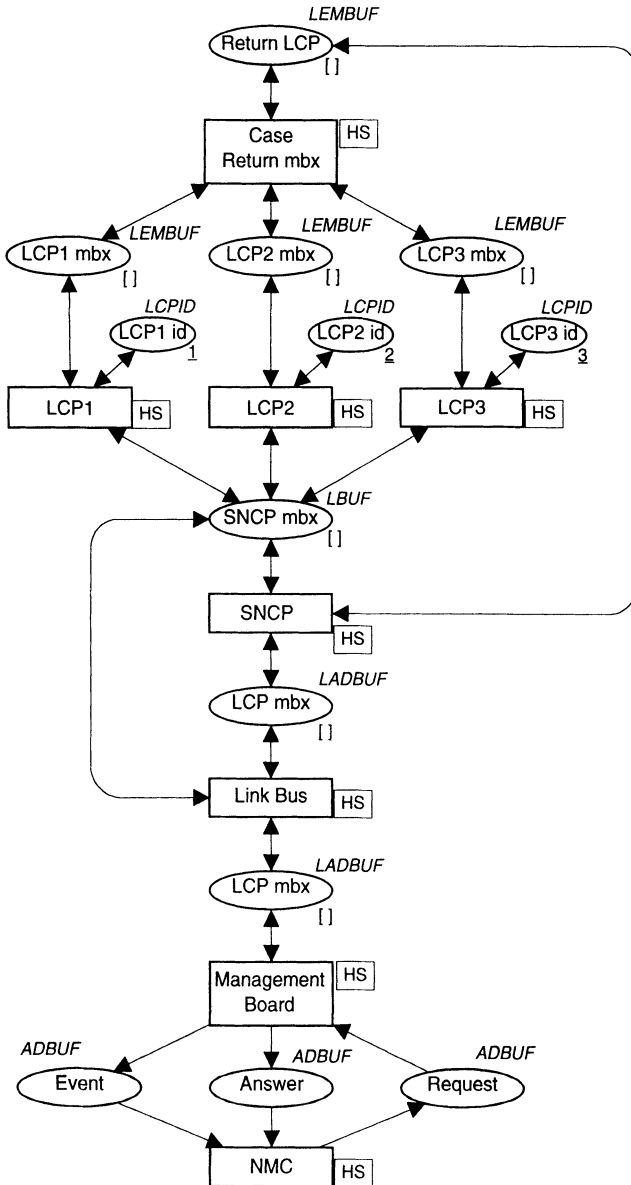


Fig. 16.4. Most abstract CPN view of network management system

nents of a single network node, and hence it resembles Fig. 16.2. However, there are a few differences:

- The Link Bus is now explicitly represented as an active component. This is necessary because we want to be able to model transmission errors between the management board and the transputer boards.
- When modelling the environment of the *SNCP*, we did not need to distinguish between the *LCP adaptor* and the *NCP*. Hence, we have represented the *Management Board* by a single substitution transition.
- The *LCPs* are modelled by three substitution transitions: *LCP1*, *LCP2*, and *LCP3*. They all use the same subpage, and each of them has a socket place with a token representing the identity of the *LCP*.
- A number of mailboxes have been added. This is done because we knew that the implementation would be done by means of a language that supports communication by means of mailboxes.

Since our main interest was to design the *SNCP*, we started out by describing this module. The most abstract view is shown in Fig. 16.5. It has one ordinary transition *Initialise* and two substitution transitions *Get Next Operation* and *Process Operation*. Thick arcs are used for control structure while thin arcs indicate data access.

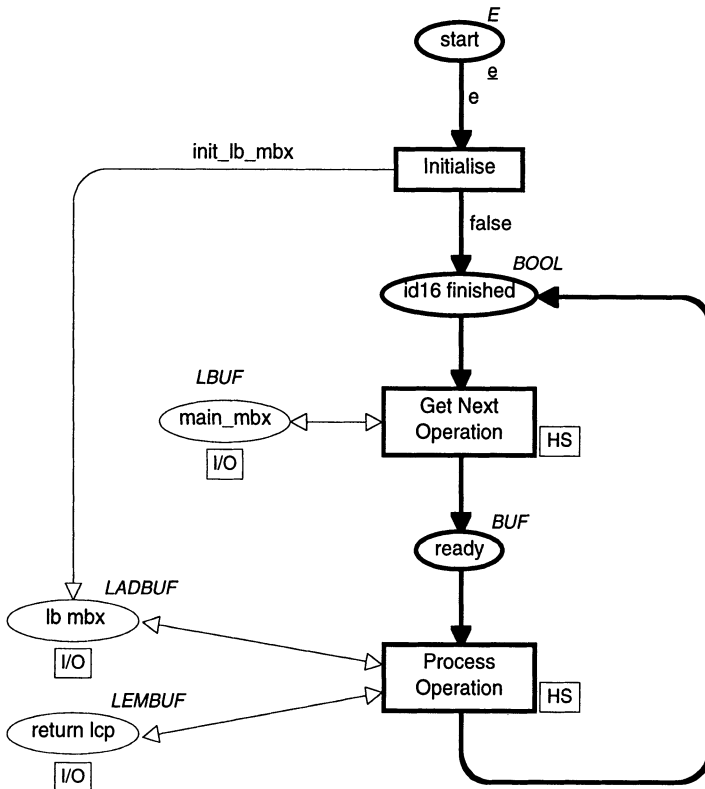


Fig. 16.5. Most abstract CPN page for the *SNCP* process

The subpage of *GetNextOperation* is shown in Fig. 16.6. For readability, we have included the declarations of the colour sets (they would usually be collected in a declaration node). When a token arrives at place *id16 finished*, the colour of it specifies whether the internal mailbox *id16mbx* should be checked or not. The check is done by the transition *Checkid16* which inspects the list *lb*. If the list is empty an e-token is positioned on place *wait main*. Otherwise an e-token is positioned on place *wait id16*. This means that transition *Next Waiting Operation* will only occur when at least one buffer is present in *id16 mbx*. The other mailbox *main mbx* is not checked before it is accessed. This means that the system may have to perform an idle wait (with a token in *wait main*) until a buffer arrives at *main mbx*.

From the initial specification of the two protocols it was known what information the individual protocol messages should contain. From this it was easy to declare the colour set *BUF* shown in the lower right part of Fig. 16.6. Note that

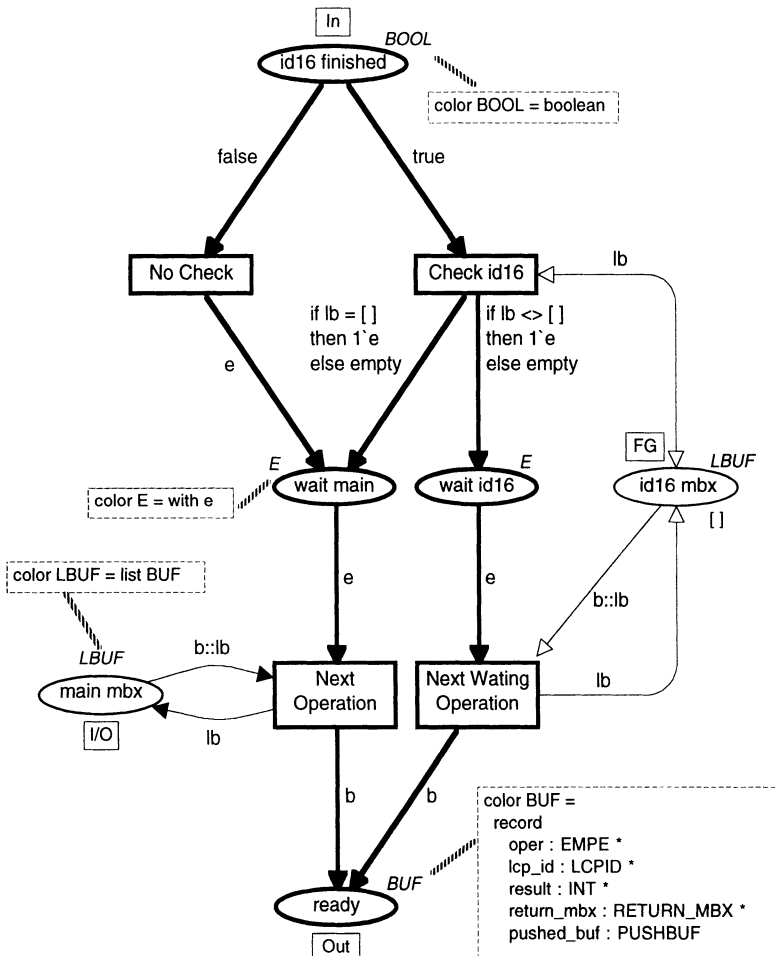


Fig. 16.6. CPN page for *GetNextOperation*

the CPN representation of a message buffer is more abstract than in the initial specification. The colour set simply describes the different kinds of data needed, without specifying how the individual data elements are located in the message buffer. A message buffer is a record with five different fields, of which the first field specifies the operation type, while the other four fields specify different kinds of message data, e.g., the identity of the LCP involved.

The page hierarchy of the CPN model is shown in Fig. 16.7. From this it can be seen that the page representing *Process Operation* contains twelve substitution transitions, one for each operation in the SNCP module. The CPN tools allowed us to model and simulate each of these operations in full detail before the rest of

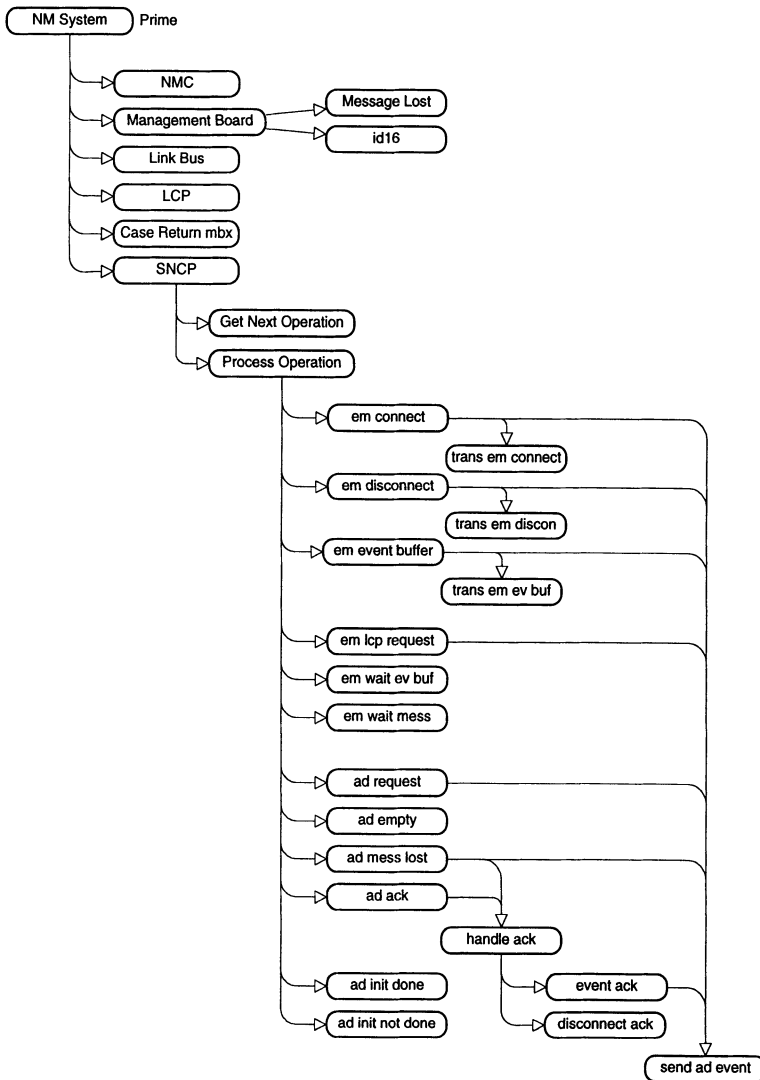


Fig. 16.7. Page hierarchy for network management system

the design was done. The first operation we designed was *connect lcp*, which is represented by the page *em connect* (operations in the SLCP/SNCP protocol are prefixed with *em*, while operations in the other protocol are prefixed with *ad*). The experience from modelling and simulation of the *em connect* page was then used in the design of the remaining operations. We have structured the CPN model in such a way that each page could be implemented later as a separate process/procedure.

The CPN model allowed us to analyse the data needed to implement the SNCP module and to evaluate whether the data should be local or global. When we started the modelling we expected to have relatively complex information on the internal state of the individual LCPs, but during the modelling it turned out that much less information was needed. Thus the explicit modelling of internal state information led to a simpler representation in the final program.

The CPN model was used as basis for the implementation. It turned out to be rather straightforward to transform a CPN page into a process/procedure of the implementation language. To see how this was done consider Fig. 1.8, which shows the Real Time Pascal code that implements the CPN page in Fig. 16.6.

In our case, one of the details we omitted was the concrete representation of buffers and data in the implementation. This meant that we did not model the different aspects of data conversion between the data representation of the transputer boards and the management board. We could omit these details without losing information necessary for designing the program structure and the internal state of the SNCP module.

```

if id16.finished
then
  if open(id16.waiting_em)
  then
    wait(next_opr, id16.waiting_em)
  else
    wait(next_opr, main_mbx^)
else
  wait(next_opr, main_mbx^)
```

Fig. 16.8. Implementation of *GetNextOperation*

16.3 Validation of Network Management System

To be able to perform realistic simulations of the CPN model, we included an environment with a rudimentary description of the LCPs, the LCP adaptor and the NMC. It is essential that the description of the environment can be much more rudimentary and fragmented than the rest of the model. Otherwise it would not be possible, in practice, to simulate models of systems that have a complex environment.

Simulation is a very efficient way to debug a model. It is a powerful way to gain insight in the behaviour of the system and it greatly enhances the chance of obtaining a consistent and error-free model/specification. The CPN simulator provides the same kind of facilities as a good debugger:

- During the simulation, it is possible to investigate all details of the model, e.g., the current state and the possible actions.
- The user can make an interactive simulation in which he single-steps through an interesting action sequence. He can observe the possible conflicts/choices and resolve them as he wants.
- The user can make fast automatic runs in which the actions that occur are randomly selected.
- Interactive and automatic simulations can be mixed.

The CPN simulator makes it possible to follow the simulation directly on the graphical representation of the CPN model. The states of the various system parts are shown as tokens and the possible actions are shown as enabled transitions. The locality of the CPN transitions makes it easy to see why a transition is enabled and what effect it will have. This is in contrast to simulation/debugging of a program written in a textual programming language. There it is often cumbersome to follow the details of an execution, because the representation of data is separated from the representation of program instructions.

During the design process, different kinds of simulations were performed. At first only a small submodel was available and we were in full control of the simulation – we chose the transitions we wanted to occur next. Later we started to simulate more automatically, triggering an automatic run by specifying an input message from an LCP and/or from the NMC. The results were compared with the message sequence charts from the initial protocol specification (such as Fig. 16.3).

During the simulations we discovered a number of errors in our model, and especially at first we had to remodel parts of the system. The errors can be divided into the following categories:

- Erroneous or insufficient tests before services were performed,
- Disappearing buffers,
- Parts of the model not being specified in sufficient detail,
- References to missing parts of the model.

The simulations allowed us to remove a considerable number of bugs before the implementation started. Moreover, the simulations gave us a much more detailed insight into the dynamics of the system. This allowed us to improve and simplify the design. Similar results can often be obtained by means of prototypes. However, this project involved both hardware and software development, and hence it would be difficult to build a prototype until late in the project. Building an early CPN model was much easier and more sensible, because we could evaluate the behaviour of the designed software using a crude and abstract description of the hardware with which it was supposed to work.

When our project took place, there was no tool support for verification of complex CP-nets. Hence, we were unable to use occurrence graphs. However, we did verify a few dynamic properties by means of manual place invariant analysis.

An important property of the management of the buffers in our system is that they do not disappear. This was expressed as a place invariant. For each place representing buffers, we used a weight that mapped each buffer into an e-token. This was slightly more complex than it sounds, because our model contains lists of buffers and also buffers which are “pushed” on top of other buffers. For all non-buffer places, the tokens were ignored (i.e., mapped into the empty multi-set). By manually inspecting the individual transitions, we convinced ourselves that the invariant was satisfied, i.e., that no buffers were created or disappeared.

We also used place invariants to prove that the system contained all the information needed to re-establish the contents of a buffer, if the contents got lost during transmission at the Link Bus. To prove this property of the system, it was necessary to make small extensions to the CPN model. The extensions allowed us to compare the information that was lost with the information that was re-established.

Our verification of the two properties mentioned above would have been faster and much more reliable, had we been able to use the kind of invariant checking tool described in Sect. 4.4 of Vol. 2. However, this tool did not exist at that time. The right tool support would also have made it possible to use the verification as an integrated part of the modelling, instead of using it when the model was finished. Early use of place invariants could have been a valuable supplement to the simulations we performed.

16.4 Conclusions for Network Management Project

The use of CP-nets to design the SNCP software module was a success. The implementation of the module was fast, it was easy to extend it afterwards, and only a few bugs were found in the test phase. The use of CP-nets in the design phase contributed to the development of a better product using fewer resources.

It is hard to test programs distributed on special-purpose hardware in a wide area network. It is difficult to create adequate debugging environments and this makes it even more important to validate and verify the detailed design of new programs. An alternative of using CP-nets for modelling and simulation could be a prototyping approach. But in our case the module should exist in an environment where different pieces of hardware and software are being developed in parallel to the module. Hence prototyping would be difficult and could only be used late in the development process.

If we compare the project to experiences from similar projects, it seems that the design phase was slightly more time-consuming while the implementation was faster.

Chapter 17

Naval Vessel

This chapter describes a project accomplished by *Jean Berger and Luc Lamontagne, Defence Research Establishment Valcartier, Quebec, Canada*. The chapter is based upon the material presented in [6]. The project was conducted in 1992.

Recently, Petri-net technology has been used to tackle different aspects of command and control problems. In most cases, the objective was to assess and compare organizational structures, and various cooperation and coordination mechanisms. Our aim is slightly different. We describe how CP-nets and the CPN tools were used to model and investigate a conceptual naval vessel. The CPN model represents the behaviour and real-time aspects of critical system components. It is used to assess the performance of different decision policies for weapon assignment. We explore the feasibility of CP-nets for providing a proper framework and a suitable simulation environment to model command and control systems and study related centralised decision-making strategies. An evolutionary approach, in which we iteratively refined the granularity of the model for each system component, has been followed. We apply modular development, independent subnet testing, partial and progressive integration, and simulation. The hierarchical structure of CPN models played a key role in the whole system development process, supporting an incremental “build a little, test a little” approach.

Section 17.1 contains an introduction to the naval vessel. Section 17.2 presents the CPN model of the naval vessel. Section 17.3 describes the simulation results for two different weapon assignment policies. Finally, Sect. 17.4 presents a number of findings and conclusions for the project. Here we compare the CPN approach with an object-oriented simulation environment developed at our research centre.

17.1 Introduction to Naval Vessel

The main components of the naval command and control system are shown in Fig. 17.1. The system has seven main entities: threats, sensors, weapons, fire control radars, track manager, action manager, and battle manager. The arrows indicate different kinds of messages being passed between the various entities. Some of the message labels are self-explanatory while others have a more technical meaning – which hopefully will become clearer when we consider some of the pages of the CPN model.

The *Threats* represent hostile anti-ship missiles. Each missile is assumed to have a deterministic behaviour, i.e., to follow a predetermined trajectory. *Threats* are periodically detected by *Sensors* responsible for data gathering and track generation. The information is sent to the *Track Manager* which is responsible for track processing and data fusion, e.g., integration of data from multiple sensors. The number of tracks obtained corresponds to the number of *Threats* observed. When a new set of track data is obtained, a new Track Evaluation and Weapon Assignment (TEWA) cycle is started.

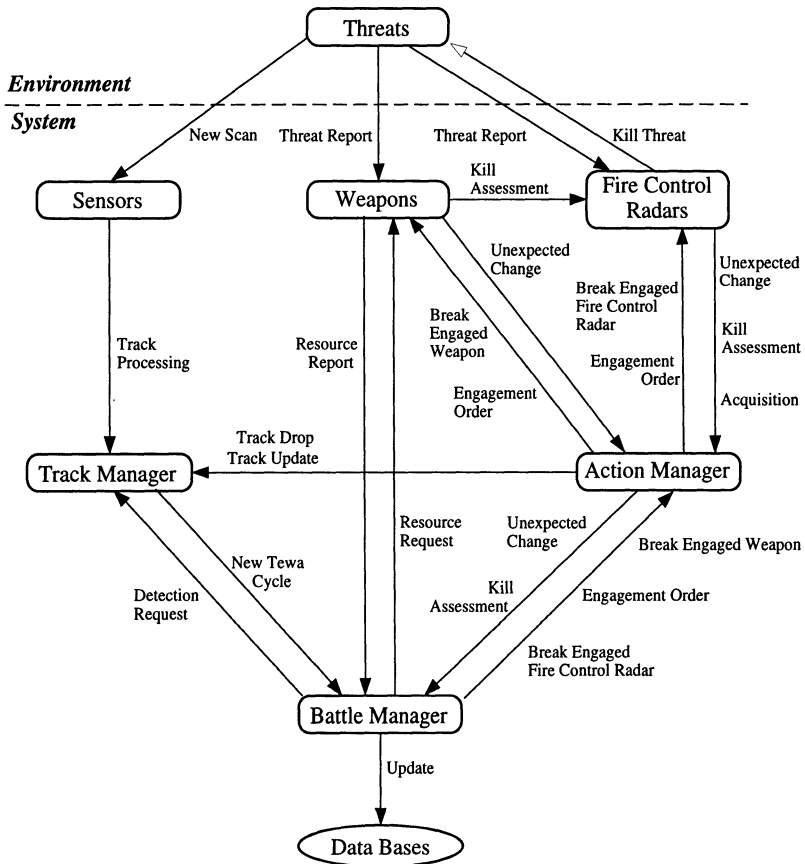


Fig. 17.1. Main components of the naval command and control system

The *Battle Manager* represents the command and control decision-making process responsible for the overall planning task. This process evaluates the threat level and determines suitable responses to attacks by selecting the necessary *Weapons* and *Fire Control Radars*. As a result, orders are sent to the *Action Manager* which executes them and provides the *Battle Manager* with feedback information as the situation unfolds. The *Action Manager* supervises the various tasks leading to the actual weapon deployment. The actual plan execution is performed by *Weapons* and by *Fire Control Radars*. They provide the *Action Manager* with a feedback, e.g., in the form of a kill assessment.

17.2 CPN Model of Naval Vessel

In this section we present the timed CP model of the defence system in Fig. 17.1. The most abstract view of the CPN model is shown in Fig. 17.2. It has a substitution transition for each of the seven main system entities. The *Projectile/Missile* node in the upper right corner is an auxiliary node (and has no colour set). This indicates that the actual deployment of missiles towards the *Threats* is not part of the CPN model. Altogether the CPN model consists of 30 pages:

- One page for declarations, two pages for initialisation and termination, and one page for the overview presented in Fig. 17.2.
- Two pages for *Threats*, one page for *Sensors*, one page for *TrackManager*, five pages for *Weapons*, six pages for *Fire Control Radars*, three pages for *AttackManager*, and eight pages for *Battle Manager*.

Figures 17.3–17.6 show the most abstract pages for *Sensors*, *TrackManager*, *Battle Manager*, and *Weapons*, respectively. Most of the other pages have a similar layout and a similar level of detail. More information about the CPN model can be found in [5] and [22].

To make the CPN model more readable, each page is divided into three parts. The left-hand part contains places used to represent the different states (i.e., modes) of the modelled entity. As an example, the three places in the upper left part of Fig. 17.3 tell us that the hardware component of the *Sensors* entity can be in three different modes: *Idle*, *Emitting*, and *Receiving*. The central part of the pages contains the actions to be executed by the entity, and the right-hand part contains the port places through which inputs are received and outputs delivered. The places in the central and right-hand parts use three different kinds of lines. Places with a full, unbroken line represent entity data while those with a dashed line represent messages. Finally, there are a few places with dotted lines. They are used for simulation management and conflict resolution. As usual, FG-tags identify places that are members of global fusion sets, while In-, Out-, and I/O-tags identify port places.

Transitions represent the actions of the various entities. Time delays are specified to describe the duration of actions (see, e.g., the arc expression in the lower left corner of the *Sensing Mode* box in Fig. 17.3). As usual, C-tags indicate the existence of a code segment, while HS-tags identify substitution transitions.

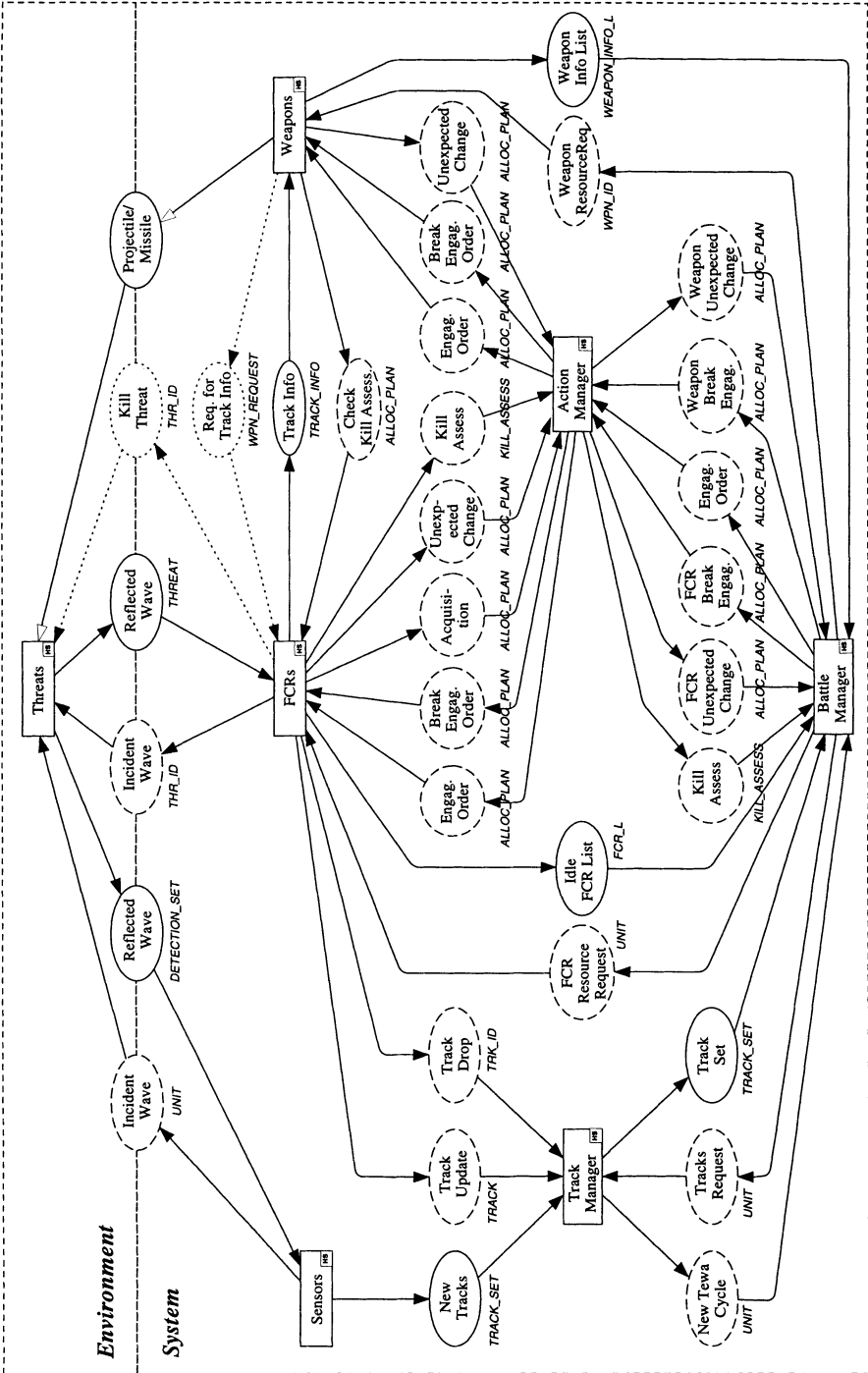


Fig. 17.2. Most abstract CPN view of naval vessel

A simulation is initiated according to a predetermined air-threats scenario and a certain set of system parameters, e.g., specifying the desired weapon assignment policy. Termination occurs when the air-threats are destroyed or the attack finished. Below we give a brief description of each of the four CPN pages in Figs. 17.3–17.6.

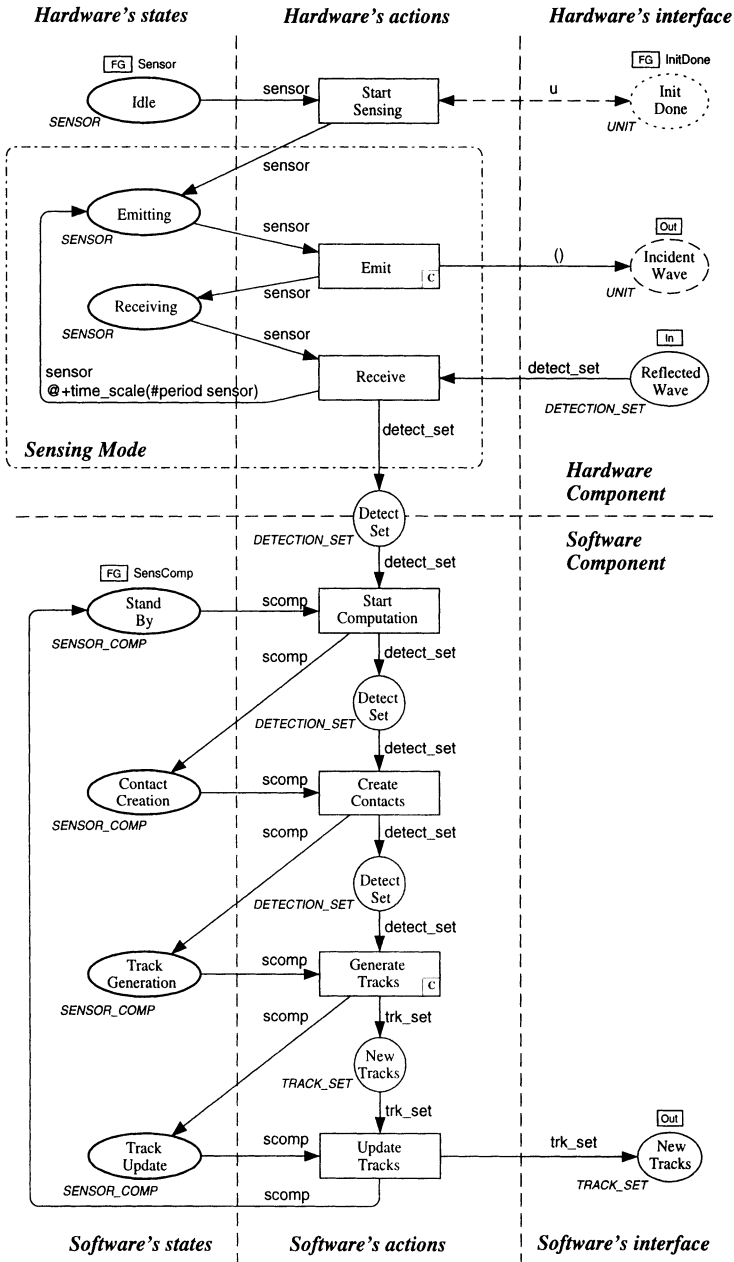


Fig. 17.3. CPN page for Sensors

Page *Sensors* in Fig. 17.3 represents interacting hardware and software components. The upper part of the diagram models the hardware. It accounts for the scan process. The entity state is either *Idle*, *Emitting*, or *Receiving*. The interactions with the *Threats* and the *TrackManager* are depicted in the right-hand part. When a scan cycle is completed, detection is achieved through the software component represented by the lower part of the CPN page. A token initially located

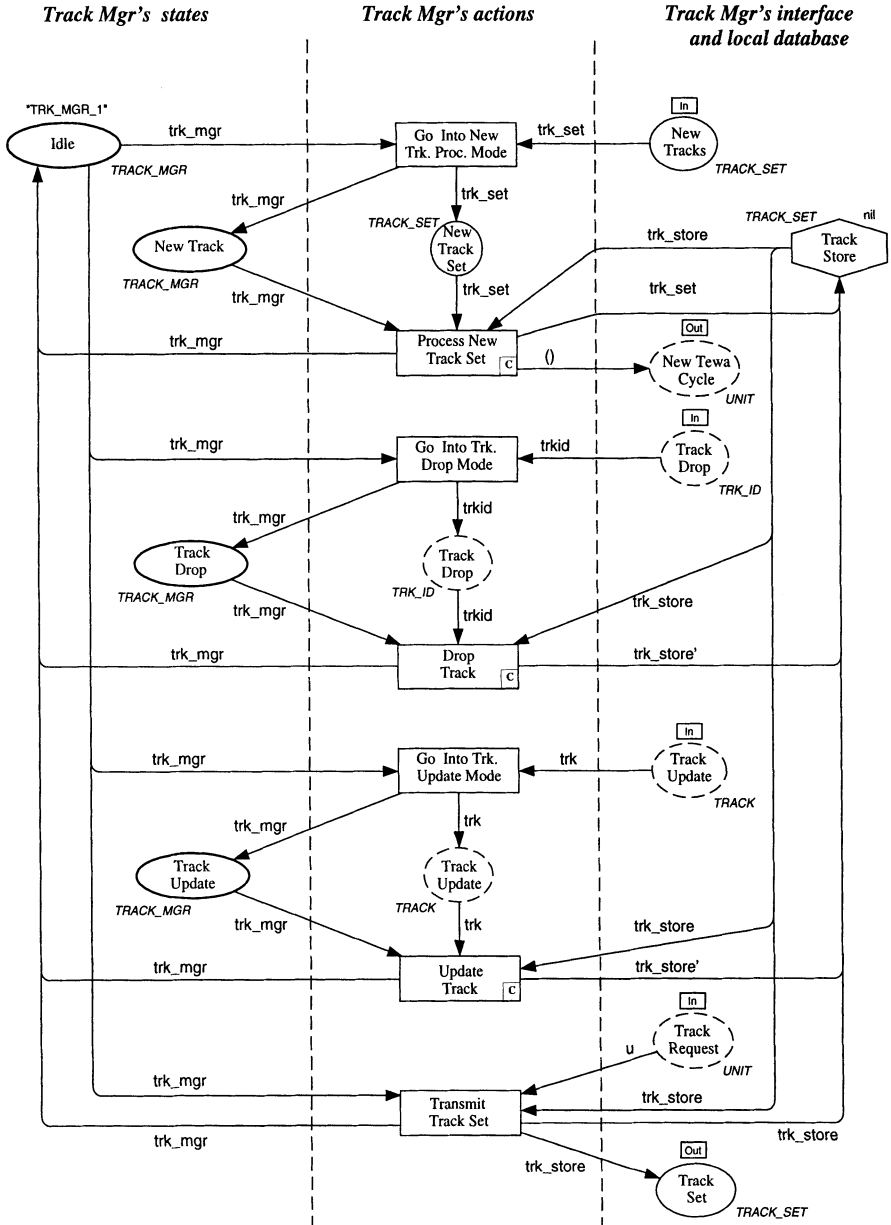


Fig. 17.4. CPN page for *TrackManager*

in the *Stand By* place moves progressively downwards, enabling transitions to *Start Computation*, *Create Contacts*, *Generate Tracks*, and *Update Tracks*. Finally, a set of *NewTracks* are sent to the *TrackManager* via the output port in the lower right corner. The software component then returns to *Stand By*.

Page *TrackManager* in Fig. 17.4 is initiated when the token in the left-hand part moves from *Idle* to one of the three other states. The various actions carried out by the *TrackManager* are shown in the central part of the diagram. A task

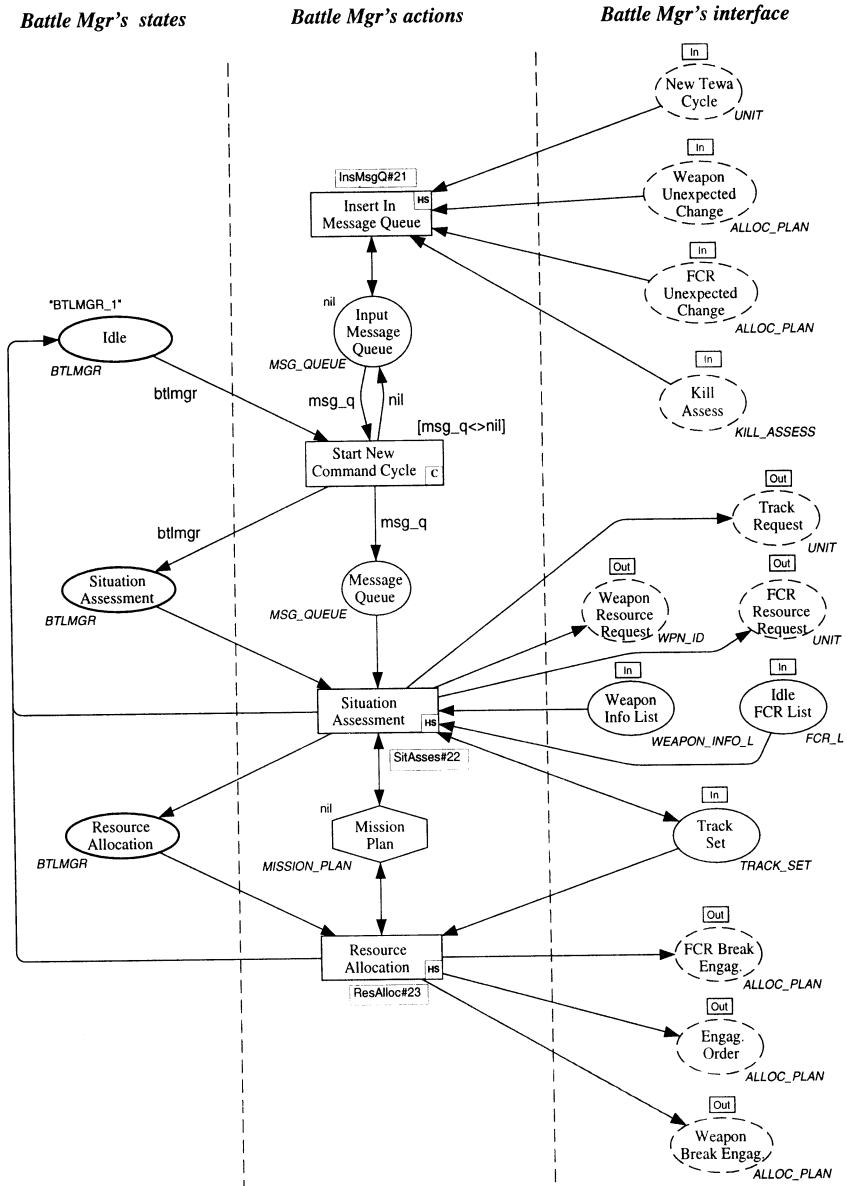


Fig. 17.5. CPN page for Battle Manager

consists in handling sensor data or external track requests from other system entities. The transition at the bottom of the CP-net *Transmits a Track Set* to the *BattleManager*, which has made a *Track Request* – after being notified of the finishing of a *NewTewa Cycle* by transition *ProcessNewTrackSet*.

Page *BattleManager* in Fig. 17.5 presents the states and actions required to carry out target evaluation and weapon assignment. The details of the actions are described at the subpages of the three substitution transitions. Four different kinds of input messages are received via the four input port places in the upper right part. They are *Inserted in Message Queue* by the uppermost transition. Then the mode changes from *Idle* to *Situation Assessment* by the occurrence of transition *Start New Command Cycle*. Finally, transition *Situation Assessment* occurs and the mode changes to *Idle* – either directly or via *Resource Allocation* (if re-

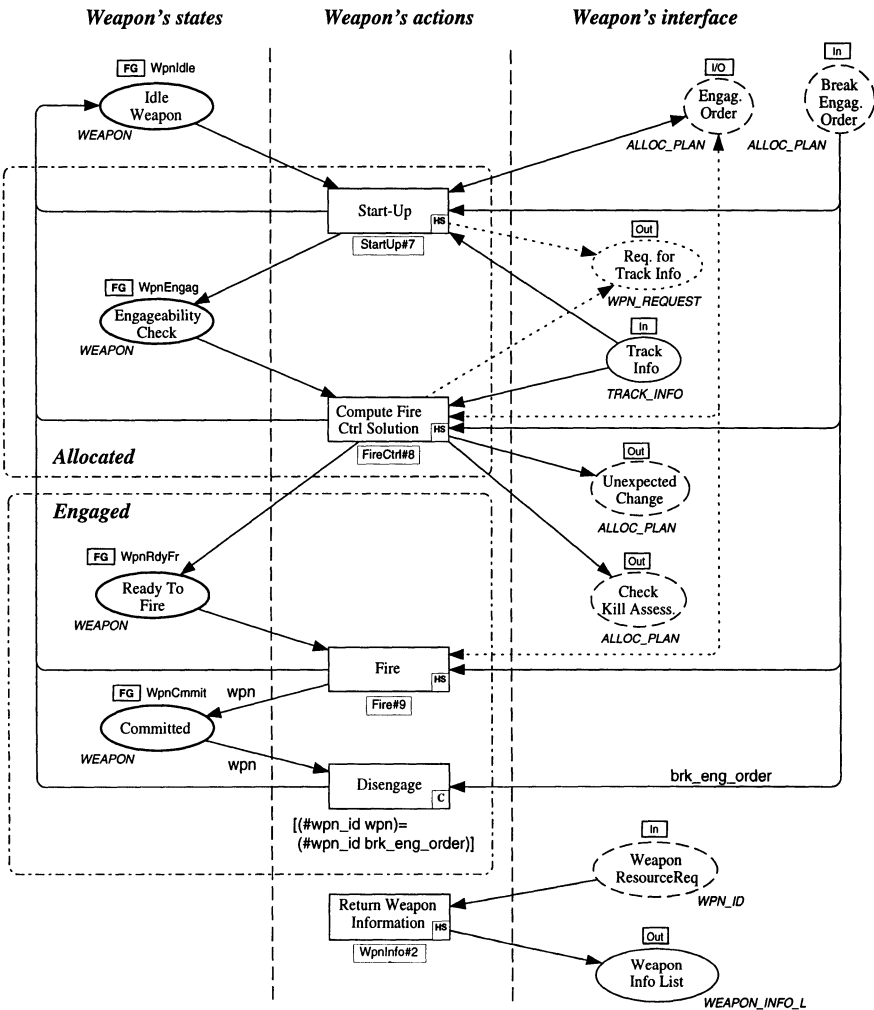


Fig. 17.6. CPN page for Weapons

planning and new resource allocation is necessary). When replanning is required, the *Mission Plan* is modified by the subpage of the substitution transition *Resource Allocation*. This may result in new engagement or disengagement orders issued to the *Action Manager* via the three output port places in the lower right part of Fig. 17.5.

Page *Weapons* in Fig. 17.6 is structured in a similar way as the previous CPN pages. In addition to *Idle* there are three other weapon modes.

The full CPN model of the naval vessel contains approximately 300 places and transitions. The modelling was done by two persons using a total of approximately two man-months over a four-month period. The two modellers had backgrounds in engineering and computer science, with some skills in system modelling and design as well as a modest experience in computer programming.

17.3 Simulation of Naval Vessel

Monte Carlo simulations have been conducted for a number of various air-threat scenarios to determine the best of two different weapon assignment policies known as *earliest intercept* and *random assignment*. For each scenario, 30 simulation runs were performed. Statistical estimates on ship survivability are shown in Fig. 17.7. For each threat level, a 95% confidence interval has been determined using a t-test. The results obtained from the CPN simulations proved to be compatible with the results obtained using a more refined simulation model generated by an object-oriented simulation environment recently developed at our research centre.

For the lowest threat intensity the two defence strategies turned out to be comparable, both yielding a reasonably high probability of ship survival. However, as the threat intensity increases, the earliest interception policy shows a better performance. The CPN model can be used to determine the critical size for the threat intensity and hence select the most suitable defence strategy. Trade-offs between the quality of resource allocation and real-time measures of performance such as execution time, responsiveness, and graceful adaptation may also be easily investigated.

Threat Intensity (size of attack)	Weapon Assignment Policy	
	Earliest Intercept	Random Assignment
4	0.967 ± 0.068	0.933 ± 0.093
5	0.833 ± 0.139	0.800 ± 0.149
6	0.310 ± 0.171	0.133 ± 0.127

Fig. 17.7. Chances for naval vessel to survive, for three different threat intensities and two different defence policies

17.4 Conclusions for Naval Vessel Project

Our Division has recently developed an object-oriented discrete-event simulation environment to investigate the performance of target evaluation and weapon assignment systems. The simulator is implemented in Sim++ and runs on Sun platforms. The object-oriented simulator has been used to model the same conceptual high-level defence system modelled by the CPN tools. Hence, it is straightforward to compare the two models and the modelling processes.

The CPN formalism offers the proper ingredients to represent the components of the problem domain, and the CPN tools facilitate system design and system validation. CP-nets capture information/data flows and system concurrency through synchronous and asynchronous processing activities and allow encapsulation and representation of behaviour at different abstraction levels – a critical feature for command and control systems. The CPN tools constitute an excellent framework for architecture analysis, minimizing the effort required to modify the model structure and network connectivity. It is also fairly simple to reconfigure activities and change communication links between key elements. Most of these characteristics make the CPN approach definitely more attractive.

The CPN approach allows the designer to focus on the model problems at hand – instead of implementation details. However, a preliminary experiment for simple system design with the CPN tools tends to suggest that even though a large variety of possible modelling mechanisms are available, the success of the CPN approach largely depends on the actual choice of aspects of the system to be modelled. Except for small systems, a single net accounting for all aspects is believed to be very unlikely. Consequently, if a complete and very detailed model is to be created, the CPN approach may be limited to simpler system components such as a specific military subsystem.

The main advantages shown by the CPN approach over the object-oriented simulation environment can be summarized as follows: rapid prototyping, formal specification, system modularity, explicit representation of concurrence, easily manageable for small models, verification capabilities, shorter development cycle, and faster maintenance. On the other hand, the weaknesses are basically related to symbolic treatment limitations, granularity of the modelling, lack of libraries supporting the implementation of sophisticated numerical algorithms, and restricted capabilities offered by the tool at hand for system design. In these areas the object-oriented approach prevails.

The CPN approach facilitates identification and analysis of the critical timing factors and impacts on overall system performance. For instance, it allows the designer to identify bottlenecks as the problem size grows, to determine reaction time (stimulus-response delay) for various critical processes, to predict overall system responsiveness and to test alternative concepts from threat detection to weapon release. These tasks can be accomplished much more easily due to the graphical representation of the model. The CPN approach also presents a capability to achieve formal analysis in order to verify system properties, e.g., via occurrence graph analysis or invariant analysis.

CPN technology appears to be a suitable approach for formal specification of concurrent system designs satisfying functional and time requirements. The underlying formalism supports rapid prototyping to test concepts and alternatives and allows the designer to focus hierarchically on problem domain modelling rather than implementation issues. Moreover, the CPN tools present a capability to perform “what-if” simulation and carry out formal analysis to verify system properties.

Chapter 18

Chemical Production System

This chapter describes a project accomplished by *Hartmann Genrich, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, Germany, Hans-Michael Hanisch, University of Magdeburg, Germany, and Konrad Wöllhaf, University of Dortmund, Germany*. The chapter is based upon the material presented in [27]. The project was conducted in 1993. The original ideas were formulated in terms of Predicate/Transition Nets, but all the modelling, simulation and occurrence graph analyses were done by means of CP-nets and the CPN tools.

We present a method for the description and validation of control procedures for multipurpose chemical batch plants (which in many respects are similar to flexible manufacturing systems). Our method is based on the use of recipes, which is a standardised concept used by the big chemical companies. The recipes and plant description are transformed into CP-nets, which are investigated by simulation and by occurrence graph analysis to find desired or critical behaviour, e.g., resource conflicts and deadlocks.

The translation of the recipes and plant description into CP-nets is very natural and straightforward. It allows us to transfer the basic concepts of CP-nets, as well as their analysis methods and tools, to a new interesting field of application. We can get a deeper understanding of the production processes described by recipes and we can prevent malfunctions before they occur in the real production system.

Section 18.1 contains an introduction to chemical production, in particular the use of recipes. Section 18.2 presents the CPN model of the recipes and the chemical plant. Section 18.3 describes the validation of the CPN model. Finally, Sect. 18.4 presents a number of findings and conclusions for the project.

18.1 Introduction to Chemical Production System

Our description of the production process is based on the concept of recipe-controlled operation. This methodology is developed by the big chemical companies, and it is used by all major suppliers of process control systems for chemical batch plants.

A **basic recipe** describes the production process in a general, plant-independent way. It has four items. The **recipe header** provides general information, such as the product name, the quantity produced, and the date for the creation of the recipe. The **list of products** describes the raw materials which

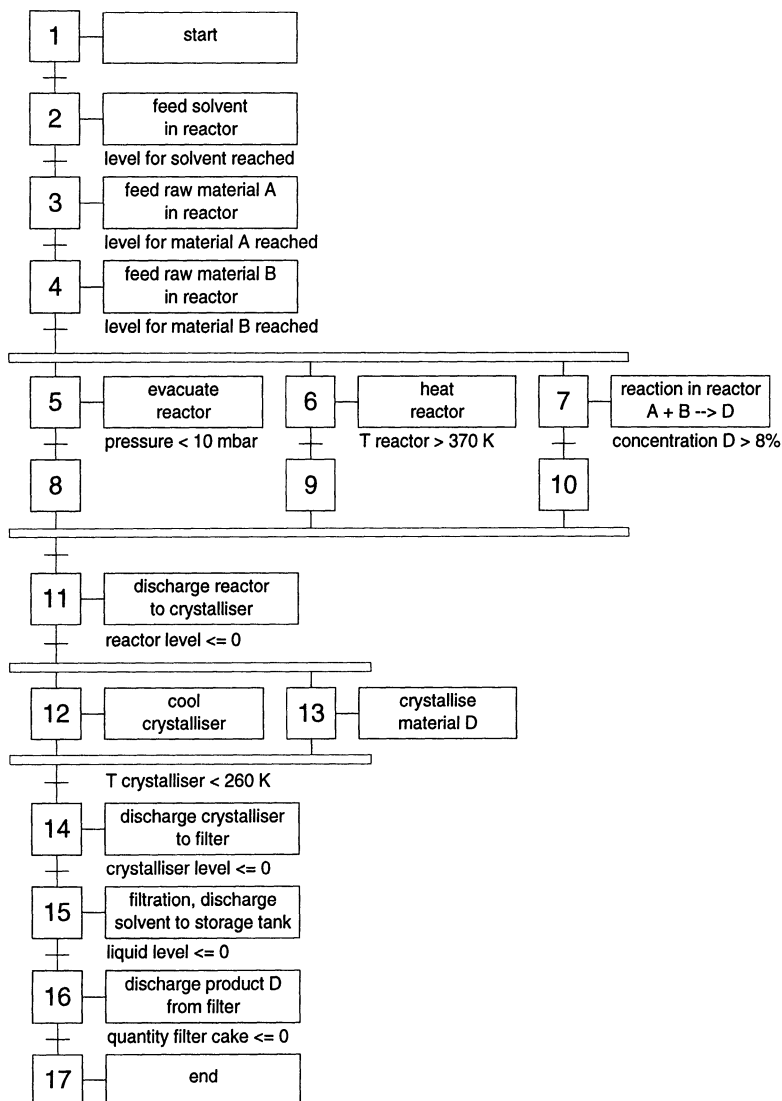


Fig. 18.1. Example of a production procedure specified in a basic recipe

are needed for the production, e.g., by specifying their chemical formula. Analogously, the **list of devices** describes the production equipment which is needed, e.g., reactors and filters. Finally, the **production procedure** describes the basic functions of the production and the sequence in which they are performed. It specifies the causal order of states and state transitions. The shifts from one operation to the next are controlled by clocks and thresholds for pressure, temperature, or concentration. Figure 18.1 shows the production procedure for a basic recipe.

First the solvent is pumped into a reactor where the raw materials A and B are added in the correct proportion (controlled by two level thresholds). Next the reactor is evacuated (to a pressure threshold) and heated (to a temperature threshold), starting the chemical reaction. When a specified concentration threshold is reached, the contents of the reactor are discharged into a crystalliser which is cooled down (to a temperature threshold). Then the crystallisation process is assumed to be finished and the whole mixture is discharged into a filter, where the product is separated from the solvent, which is moved to a storage tank to be reused in a subsequent production cycle.

From the basic recipe a **control recipe** is developed. It is similar to the basic recipe except that a number of parameters now are replaced by real values, determined from the production plan and the specific properties of the chemical plant. As examples, we now know the desired production quantity and the exact production equipment to be used. The basic recipe specifies that a reactor with certain properties must be used. The control recipe tells us exactly which reactor it is.

The chemical plant is characterised by the individual pieces of production equipment, their connections, and the **technical functions**, i.e., the actions which can be performed on them. Figure 18.2 shows a small part of a chemical plant containing a crystalliser, two filters, a cooling device, and five valves. On this part it is possible to perform the technical function *Discharge from C1 to*

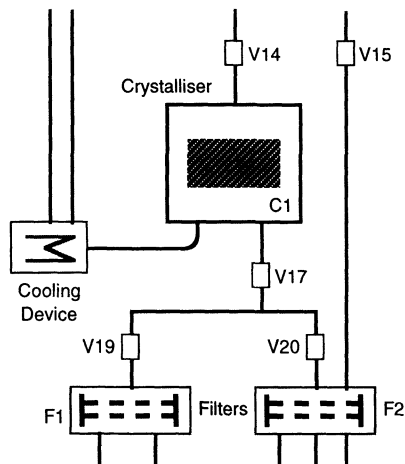


Fig. 18.2. Small part of a chemical plant

F2. The crystalliser C1 is the source of the material, while the filter F2 is the target. The valves V17 and V20 are resources that must be used exclusively by this technical function. Moreover, it must be guaranteed that no other technical function will open the valve V19 while this function is active, so this valve must be blocked. Another technical function for this production unit may be *Cooling of C1 by Cooling Device*. The multipurpose plant is the “hardware” to execute the production processes defined in the basic recipes. It offers technical functions providing the realisation of the basic functions in the basic recipes.

For specifying the recipes, as well as the chemical plant, special graphical and textual editors are available. We derived the CPN model from specifications made in these editors. The translation was manual. However, it is possible to formalise the translation and automate it. To use our approach in practice this must be done.

The concepts of basic recipes and control recipes are now widely used for the design of chemical processes. However, in their standard form, there are a number of problems:

- Recipes are designed by intuition and they are not systematically analysed, e.g., to check whether the individual items of a control recipe are consistent with those in the corresponding basic recipe.
- Each control recipe describes the sequence of technical functions needed to manufacture a single product, and there is no attempt to describe or analyse the interactions between different recipes concurrently performed in a plant and sharing resources such as reactors, cooling devices, filters, and valves.
- Recipes usually describe the desired sequences of steps, while the control operations necessary to handle disturbances are specified separately, although they can fundamentally change the interactions between recipes, leading to critical and dangerous situations.

Below, we focus on the second of these problems, i.e., the normal-mode interactions between recipes. However, we are also able to model disturbances and the modification of resource allocation caused by these.

18.2 CPN Model of Chemical Production System

The basic idea behind our work is to create a single CPN model which encounters the chemical plant as well as the control procedures. The model describes:

- The allowed sequences of basic functions in the basic recipes,
- The process devices in the chemical plant and the technical functions of these,
- The allocation of process devices to the control recipes.

Figure 18.3 shows the CPN page for the production procedure in Fig. 18.1. The places p1–p17 represent the execution of the different basic functions, while the remaining places represent the raw material and production equipment needed. The transitions represent the events that start and supervise the basic functions.

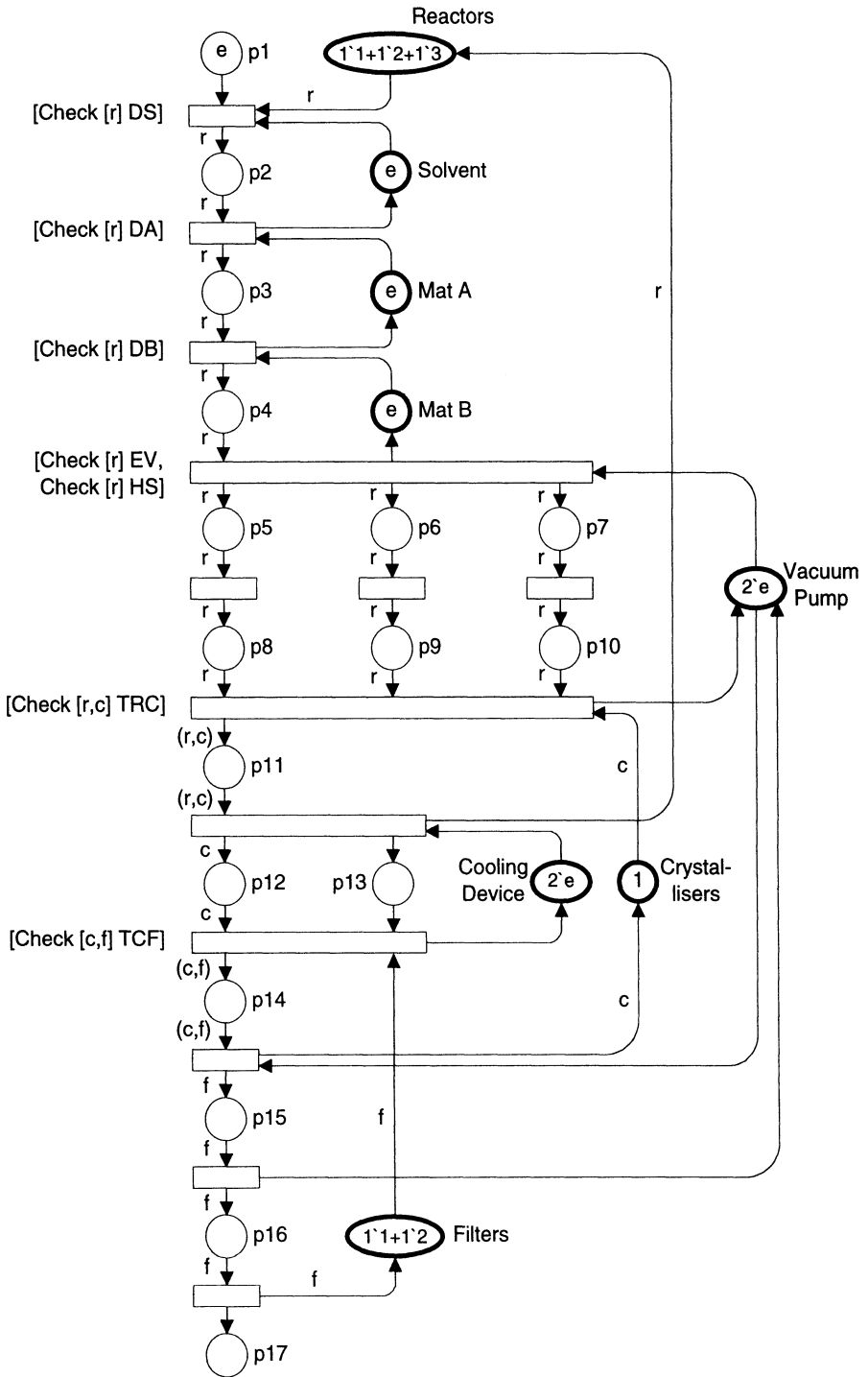


Fig. 18.3. CPN page for the production procedure in Fig. 18.1

The variables r , c , and f are used to denote reactors, crystallisers, and filters, respectively. We have hidden all colour sets, since they can be deduced easily from the arc inscriptions. Arcs with hidden arc inscriptions have e as arc expression.

Each process device is represented by a token. For a reactor the token could look as follows, where *TechFns* specifies the technical functions in which the reactor is able to participate:

```
{ Name = R1,
  Volume = 6.3,          (* m3 *)
  MaxTemp = 473.0,      (* Kelvin *)
  MaxPres = 10.0,       (* bar *)
  Material = Mat0,
  TechFns = [DS, DA, DB, DC, EV, HS, TRC1, TRF2] }.
```

```
DS    Dosing from solvent to reactor
DA    Dosing from raw material A to reactor
DB    Dosing from raw material B to reactor
DC    Dosing from raw material C to reactor
EV    Evacuation from reactor by vacuum pump
HS    Heating by steam device
TRC1  Transfer from reactor to crystalliser C1
TRF2  Transfer from reactor to filter F2.
```

However, for our work, it turned out to be more convenient to take a simpler approach, in which each reactor is represented by a token with colour i , where i is a natural number describing the identity of the reactor. The position of the token indicates the current use of the reactor. When the i -token is positioned on place *Reactors*, the reactor R_i is free. When the i -token is positioned on one of the places p_1 – p_{17} , the reactor R_i is in use for the corresponding basic function.

The other kinds of process devices are represented in a similar way. This means that tokens for the places representing basic functions have a colour of the form (i, j, \dots, k) , where i, j, \dots, k are natural numbers identifying the process devices which are in use for the execution of the corresponding basic function. As an example, p_{11} has tokens where the token colour is a pair in which the first element identifies a reactor while the second identifies a crystalliser.

From the initial marking of Fig. 18.3, it can be seen that the plant has three *Reactors*, two *Filters*, and one *Crystalliser*. The guards of the transitions check that the chosen process devices are able to perform the required technical functions. This is done by an ML function *Check*, in which the information about technical functions is hard-coded. For example, the topmost transition has a guard which checks that the reactor r is able to perform the technical function DS needed for the execution of the basic function represented by p_2 .

Each storage tank is represented by a place, which contains a token when the tank is available and no token when the tank is in use. For example, Fig. 18.3 contains three places representing tanks for the *Solvent*, raw *Material A*, and raw *Material B*. For this kind of equipment no identification information is needed and hence the tokens are e-tokens. Alternatively, we could have used a single

place to represent all storage tanks, using the token colour to denote the identity of the tank.

Some kinds of resources may be used by several processes at a time, but still have some limits. Examples are the *Vacuum Pump* and the *Cooling Device*. The initial marking of these places reflects that they both have the capacity to serve two production processes at a time. Again, we could have used a single place, with an initial marking of $2 \wedge \text{Vacuum Pump} + 2 \wedge \text{Cooling Device}$.

Usually, the resource allocation is not explicitly specified – neither in the basic recipes nor in the plant description. The decisions are left to the plant operator or to the programmer of the process control system. Inadequate resource allocation may cause hazardous states or blockage in the production system. Hence, we want to apply the analysis techniques of CP-nets to detect such errors before the production process is performed.

18.3 Validation of Chemical Production System

The CPN pages for the individual recipes can be simulated either separately or as a joint CPN model. In the latter case, they interact via the common resource places for reactors, storage tanks, cooling devices, etc. All these places are global fusion places. To improve readability we have drawn these places with thicker lines and hidden the fusion tags, since they provide no additional information. The initial marking of the topmost place (in each recipe model) reflects the production target, i.e., the number of batches that should be manufactured. The conflicts in the CPN model represent resource allocation decisions to be made. Suitable decisions will ensure that no deadlocks occur.

Each CPN page represents all possible control recipes for a basic recipe in the chemical plant considered. A specific control recipe is obtained by binding each of the variables (r , c , f in Fig. 18.3) to a specific piece of the production equipment. In this way, we may, e.g., get a control recipe that use reactor R1, filter F2, and crystalliser C1.

The CPN models can be debugged by interactive simulation. We can study the execution of specific control recipes and investigate whether the sequences of technical functions prescribed by the recipes meet the requirements of the manufacturing processes. We can also study the flow of resources, as well as the flow of batches through the production system.

For a systematic detection of possible malfunctions, such as deadlocks, we use occurrence graph analysis. If a deadlock is found, it is possible to investigate why it appears. To portray the occurrence sequences leading to the deadlock, we have extended the occurrence graph tool with an ML library that allows the user to obtain graphs like the one shown in Fig. 18.4. The graph is automatically derived from the occurrence graph, but the layout is improved, manually, by using the alignment commands of the CPN tools. The graph displays the history of a deadlock caused by choosing reactor R3 for the manufacturing of product I and reactor R1 for manufacturing of product II. Product I is modelled by the CPN page in Fig. 18.3, while product II is a similar, but slightly different product.

The deadlock results from the fact that reactor R1 cannot be cooled (because no cooling is provided) and reactor R3 cannot be evacuated. The graph displays the occurring transitions – input places display the marking before the occurrence of the corresponding transition, while output places display the marking after the occurrence (output places with empty markings are omitted). The graphs constitute a convenient and fast way to investigate abnormal behaviour. They are much easier to overview than the full occurrence graph, which in this case has more than 800 nodes and nearly 2000 arcs.

To prevent problems like the one described above, each process device must be checked, at the very beginning, for all the technical functions needed during the processing of the entire recipe. But even if this is done, there may easily exist inadequate resource allocation strategies causing delays or blockage of one recipe by another. As an example, we may choose filter F2 for the manufacturing of product I. This is possible, because F2 has all the required technical functions. However, this only leaves filter F1 for product II, and this filter cannot be used (since the only way material can be transferred from a reactor to F1 is via the crystalliser C1 which is occupied by product I).

The kinds of malfunction described above may seem trivial and easy to detect, but this is not at all the case. Multipurpose chemical plants are often very complex, producing more than 100 different products with up to 10 control recipes running concurrently. In such cases, a subtle interplay of several recipes may lead to occasional malfunctions that are hard to detect and correct – unless one uses a systematic approach such as our occurrence graph analysis.

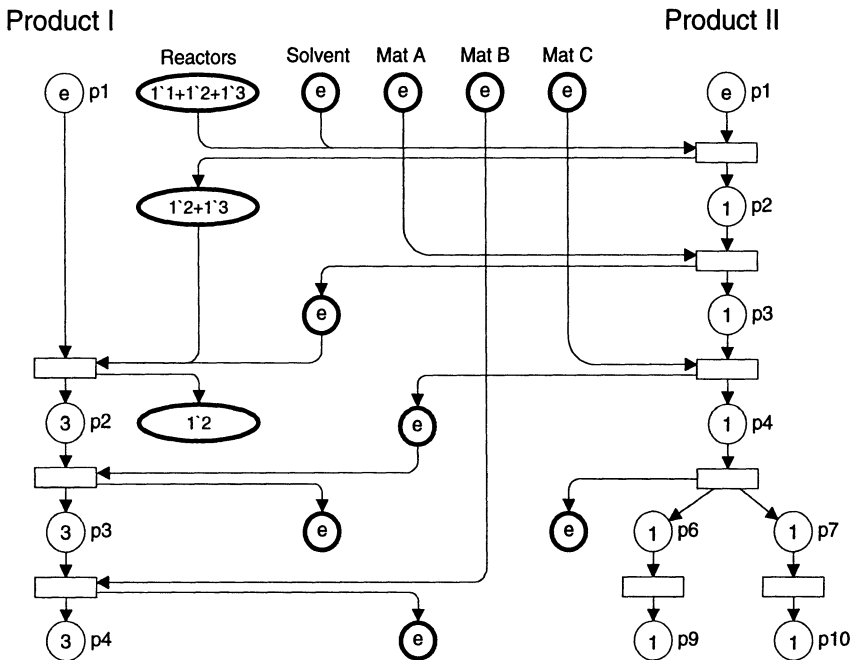


Fig. 18.4. The history of a possible deadlock in the chemical production system

18.4 Conclusions for Chemical Production Project

We have developed an approach for analysing recipe-controlled chemical production processes by means of CP-nets. The approach can be extended to handle larger and more complicated problems, as shown in [9].

It is easy to extend the guard expressions to check for the temperature, pressure, and material constraints (mentioned in the description of the reactors). It is also straightforward to model the use and blockage of valves, in a similar way as we model the use of other kinds of production equipment. Another possible extension is to include time issues, obtaining a timed CPN model which can be used to solve scheduling and performance problems.

Finally, it is possible and necessary to develop bridges between the different tools. This will allow the CPN model to be derived automatically (or at least semi-automatically) – removing the rather trivial, but time-consuming and error-prone task of creating the CPN model.

It is our hope that the basic ideas described in this chapter may lead to a computer-aided verification tool for recipes ensuring more effective and safer control procedures in the chemical industry. Work in this direction is in progress.

Chapter 19

Nuclear Waste Management Programme

This chapter describes a project accomplished by *Kjeld H. Mortensen and Valerio O. Pinci, Meta Software Corporation, Cambridge MA, USA*. The chapter is based upon the material presented in [41]. The project was conducted in 1991.

We describe how CP-nets and the CPN tools were used to model and improve a nuclear waste management programme in charge of the creation of a new system for permanent disposal of nuclear waste. We used Structured Analysis and Design Technique (SADT) together with CP-nets. This was done in a similar way as described in Chap. 14. We used the SADT tool to obtain a work flow description of the activities to be performed by the waste management programme. The SADT description was then translated into a number of CP-nets which were augmented with additional behavioural information. Each of the CP-nets was simulated to produce event charts displaying the activities of the corresponding part of the waste management programme. The event charts were used to investigate whether there were any blockages in the information flow of the individual parts. Finally, all CPN models were merged into a single simulation model that was used to validate the interaction and co-operation between the different parts of the waste management programme.

Section 19.1 contains an introduction to the nuclear waste management programme and describes the organisation of the project. Section 19.2 describes how the SADT model was translated into a number of CPN models. It also describes how decision tables were used to convey the intended behaviour of the individual activities. Section 19.3 reports upon the simulation of the CPN models. We also briefly discuss how the CPN models were merged into a single simulation model. Finally, Sect. 19.4 presents a number of findings and conclusions for the project.

19.1 Introduction to Nuclear Waste Management Programme

A large nuclear waste programme in USA is responsible for permanently disposing of used nuclear fuel and similar high-level nuclear waste. By programme we here mean an organised set of activities directed towards a common purpose.

The objective of the nuclear waste programme is to establish a capability to accept, transport, and store nuclear waste by 1998, and to start the storage of nuclear waste in a geological repository by 2010. The programme has quite unique characteristics. It provides safe nuclear waste isolation for 10 000 years with unprecedented oversight and control by different affected and interested groups. Additionally, the programme must take into account changing conditions in its environment, e.g., changes in the current legislation. The programme director therefore decided to develop the management strategy and to carefully design the programme, much like physical systems are designed. A general design methodology is used to capture the functionality of the organisation.

The physical waste management system is composed of groups of people, documents, and equipment. They need to co-operate and interact with each other. A group of people in charge of a specific domain needs to exchange many kinds of information with other groups. In order to ensure an efficient and consistent co-operation and interaction between groups, the Nuclear Waste Management System (NWMS) is modelled with SADT, and the resulting model is analysed by translating it into CP-nets which subsequently are simulated. For more information about SADT, see Sect. 14.1 and [38].

Certain major components of the system are called the programmatic functions. These are the activities that bring the physical nuclear waste management system into being. The process of modelling and analysing the programmatic functions will be referred to as the programmatic functional analysis (PFA). The modellers take the perspective of functional behaviour on the nuclear waste management system. Models will provide a means for people in the programme to better understand their position in the overall programme. They will be able to identify their own activities, e.g., how co-operation and interaction should take place with other people in the management system. The result of the analysis is to be used for developing policies and guidelines to improve the management system.

The modelling work was done during a six-month period. Two groups of people with very different backgrounds and qualifications participated in the project. The main group performing programmatic functional analysis (which we will refer to as the PFA team) was responsible for designing the new system management strategy. The PFA team produced, among other things, functional descriptions in the form of SADT diagrams. The size of the team varied between 15 and 25 persons. The waste management programme interacts with and is restricted by a variety of groups in the world outside. As an example, the programme has to work within changing laws of the government, and it has to handle and interpret many sorts of data coming from, e.g., geologists who analyse potential storage sites. Hence, the PFA team contained people with very different

knowledge and experience, e.g., lawyers and geologists. There was no attempt to train the PFA team in using CPN.

In this chapter the word “we” refer to the members of the CPN team. The CPN team consisted of two persons having several years of experience with SADT and CPN. It was the responsibility of the CPN team to convert the SADT diagrams into executable CPN models, based on additional written descriptions of the intended behaviour of the individual activities. In this way, we created a number of CPN models which were used to simulate and analyse different aspects of the waste management system. An example of an interesting question is whether there is any unintended blockage of the information flow in the system. Such a blockage may be discovered as a deadlock in the model. In the real world there will not be a deadlock, but some activities will be suboptimal and hence imply delays. The purpose of our involvement was to provide an executable model of the PFA team’s SADT model, to provide a basis for validation and improvement of the accuracy and completeness of the programmatic functional analysis.

From the PFA team’s point of view, the interaction and co-operation between the various submodels can be compared with a protocol. As the SADT submodels are made independently by different people, it is not guaranteed that the submodels will be consistent, i.e., fit nicely together when the model is viewed as a whole. The SADT modellers used the CPN models and the CPN simulations to validate that the SADT submodels had consistent interfaces and were able to interact and co-operate as expected. Via the event charts produced by the simulations the SADT modellers also obtained a much more concrete and detailed understanding of the system behaviour than could be accomplished by just looking at the static SADT diagrams.

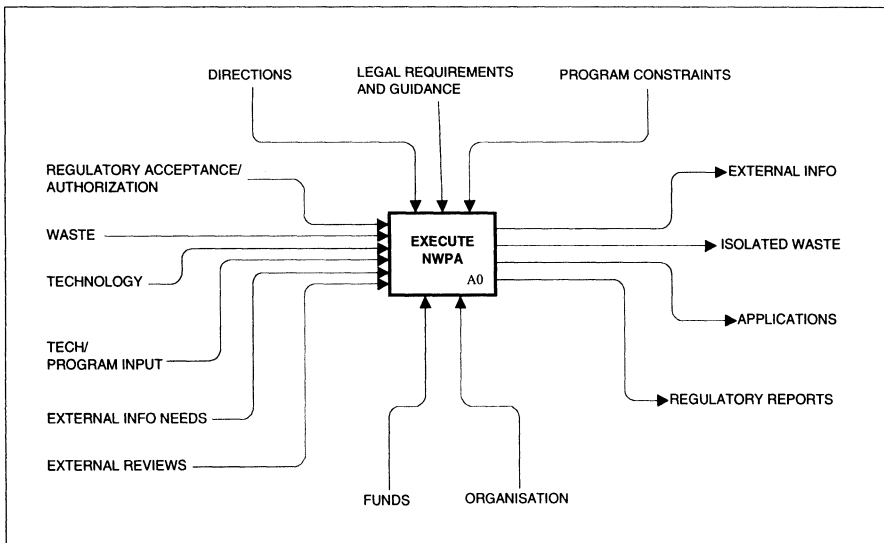


Fig. 19.1. Most abstract SADT page for nuclear waste management programme

Figure 19.1 shows the most abstract SADT page. It provides an overview of the interaction between the waste management system and the external environment. The activity has a number of inputs, e.g., *Waste* and *Technology*. It has a number of outputs, e.g., *Isolated Waste*. It operates under the control of, e.g., *Legal Requirements* and *Programme Constraints*. Finally, it uses mechanisms, e.g., an *Organisation* (people and machinery). At the lower-level pages all mechanism arrows are omitted. This implies that the model does not deal with the use of resources.

To finish the CPN model within the available resources, the CPN model only deals with the most important parts of the management programme, known as the “seven major programmatic functions”:

- Provide Programme Control (PPC) controls and provides overall management direction for the NWMS programme.
- Ensure Regulatory Compliance (ERC) identifies regulations that apply to the programme and the physical system.
- Perform Systems Engineering (PSE) transforms NWMS mission requirements into functions, requirements, and interfaces for physical system.
- Design Engineered System (DES) is divided into four phases: conceptual, preliminary, final, and as-built design.
- Identify and Characterise Sites (ICS) finds and screens potential sites for nuclear waste storage.
- Evaluate Integrated System (EIS) is intended to reduce programme technical performance risks.
- Perform Confirmation/Construction/Operational Testing (PCOT) plans, conducts, and documents tests to verify that the physical system conforms to, e.g., technical requirements.

The latter five functions are the activities of the SADT diagram shown in Fig. 19.2. The remaining two functions are located in other parts of the model. The five activities in Fig. 19.2 have a close interaction. Typically, PSE provides input information to one of the other four activities, which in turn provides a result which is either a success (i.e., a final result) or a request for more information or additional action. The result is processed by PSE, sometimes in co-operation with PPC. And so it goes on through all stages of the NWMS programme.

Typically, the PFA team finished a first version of one of the submodels. Our task was then to translate the SADT diagram into a CPN model and to finish the model by adding behavioural information so that a simulation could take place. An example of a typical SADT page is shown in Fig. 19.3. The first activity is to *Identify Variances* (i.e., possible delays). Then we *Determine Causes of Variances* and *Determine Variance Impacts*. Finally, we *Identify Corrective Actions* to be taken. The results are used to make change requests. Approved changes are sent to ERC, which is responsible for realising the requests.

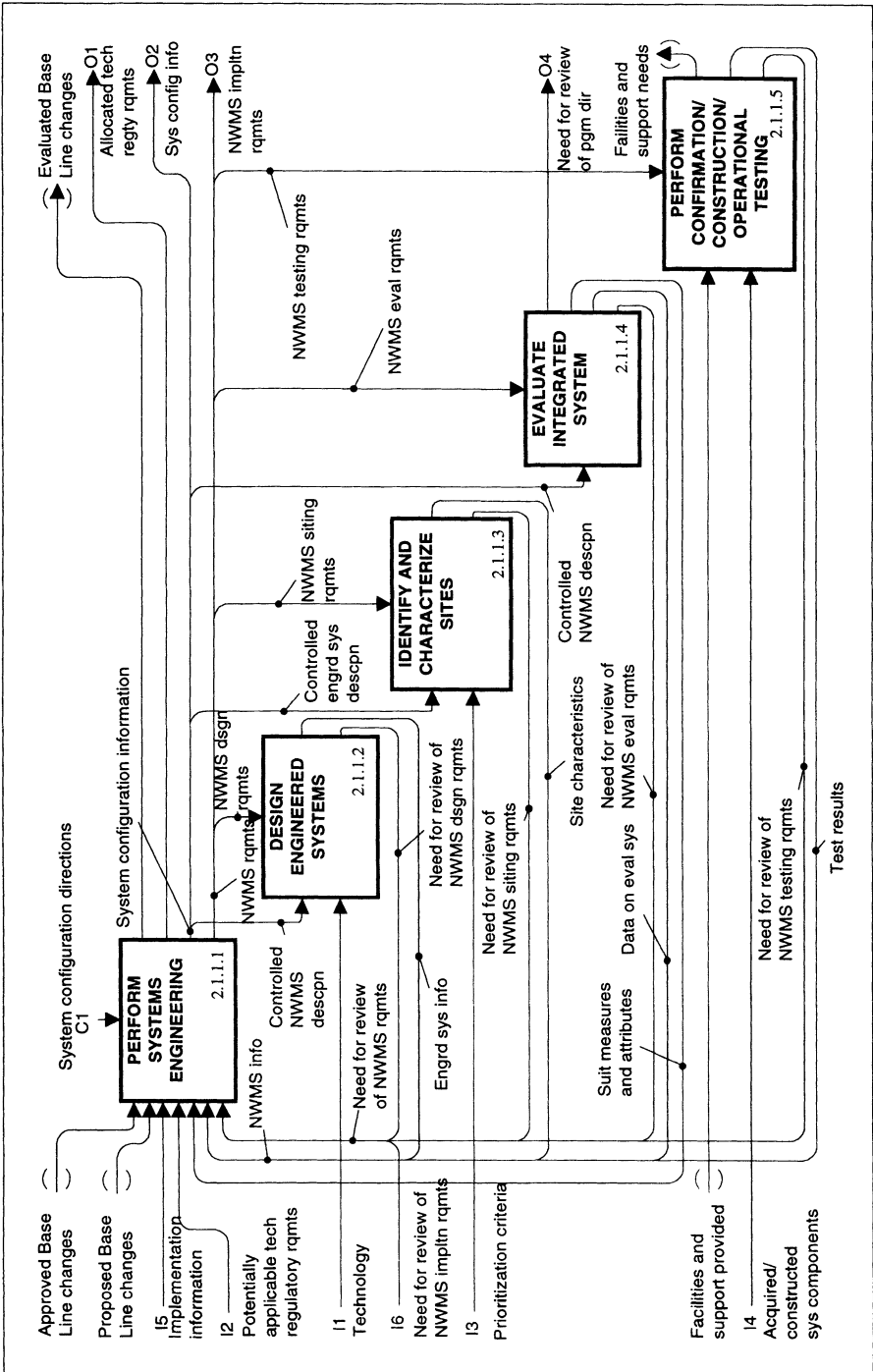


Fig. 19.2. SADT page showing the relation between five programmatic functions

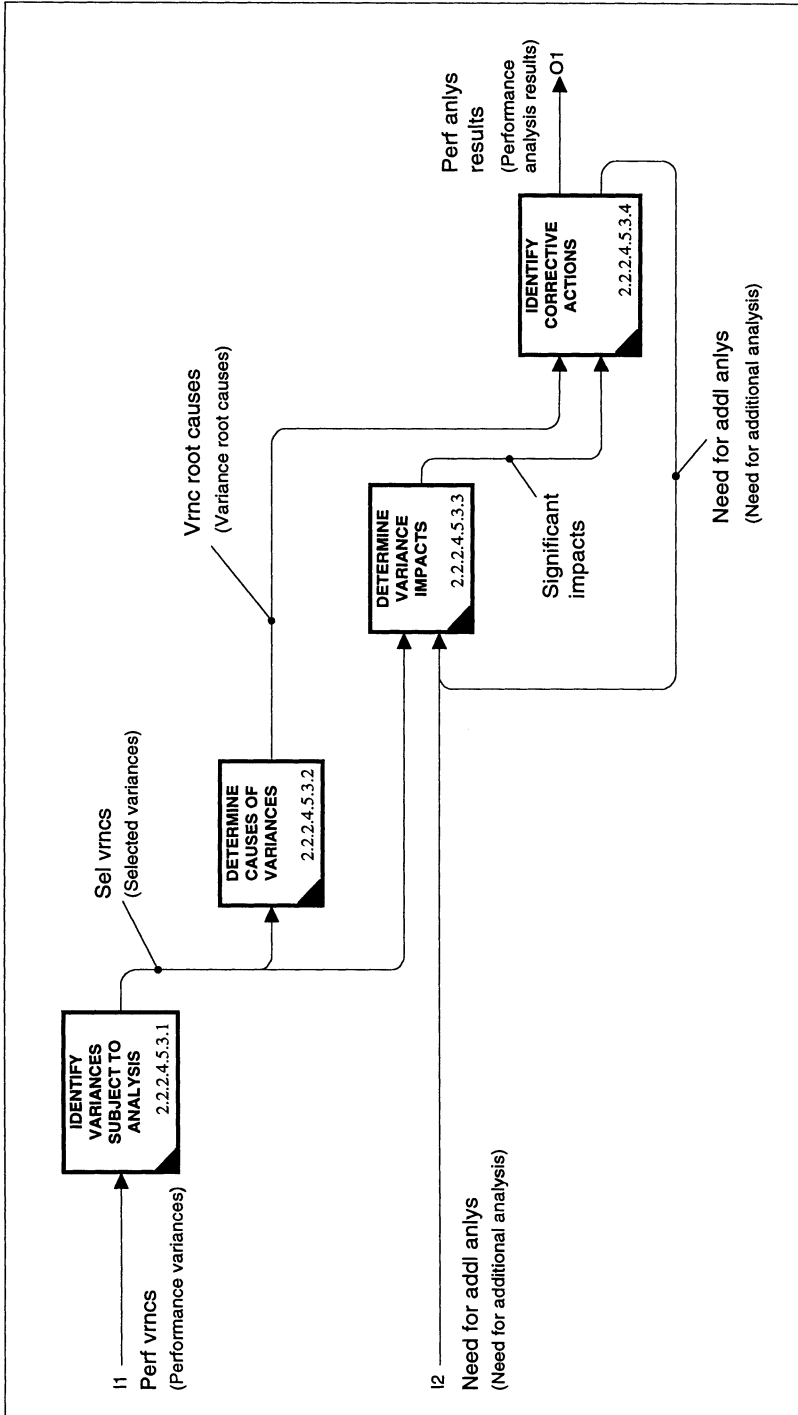


Fig. 19.3. Typical SADT page from nuclear waste management programme

19.2 CPN Model of Nuclear Waste Management Programme

Together with the SADT models the CPN team got a clarifying description of the intended behaviour of the individual activities. Initially these descriptions were made in unstructured prose, as illustrated by the following example. The mission part explains what the activity is supposed to do in terms of input and output. The scope part explains to what extent the rest of the waste management programme is involved in the actions of the activity.

Identify Corrective Actions

Mission: Develop alternative actions to correct root causes and mitigate impacts of the variances affecting the programme.

Scope: Analyse the variance root causes and impacts and develop alternative corrective actions to correct the cause or mitigate the impact. Analysis includes identifying and evaluating the risks associated with each corrective action.

Communicating the intended behaviour in the above form turned out to be insufficient and ambiguous. Hence a more structured kind of descriptions was introduced as shown below. The behaviour is now described by means of a decision table. There is a row for each arrow and there is a column for each possible action pattern. Patterns to be used the first time the activity is executed is marked with a “1”, while patterns to be used at subsequent executions are marked with an “S”. The crosses indicate the inputs to be received and the outputs to be produced. From the decision table, we learn that *Identify Corrective Actions* needs input along both of its incoming arrows, while it produces output along exactly one of the outgoing arrows. As can be seen, the decision table was accompanied by a number of notes, which basically tell the same story in prose.

Identify Corrective Actions	Operative Cycles				
	Type	1	1	S	S
Arrow Label					
Significant impacts	In	x	x	x	x
Variance root causes	In	x	x	x	x
Performance analysis results	Out	x		x	
Need for additional analysis	Out		x		x
<p>1) Initial execution requires both the Significant Impacts and Variance Root Causes and will produce either the Performance Analysis Results (which includes alternative corrective actions if applicable) or the Need for Additional Analysis.</p> <p>2) Subsequent executions of the activity behave in the same manner as the initial execution.</p>					

The use of decision tables gave a significant speedup in our process of understanding the SADT model. We were now also, at an early stage, able to predict unintended behaviour. As an example, the above decision table may cause a blockage in the information flow. When the result of the activity is *Need for Additional Analysis*, the third activity in Fig. 19.3 is executed once more. However, this only creates one of the inputs needed by *Identify Corrective Actions*, while the decision table tells us that we always need both inputs. This problem was reported to the PFA team, which modified the decision table so that subsequent executions of the activity only require one of the inputs. The modified decision table looks as shown below. By identifying and fixing this type of error at an early stage, we saved a lot of time.

Identify Corrective Actions	Operative Cycles						
	Type	1	1	S	S	S	S
Significant impacts	In	x	x	x		x	
Variance root causes	In	x	x		x		x
Performance analysis results	Out	x		x	x		
Need for additional analysis	Out		x			x	x

We transformed the SADT model into a number of CPN models – one for each of the seven major programmatic functions considered. This was done as described in Chap. 14. The net structure, the hierarchy structure and the colour set names were obtained, automatically, from the SADT diagram. Then the net inscriptions and the detailed colour set declarations were added manually.

During the automatic translation from SADT diagrams into CP-nets, the graphical layout is preserved. The CPN transitions and arcs get the same name, shape, size, position, and graphical attributes as the corresponding SADT activities and arcs. Figure 19.4 shows the CPN page obtained from the SADT page in Fig. 19.3. A blow-up of the rightmost transition is shown in Fig. 19.5. Here, we also show some of the net inscriptions that are hidden in Fig. 19.4.

The dashed places and arcs and all net inscriptions (except colour sets) are added manually (as explained below). There are a lot of details in Fig. 19.4, but we will only explain the most important ones.

As can be seen from Figs. 19.3 and 19.4, the colour set names (of the non-dashed places) are inherited from the arrow labels of the SADT page. All the colour sets are structurally equivalent and defined as follows:

color C = record Ver: int * Info: string * Av: bool;

The *Version* field is used to identify whether new tokens have arrived. The *Information* field contains the name of the SADT arrow along which the token just travelled. It tells us where the token came from. Finally, there is an *Availability* field. This field is only rarely used, and hence we will not explain it.

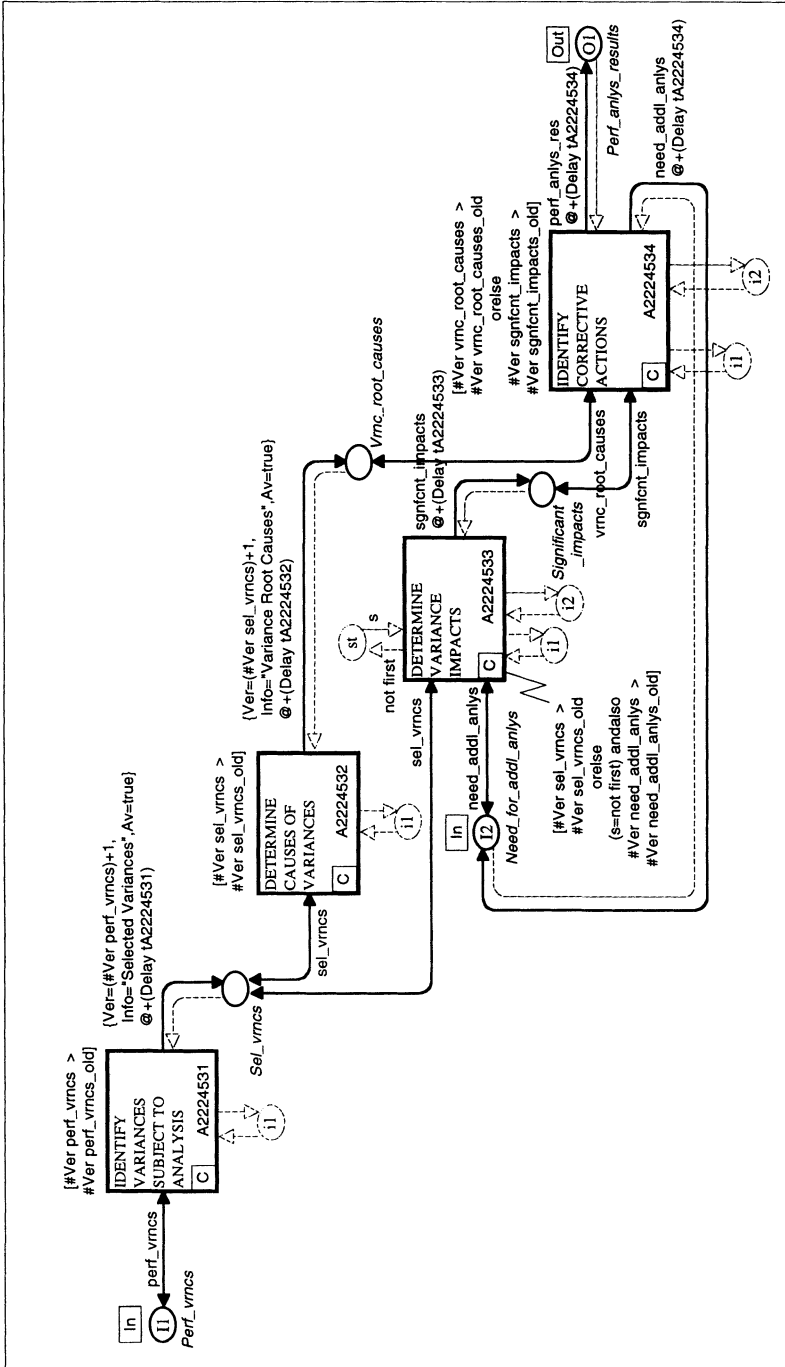


Fig. 19.4. CPN page for the SADT page in Fig. 19.3.

It would of course have been possible to use a single common colour set name instead of defining a number of identical colour sets. However, this would have made the CPN models less comprehensible for the PFA team, because the SADT arrow labels convey important informal information about the purpose of the arrow.

The CPN model is constructed in such a way that each place (dashed or not) always contains exactly one token. Hence there are always two arcs (or a double arc) between a transition and its surrounding places. This way of modelling may seem a bit strange for CPN people, but it is quite natural for SADT people who usually consider the inputs of an activity to be persistent material that can be used by several activities without destroying it. Actually, a similar modelling practice is used in many CPN models of hardware, where a physical line always has a value, high or low. Examples of this can be found in Chap. 11.

The dashed places below each transition contain copies of the last tokens removed from the input places. (there is a dashed place for each input place). This is used to determine whether the input places have received new tokens, i.e., tokens the transition has not yet dealt with.

The dashed place above the third transition tells us whether it is the first time the transition occurs or not. Initially all places contain one token with colour:

{Ver = 0, Info = "", Av = true}.

The only exception is the dashed places above transitions. They contain a token with value: first.

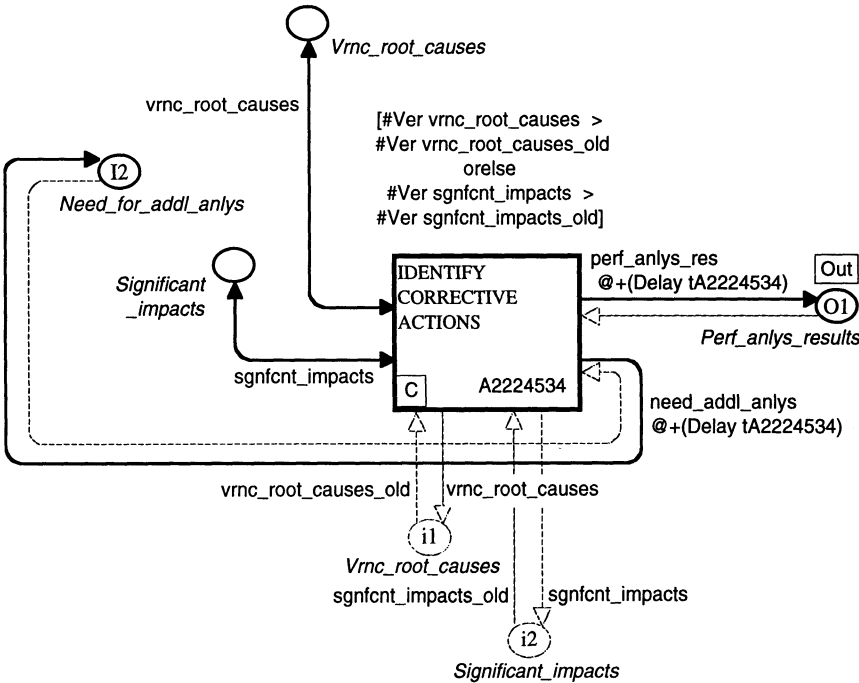


Fig. 19.5. Detailed look at one of the transitions in Fig. 19.4

Each guard checks whether there are enough input places with new tokens to match the crosses in a column of the decision table. When this is the case the transition can occur. Notice that the third guard contains logic to distinguish the first occurrence (specified by 1-columns) from subsequent occurrences (specified by S-columns). At first glance, one would expect the transition in Fig. 19.5 to make a similar test (because the modified decision table describes different enabling rules for the first/subsequent occurrences). However, for the first occurrence the two inputs will be available simultaneously, and hence the description in Fig. 19.5 is correct. A more detailed explanation of this can be found in [41].

As indicated by the small C in the lower left corner, each transition has a code segment, i.e., some sequential code which is executed each time the transition occurs. The code segments are, among other things, responsible for creating the colour values of the output tokens. The actual values are determined from the input tokens and from a set of configuration data (accessed via reference variables). The configuration data determine the maximum number of times an activity may fail (i.e., create output with demand for iteration), the probability of failure, and how much this probability is decreased each time the activity fails. The code segments are also responsible for updating the graphics of the event charts (to be explained below). The contents of the code segments are not shown in Fig. 19.4. It is just a bunch of trivial code for manipulating reference values and graphical objects.

Each transition has a time delay describing the duration of the activity. The time delays are also part of the configuration data. They are accessed via reference variables by the ML function *Delay*. When the CPN model is initialised the reference variables get values that are read from a text file. In this way it becomes easy to experiment with different configurations.

In our project, we manually added the dashed places and arcs and all arc expressions and guards. This was a rather trivial but time-consuming task. The following table shows the size of three of the submodels together with the number of person-days used to add net structure/inscriptions and the number of person-days used for testing:

Model	Pages	Transitions	Additions	Testing
PPC	12	34	6 days	5 days
ERC	17	45	7 days	6 days
PCOT	11	18	3 days	3 days

From the examples earlier in the section, it should be obvious that most of the addition process can be automated. This will not only remove the time used for additions, it will also dramatically reduce the time used for testing. This idea is used in a new tool developed several years after our project. The tool allows the user to specify work flow models by means of SADT like diagrams accompanied by configuration data. The work flow models are then, totally automatically, translated into CPN models. For more information about this approach, see [40].

19.3 Simulation of Nuclear Waste Management Programme

Whenever one of the seven CPN submodels was finished, it was simulated, using a simple environment representing the interaction with the other submodels. The result of the simulation was a set of graphical reports, called event charts. A typical event chart is shown in Fig. 19.6. Each horizontal line represents an activity. The leftmost field of the line contains the SADT identification number for the activity (displayed in the lower part of each transition). The rest of the line tells us when the transition has occurred. Each asterisk indicates an occurrence at the time shown in the upper line. If an activity occurs repeatedly at the same time, the asterisk is replaced by a digit (indicating the number of occurrences).

The event charts were sent back to the PFA team for review. The charts gave the team members a more detailed understanding of the dynamics behind the SADT models they created. The event charts were used to validate the accuracy and completeness of the SADT submodels, e.g., to investigate whether the model reflected the intended behaviour and how well it interacted and co-operated with other submodels. Often the inspections resulted in requests for changes in the CPN model or redesign of pages in the SADT model. During the construction and simulation of the individual submodels a lot of information flow blockages were discovered (like the one described in Sect. 19.2).

When all seven CPN models were finished they were put together and simulated to investigate the behaviour of the entire model. This simulation revealed, e.g., a number of information flow blockages caused by inadequate interaction and co-operation between the different submodels, i.e., the seven major programmatic functions.

Due to hardware and software limitations at the time our project was conducted, we used a number of special techniques to combine the seven rather large

Activity		110	120	130	140	150
A222451	>.....
A222452	>.....*
A2224531	>.....*
A2224532	>.....*
A2224533	>.....*	*..*
A2224534	>.....*	*..*
A222454	>.....	*..*

Fig. 19.6. Extract of an event chart from nuclear waste management programme

CPN models into a single simulation model. To obtain a simulation model that could be executed on the available computer, we turned each CPN model into a stand-alone ML program (without any graphics). Then the seven ML programs were combined into a single program by adding a few lines of ML code connecting the seven subprograms. Today this approach would no longer be necessary, since it would be possible to combine the seven submodels directly into a single CPN model without exceeding the capacity of the available hardware and software. Here we will not describe in detail how we put the ML models together. Such a description can be found in [41].

Having combined the CPN submodels into a single simulation model, we were able to perform simulations of a model with more than 100 pages, 2 000 places, and 300 transitions (not counting substitution transitions).

As explained above, our simulations produce a set of event charts, each of which represents the activities of a single submodel. One of the submodels uses three different event charts and hence the combined simulation model produces nine different event charts, each of which is split into a number of pages (windows) representing different sections of the simulated time interval. Hence, a typical simulation may produce more than 60 pages (i.e., 60 windows) with event charts. To ease navigation through all this information, we created the overview chart shown in Fig. 19.7. This chart has a line for each of the nine event charts. Otherwise the overview chart has the same format as the event charts. By double-clicking one of the rows in the overview chart the user jumps to the corresponding event chart. It is also possible to follow the link backwards, in order to return quickly to the overview chart.

The overview chart in Fig. 19.7 shows the order in which the individual submodels become active. By looking at the chart is often easy to spot malfunctions. As an example it can be seen that the ICS submodel has an activity at time 339. This was not intended and hence it needs further investigation.

Activity Time	310	320	330	340	350	360	370
PPC	>*****	*****	*****2	*.....	*****
ERC	>.....*******
PSE	>*2*.....2**2	2**.....***22***2*2*
DES_CD	>.....****2	*****2	*****
DES_PD	>.....
DES_FD_AB	>.....
ICS	>.....*
EIS	>.....
PCOT	>.....

Fig. 19.7. Overview chart from nuclear waste management programme

19.4 Conclusions for Nuclear Waste Management Project

Both the PFA team and the CPN team have learned useful lessons and gained valuable experience in this project. The PFA team obtained an improved SADT model. They were “forced” to provide a more precise description of the behaviour of the activities in the SADT model. To accomplish this level of description the PFA team had to consider each activity in the model more carefully. As a result they discovered many ambiguities, inconsistencies, and gaps in the SADT model. The SADT model was improved and many errors were found at an early stage. Additionally, the ability to simulate the CPN models gave the PFA team new insight into the detailed behaviour of the modelled waste management programme – knowledge which it would have been impossible, or at least very difficult, to obtain directly from the non-executable SADT model.

The CPN team identified and implemented improvements in the modelling and simulation tool as a consequence of the practical work with a large and non-trivial model. We developed a technique for merging large simulation models. Furthermore, we discovered that it would be possible to make a totally automatic translation of (certain kinds) of SADT models into CPN models. This led to the creation of the work flow tool described in [40]. The tool makes the power of CPN simulation available to non-CPN experts.

An obvious extension to the work reported in this chapter would be to take also the use of resources into account. Another extension would be to perform simulations with more realistic time delays. In most of our simulations each activity was defined to use one time unit. Hence, we primarily investigated the order in which activities occurred.

References

1. T. Andersen: *Improved Methodology for the Design of Communication Protocols in Security Systems*. SECU-DES, ESSI Project 10937, 1996.
2. AT&T: *ISDN Basic Rate Interface Specification: Network Layer, Basic Voice Services*. AT&T Technical Reference Publication 8A-802-100, 1989.
3. AT&T: *ISDN Basic Rate Interface Specification: Four Supplementary Voice Services*. AT&T Technical Reference Publication 8A-802-100, 1989.
4. G. Balbo, S.C. Bruell, P. Chen, G. Chiola: *An Example of Modelling and Evaluation of a Concurrent Program Using Colored Stochastic Petri Nets: Lamport's Fast Mutual Exclusion Algorithm*. IEEE Transactions on Parallel and Distributed Systems, 3 (1992), IEEE Computer Society, 221–240. Also in [33], p. 533–559.
5. J. Berger: *Target Evaluation and Weapon Assignment Demonstrator Design*. Defence Research Establishment Valcartier, Quebec, Canada, M-3138, Unclassified, 1992.
6. J. Berger, L. Lamontagne: *A Colored Petri Net Model for a Naval Command and Control System*. In [54], p. 532–541.
7. J. Billington, M.C. Wilbur-Ham, M.Y. Bearman: *Automated Protocol Verification*. In: M. Diaz (ed.): *Protocol Specification, Testing, and Verification Vol. 5*, Elsevier Science Publishers 1986, p. 59–70.
8. S. Brandt, O.L. Madsen. *Object-Oriented Distributed Programming in Beta*. In: R. Guerraoui, O.M. Nierstrasz, and M. Riveill (eds.): *Object-Based Distributed Programming, Proceedings of ECOOP'93, Kaiserslautern 1993*, Lecture Notes in Computer Science Vol. 791, Springer-Verlag 1994, p. 185–212.
9. H. Brettschneider, H.J. Genrich, H.-M. Hanisch: *Verification and Performance Analysis of Recipe-Based Controllers by Means of Dynamic Plant Models*. In: J.C. Fransoo and W.G.M.M. Rutten (eds.): *Proceedings of 2nd International Conference on Computer Integrated Manufacturing in the Process Industries, Eindhoven 1996*, p. 128–142.
10. C. Capellmann, H. Dibold: *Petri Net Based Specifications of Services in an Intelligent Network. Experiences Gained from a Test Case Application*. In [54], p. 542–551.
11. C. Capellmann, H. Dibold: *Formal Specifications of Services in an Intelligent Network Using High-Level Petri Nets*. In: *Case Study Proceedings of the 15th International Conference on Application and Theory of Petri Nets, Zaragoza 1994*.

12. L. Cherkasova, V. Kotov, T. Rokicki: *On Net Modelling of Industrial Size Concurrent Systems*. In: Case Study Proceedings of the 15th International Conference on Application and Theory of Petri Nets, Zaragoza 1994.
13. S. Christensen, L.O. Jepsen: *Modelling and Simulation of a Network Management System Using Hierarchical Coloured Petri Nets*. In: E. Mosekilde (ed.): *Modelling and Simulation 1991*. Proceedings of the 1991 European Simulation Multiconference, Copenhagen 1991, Society for Computer Simulation 1991, p. 47–52.
14. S. Christensen, J.B. Jørgensen: *Analysing Bang & Olufsen's BeoLink Audio/Video System Using Coloured Petri Nets*. In: G. Balbo and P. Azema (eds.): *Application and Theory of Petri Nets 1997*. Proceedings of the 18th International Petri Net Conference, Toulouse 1997, Lecture Notes in Computer Science, Springer-Verlag, to appear 1997.
15. S. Christensen: *Message Sequence Charts for Design/CPN. User's Manual*. Computer Science Department, University of Aarhus, Denmark. On-line version: <http://www.daimi.aau.dk/designCPN/>.
16. CCITT: *Functional Specification and Description Language SDL*. CCITT Red Book, Vol. 6, Recommendations Z.100–Z.104, CCITT, Geneva, 1984.
17. H. Clausen, P.R. Jensen: *Validation and Performance Analysis of Network Algorithms by Coloured Petri Nets*. In [44], p. 280–289.
18. H. Clausen, P.R. Jensen: *Usage Parameter Control Algorithms in High Speed Networks*. Master's Thesis, Computer Science Department, Aarhus University, Denmark, 1993.
19. H. Clausen, P.R. Jensen: *Analysis of Usage Parameter Control Algorithms for ATM Networks*. In: S. Tohmé and A. Casaca (eds.): *Broadband Communications, II (C-24)*, Elsevier Science Publishers 1994, p. 297–310.
20. A.L. Davis: *Mayfly. A General-Purpose, Scalable, Parallel Processing Architecture*. *Journal of LISP and Symbolic Computation* 5 (1993), No. 1/2.
21. A.L. Davis, B. Coates, R. Hodgson, R. Schediwy, K. Stevens: *Mayfly System Hardware*. Hewlett-Packard Laboratories, Technical Report HPL-SAL-89-23, 1989.
22. DREV: *On the Use of Petri Nets in Naval Command and Control Systems*. Defence Research Establishment Valcartier, Quebec, Canada, Informission Ltée, Contract W7701-1-0665/01-XSK, 1992.
23. G.A. Findlow, G. Gerrand: *A Coloured Petri Net Model of ISDN Supplementary Services*. Telecom Australia Research Laboratories, Report 8115, 1992.
24. G.A. Findlow, G.S. Gerrand, J. Billington, R.J. Fone: *Modelling ISDN Supplementary Services Using Coloured Petri Nets*. Proceedings of Communications '92, Sydney, Australia, p. 37–41.
25. D.J. Floreani, J. Billington, A. Dadej: *Designing and Verifying a Communications Gateway Using Coloured Petri Nets and Design/CPN*. In [56], p. 153–171.

26. P.T. Gaughan, S. Yalamanchili: *Adaptive Routing Protocols for Hypercube Interconnection Networks*. Computer 26 (1993), 12–24.
27. H.J. Genrich, H.-M. Hanisch, K. Wöllhaf: *Verification of Recipe-Based Control Procedures by Means of Predicate/Transition Nets*. In [55], p. 278–297.
28. H.J. Genrich, R.M. Shapiro: *Formal Verification of an Arbiter Cascade*. In: K. Jensen (ed.): *Application and Theory of Petri Nets 1992*. Proceedings of the 13th International Petri Net Conference, Sheffield 1992, Lecture Notes in Computer Science Vol. 616, Springer-Verlag 1992, p. 205–223.
29. H.J. Genrich, R.M. Shapiro: *A Design of a Cascadable Nacking Arbiter*. In: Case Study Proceedings of the 14th International Conference on Application and Theory of Petri Nets, Chicago 1993. Extended version available as: Technical Report TR-93, Meta Software Corporation, 125 Cambridge Park Drive, Cambridge MA 02140, USA, 1993.
30. J. Gray (ed.): *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann Publishers, San Mateo, California, 1991.
31. S. Haddad: *A Reduction Theory for Coloured Nets*. In: G. Rozenberg (ed.): *Advances in Petri Nets 1989*, Lecture Notes in Computer Science Vol. 424, Springer-Verlag 1990, p. 209–235. Also in [33], p. 399–425.
32. P. Huber, V.O. Pinci: *A Formal Executable Specification of the ISDN Basic Rate Interface*. In: Proceedings of the 12th International Conference on Application and Theory of Petri Nets, Aarhus 1991, p. 1–21.
33. K. Jensen, G. Rozenberg (eds.): *High-Level Petri Nets. Theory and Application*. Springer-Verlag 1991.
34. J.B. Jørgensen, L.M. Kristensen: *Computer Aided Verification of Lamport's Fast Mutual Exclusion Algorithm Using Coloured Petri Nets and Occurrence Graphs with Symmetries*. Computer Science Department, Aarhus University, Denmark. Submitted to IEEE Transactions on Parallel and Distributed Systems, 1996.
35. J.B. Jørgensen, K.H. Mortensen: *Modelling and Analysis of Distributed Program Execution in Beta Using Coloured Petri Nets*. In [56], p. 249–268.
36. L. Lamport: *A Fast Mutual Exclusion Algorithm*. ACM Transactions on Computer Systems 5 (1987), 1–11.
37. O.L. Madsen, B. Møller-Pedersen, K. Nygaard. *Object-Oriented Programming in the Beta Programming Language*. ACM Press Books, Addison-Wesley, 1993.
38. D.A. Marca, C.L. McGowan: *SADT*. McGraw-Hill, New York, 1988.
39. Meta Software: *Design/IDEF User's Manual*. Meta Software Corporation, 125 Cambridge Park Drive, Cambridge MA 02140, USA, 1992.
40. Meta Software: *Work Flow Analysis. User's Manual*. Meta Software Corporation, 125 Cambridge Park Drive, Cambridge MA 02140, USA, 1994.

41. K.H. Mortensen, V.O. Pinci: *Modelling the Work Flow of a Nuclear Waste Management Program*. In [55], p. 376–395.
42. V.O. Pinci, R.M. Shapiro: *An Integrated Software Development Methodology Based on Hierarchical Colored Petri Nets*. In: G. Rozenberg (ed.): *Advances in Petri Nets 1991*, Lecture Notes in Computer Science Vol. 524, Springer-Verlag 1991, p. 227–252. Also in [33], p. 649–667.
43. V.O. Pinci: *The Shawmut Project*. In: *Case Study Proceedings of the 14th International Conference on Application and Theory of Petri Nets*, Chicago 1993.
44. *PNPM93: Petri Nets and Performance Models*. Proceedings of the 5th International Workshop, Toulouse, France 1993, IEEE Computer Society Press.
45. J.L. Rasmussen and M. Singh: *Designing and Analysing a Security System by Means of Coloured Petri Nets*. Master's Thesis, Computer Science Department, Aarhus University, Denmark, 1995.
46. J.L. Rasmussen, M. Singh: *Designing a Security System by Means of Coloured Petri Nets*. In [56], p. 400–419.
47. J.L. Rasmussen, M. Singh: *Mimic/CPN. A Graphical Animation Utility for Design/CPN*. Computer Science Department, Aarhus University, Denmark. On-line version: <http://www.daimi.aau.dk/designCPN/>.
48. E.P. Rathgeb: *Modelling and Performance Comparison of Policing Mechanisms for ATM Networks*. IEEE Journal on Selected Areas in Communications 9, (1991), 325–334.
49. T. Reenskaug et. al.: *OORASS. Seamless Support for the Creation and Maintenance of Object Oriented Systems*. Journal of Object-Oriented Programming, 1992, 27–41.
50. G. Scheschonk, M. Timpe: *Simulation and Analysis of a Document Storage System*. In [55], p. 454–470.
51. G. Scheschonk, M. Timpe: *Modelling, Simulation and Evaluation of a Large Scale Document Storage System*. In: *Case Study Proceedings of the 16th International Petri Net Conference*, Turin 1995, p. 1–27.
52. R.M. Shapiro: *Validation of a VLSI Chip Using Hierarchical Coloured Petri Nets*. Journal of Microelectronics and Reliability, Special Issue on Petri Nets, Pergamon Press, 1991. Also in [33], p. 667–687.
53. A. Valmari: *Compositionality in State Space Verification Methods*. In [56], p. 29–56.
54. M. Ajmone-Marsan (ed.): *Application and Theory of Petri Nets 1993*. Proceedings of the 14th International Petri Net Conference, Chicago 1993, Lecture Notes in Computer Science Vol. 691, Springer-Verlag 1993.
55. In: R. Valette (ed.): *Application and Theory of Petri Nets 1994*. Proceedings of the 15th International Petri Net Conference, Zaragoza 1994, Lecture Notes in Computer Science Vol. 815, Springer-Verlag 1994.

56. J. Billington and W. Reisig (eds.): *Application and Theory of Petri Nets 1996*. Proceedings of the 17th International Petri Net Conference, Osaka 1996, Lecture Notes in Computer Science Vol. 1091, Springer-Verlag 1996.

Monographs in Theoretical Computer Science – An EATCS Series

C. Calude
Information and Randomness
An Algorithmic Perspective

K. Jensen
Coloured Petri Nets
Basic Concepts, Analysis Methods
and Practical Use, Vol. 1
2nd ed.

K. Jensen
Coloured Petri Nets
Basic Concepts, Analysis Methods
and Practical Use, Vol. 2

K. Jensen
Coloured Petri Nets
Basic Concepts, Analysis Methods
and Practical Use, Vol. 3

A. Nait Abdallah
The Logic of Partial Information

Texts in Theoretical Computer Science – An EATCS Series

J. L. Balcázar, J. Díaz, J. Gabarró
Structural Complexity I
2nd ed. (see also overleaf, Vol. 22)

M. Garzon
Models of Massive Parallelism
Analysis of Cellular Automata
and Neural Networks

J. Hromkovič
Communication Complexity
and Parallel Computing

A. Leitsch
The Resolution Calculus

A. Salomaa
Public-Key Cryptography
2nd ed.

K. Sikkel
Parsing Schemata
A Framework for Specification
and Analysis of Parsing Algorithms

Former volumes appeared as EATCS Monographs on Theoretical Computer Science

Vol. 5: W. Kuich, A. Salomaa
Semirings, Automata, Languages

Vol. 6: H. Ehrig, B. Mahr
Fundamentals of Algebraic Specification 1
Equations and Initial Semantics

Vol. 7: F. Gécseg
Products of Automata

Vol. 8: F. Kröger
Temporal Logic of Programs

Vol. 9: K. Weihrauch
Computability

Vol. 10: H. Edelsbrunner
Algorithms in Combinatorial Geometry

Vol. 12: J. Berstel, C. Reutenauer
Rational Series and Their Languages

Vol. 13: E. Best, C. Fernández C.
Nonsequential Processes
A Petri Net View

Vol. 14: M. Jantzen
Confluent String Rewriting

Vol. 15: S. Sippu, E. Soisalon-Soininen
Parsing Theory
Volume I: Languages and Parsing

Vol. 16: P. Padawitz
Computing in Horn Clause Theories

Vol. 17: J. Paredaens, P. DeBra, M. Gyssens,
D. Van Gucht
The Structure of the
Relational Database Model

Vol. 18: J. Dassow, G. Páun
Regulated Rewriting
in Formal Language Theory

Vol. 19: M. Tofte
Compiler Generators
What they can do, what they might do,
and what they will probably never do

Vol. 20: S. Sippu, E. Soisalon-Soininen
Parsing Theory
Volume II: LR(k) and LL(k) Parsing

Vol. 21: H. Ehrig, B. Mahr
Fundamentals of Algebraic Specification 2
Module Specifications and Constraints

Vol. 22: J. L. Balcázar, J. Díaz, J. Gabarró
Structural Complexity II

Vol. 24: T. Gergely, L. Úry
First-Order Programming Theories

R. Janicki, P. E. Lauer
Specification and Analysis
of Concurrent Systems
The COSY Approach

O. Watanabe (Ed.)
Kolmogorov Complexity
and Computational Complexity

G. Schmidt, Th. Ströhlein
Relations and Graphs
Discrete Mathematics for Computer Scientists

S. L. Bloom, Z. Ésik
Iteration Theories
The Equational Logic of Iterative Processes